Programming Research Group

THEOREM PROVING
IN
HIGHER ORDER LOGICS:
EMERGING TRENDS PROCEEDINGS

Joe Hurd
Edward Smith
Ashish Darbari (Eds.)

PRG-RR-05-02

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Preface

This volume is the Emerging Trends proceedings of the *18th International Conference on Theorem Proving in Higher Order Logics* (TPHOLs 2005), which was held during 22–25 August 2005 in Oxford, UK. 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 2005 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.

The organizers are grateful to Wolfgang Paul and Andrew Pitts for agreeing to give invited talks at TPHOLs 2005.

The TPHOLs conference traditionally changes continents each year to maximize the chances that researchers from around the world can attend. Starting in 1993, the proceedings of TPHOLs and its predecessor workshops have been published in the Springer Lecture Notes in Computer Science series:

1993 (Canada)	Vol. 780	2000 (USA)	Vol. 1869
1994 (Malta)	Vol. 859	2001 (UK)	Vol. 2152
1995 (USA)	Vol. 971	2002 (USA)	Vol. 2410
1996 (Finland)	Vol. 1125	2003 (Italy)	Vol. 2758
1997 (USA)	Vol. 1275	2004 (USA)	Vol. 3223
1998 (Australia)	Vol. 1479	2005 (UK)	Vol. 3603
1999 (France)	Vol. 1690		

Finally, we thank our sponsors: Intel Corporation and the EPSRC UK Network in Computer Algebra.

July 2005 Joe Hurd, Edward Smith and Ashish Darbari

Programme Committee

Mark Aagaard (Waterloo)
Clark Barrett (NYU)
David Basin (ETH Zürich)
Yves Bertot (INRIA)
Ching-Tsun Chou (Intel)
Thierry Coquand (Chalmers)
Amy Felty (Ottawa)
Jean-Christophe Filliâtre (Paris Sud)
Jacques Fleuriot (Edinburgh)
Jim Grundy (Intel)
Elsa Gunter (UIUC)
John Harrison (Intel)
Jason Hickey (Caltech)
Peter Homeier (US DoD)
Joe Hurd (Oxford)
Paul Jackson (Edinburgh)
Thomas Kropf (Tübingen & Bosch)
Pete Manolios (Georgia Tech)
John Matthews (Galois)
César Muñoz (Nat. Inst. Aerospace)
Tobias Nipkow (München)
Sam Owre (SRI)
Christine Paulin-Mohring (Paris Sud)
Lawrence Paulson (Cambridge)
Frank Pfenning (CMU)
Konrad Slind (Utah)
Sofiène Tahar (Concordia)
Burkhart Wolff (ETH Zürich)

Table of Contents

Towards Automated Verification of Database Scripts

A. Azurat[1], I.S.W.B. Prasetya[2], T.E.J. Vos[3], H. Suhartanto[1], B. Widjaja[1],
L.Y. Stefanus[1], R. Wenang[1], S. Aminah[1], and J. Bong[1]

[1] Fakultas Ilmu Komputer, Universitas Indonesia, Indonesia. *
{ade,heru,bela,yohanes,wenang,aminah,jimmy}@cs.ui.ac.id
[2] Informatiekunde en Informatica Instituut, Universiteit Utrecht, Netherlands.
wishnu@cs.uu.nl
[3] Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, Spain.
tanja@iti.upv.es

Abstract.
The article reports on our preliminary research activities towards the verification of database transaction scripts. It gives a first order specification language for database transaction scripts and its accompanying logic. The logic is expressed in terms of weakest pre-condition rules and is quite simple and intuitive. The logic is sound, and, if the underlying basic expression language is limited, specifications in our language are decidable. Tables in a database are usually guarded by various integrity constraints. At the moment the logic only supports single attribute primary key constraint. The language has been embedded in HOL theorem prover. A verification case study on Student Entrance Test Application is described.

1 Introduction

Many organizations, like banks and ministries, run mission critical data processing applications that must be highly reliable. Unfortunately, however, in practice there are only few organizations that seriously verify and test their code in order to assure a certain level of quality. Moreover, programmers and managers do not consider verification and testing to be as important as coding, and most of the time they consider it as something you do if there is some time and budget left. Programming languages also contribute to the fact that data processing applications are not always as reliable as they should be. Modern database applications are built with high-level languages like PL/SQL, that, while providing good abstraction, also offer lower level programming constructs for optimization. Although, this means that performance can be improved, the resulting code degrades in reliability and readability, and the cost of debugging and maintenance

* Supported by the Menristek-RUTI II grant 2003-2005.

can increase. In addition, most languages do not treat verification and validation (V&V) as an integral part of programming, which translates into the current attitude of engineers and managers towards testing.

This paper discusses our ideas for a database transaction scripting language. We consider a simple but still quite expressive database transaction scripting language, which we will refer to as $\mathcal{L}ingu$. $\mathcal{L}ingu$ is a light weight high level language to program data transformations on databases. Optimization features, usually present in other languages, are absent for example, $\mathcal{L}ingu$ does not have arrays. This forces the programmers to keep their code abstract.

Even when compared to the abstract part of other languages, $\mathcal{L}ingu$ is small. In $\mathcal{L}ingu$ we will not be able to write all kind of database constraints and operations like those available in PL/SQL. For example, $\mathcal{L}ingu$ does not have a String[32] type and the sort by modifier. However, $\mathcal{L}ingu$ should provide enough expressibility to program a large class of useful data transformations. Keeping the language small simplifies $\mathcal{L}ingu$ s internal logic and the verification of $\mathcal{L}ingu$ programs.

The $\mathcal{L}ingu$ language and logic has been externally[1] embedded in the HOL theorem prover[2]. This allows us to reason about a $\mathcal{L}ingu$ programs in the HOL environment as well as the availability of an ML implementation of a $\mathcal{L}ingu$ verification condition generator.

For illustration purposes, we have conducted a case study with a Student Entrance Test (SET) application. Based on the positive results of this case study, we believe that the $\mathcal{L}ingu$, in the future, could contribute to more reliable database applications.

The article is organized as follows. Section 2 and 3 summarize the $\mathcal{L}ingu$ scripting and specification language, Section 4 summarizes the logic, Section 5 describes the semantics. We will briefly discuss the soundness and decidability of the logic in Section 6. A very brief possible optimization for the verification condition generator is mentioned in Section 7. The case study is explained in Section 8 . Section 9 briefly describes how the language is used on the case study as HOL embedding. Section 10, finally, contains some discussions on related and future work.

2 The Scripting Language

An example of a script is shown below:

Example 1. $\mathcal{L}ingu$ Code of *Safe* Module

```
safe ( HealthyAFormTab : AnswerFormTable,
   SolutionsTab : SolutionTable,
   MasterTab : RegistrationTable,
   PassTab : RegistrationTable,
  ): Bool
var r : Bool;
```

```
{
  r := forall (0<=i /\ i < #d.PassTab)
              (find r<-d.MasterTab
                    where r.ID = id  found T otherwise F
                /\
               find r<-d.HealthyFormTab
                    where r.ID = id found T otherwise F
               );
  return r
}
```

Safe module is used to check and guarantee that the evaluation process of answer forms will not pass a person that is illegal. Illegal term refers to a person that does not take the test or is not officially registered in the examinee list.

A script starts with a header, consisting of the script's name followed by a list of formal parameters and their types. Formal parameters can only be of primitive or record type. All parameters are passed by value. After the header, there is a list of declarations of local variables, followed by a series of instructions[4] enclosed within brackets { }. The syntax of the possible instructions is given below.

$$\begin{array}{lll}
Instr & \rightarrow & \textsf{skip} \\
 & & \mid Var \ := Tpred \\
 & & \mid \textsf{if } Bexpr \textsf{ then } Instr \textsf{ else } Instr \\
 & & \mid \{Instr; \ldots ;Instr\} \\
 & & \mid Var := SelectExpr \\
 & & \mid \textsf{insert } Bexpr \textsf{ to } Var \\
 & & \mid \textsf{delete } Var \textsf{ from } Var \textsf{ where } Tpred \\
 & & \mid \textsf{update } Var \textsf{ in } Var \textsf{ to } BExpr \textsf{ where } Tpred
\end{array}$$

$$\begin{array}{ll}
SelectExpr & \rightarrow \textsf{select } Vars \rightarrow Bexpr \textsf{ from } DomainExpr \textsf{ where } Tpred \\
Vars & \rightarrow Var \mid (Var, \ldots, Var) \\
DomainExpr & \rightarrow Var \mid Var \times \ldots \times Var
\end{array}$$

Note that $\mathcal{L}ingu$ does not have an imperative loop (in comparison, neither does SQL without a PL extension). In this paper, we will not concern ourselves with the question of how to efficiently implement $\mathcal{L}ingu$. In principle, however, it should map easily to SQL, of which many efficient implementations exist.

The notation $t_1 \times t_2$ in a select expressions stands for a full cartesian product of t_1 and t_2.

In the above grammar, *Tpred* stands for a predicate over a table. The grammar of *Tpred* is presented in the next section. *Bexpr* stands for a basic expression; it is an expression that only concerns values of primitive types and record types. Available primitive types are limited to Bool, numerical types, and String.

[4] Notice the use of instruction `find` which is not mentioned in the grammar. It is just a syntactic sugar of $\exists$ in *Tpred*.

$$\begin{array}{lll} Bexpr & \rightarrow & \mathit{Literal\ of\ primitive\ type} \\ & & \mid \mathit{Var\ of\ primitive\ or\ record\ type} \\ & & \mid \{\ AttributeName{=}Bexpr, \ldots, AttributeName{=}Bexpr\ \} \\ & & \mid Bexpr.AttributeName \\ & & \mid Bexpr\ =\ Bexpr \\ & & \mid Bexpr\ NumOp\ Bexpr \\ & & \mid Bexpr\ BoolOp\ Bexpr \\ & & \mid \neg Bexpr \\ & & \mid \mathsf{if}\ Bexpr\ \mathsf{then}\ Bexpr\ \mathsf{else}\ Bexpr \end{array}$$

Available numerical operators are limited to $+, -, <, \leq, >, \geq$, and multiplication with constants. Available Boolean operators are $\wedge, \vee, \Rightarrow$.

3 The Specification Language

We will use Hoare triples to specify various properties that a $\mathcal{L}ingu$ script must satisfy. The pre- and post-conditions of a Hoare triple are the aforementioned table predicates $Tpred$, which are first order formulas over $Bexpr$:

$$\begin{array}{lll} Tpred & \rightarrow & Bexpr\ \ \text{(of Bool type)} \\ & & \mid \forall Rvar{::}Tvar.\ \ Tpred \\ & & \mid \exists Rvar{::}Tvar.\ \ Tpred \\ & & \mid \neg Tpred \\ & & \mid Tpred\ BoolOp\ Tpred \end{array}$$

Where, $Rvar$ is a (bound) variable of a record type and $Tvar$ is a variable of a table type. A predicate $\forall r{::}t.\ P$ means that all records r in the table t satisfies P. The meaning of $\exists r{::}t.\ P$ is analogous.

Some Additional Notation

We write $r \in t$ to mean $\exists r'{::}t.\ r' = r$. We write $\mathsf{if}\ P\ \mathsf{then}\ Q\ \mathsf{else}\ R$ to mean $(P \Rightarrow Q) \wedge (\neg P \Rightarrow R)$. We abbreviate $\forall r{::}t.\ (\forall s{::}t.\ P)$ by $\forall r, s{::}t.\ P$. An example of a script property is below:

$$\{r \in t0\ \vee\ r \in tP\}\ \ collectPass(n)\ \ \{r \in t0\ \vee\ r \in tP\}$$

which says that with respect to the tables $t0$ and tP, the script `collectPass` can only move (or duplicate) records among them.

4 The Logic

Let U be a $\mathcal{L}ingu$ script. A specification for U can be expressed in terms of a Hoare triple specification of U's instructions, with, when necessary, additional assignments to auxiliary variables. Consider a specification $\{P\}\ S\ \{Q\}$ where S is an instruction. As usual, $\mathsf{wp}\ S\ Q$ denotes the weakest pre-condition of S with respect to Q. Since the language $\mathcal{L}ingu$ does not contain imperative loops nor recursions, we can construct $\mathsf{wp}\ S\ Q$. Consequently it is sufficient to prove that P implies $\mathsf{wp}\ S\ Q$.

For the first four kinds of instructions (i.e. skip, assignment to a variable of a primitive or record type, conditional, and sequence) wp can be constructed in the standard way.

The remaining $\mathcal{L}ingu$ instructions are called *table instruction*, because they may change a table. These instructions behave as assignments in that they change some rows in a table t.

Given a post-condition Q, the standard way of dealing with an assignment $t := e$ is to replace t in Q with e. In $\mathcal{L}ingu$, however, we cannot do that since the variable t can occur in a quantified expression, such as $\forall r{::}t.\ P$ and $\exists r{::}t.\ P$ (actually, this is the only way a variable of type table can occur in a specification). We cannot replace t in those expressions with an arbitrary e because $\mathcal{L}ingu$'s grammar requires that only a variable of a table type can occur after ::. As we will see later in Section 6, this restriction has something to do with the decidability of $\mathcal{L}ingu$. The trick to get around this is to use a notion of 'substitution' that is a bit more complicated than usual and rewrite t's enclosing expression instead.

We will represent a 'substitution' on t with a pair (t, π) where π is a function that will replace $t's$ enclosing expression with something else. The function subst below will recursively apply π on a given target $Tpred$.

Definition 1. *Basic Substitution*

$$\begin{array}{ll}
\mathsf{subst}\ (t,\pi)\ bexpr & = bexpr \\
\mathsf{subst}\ (t,\pi)\ (\neg R) & = \neg(\mathsf{subst}\ (t,\pi)\ R) \\
\mathsf{subst}\ (t,\pi)\ (R_1\ BoolOp\ R_2) & = (\mathsf{subst}\ (t,\pi)\ R_1)\ BoolOp\ (\mathsf{subst}\ (t,\pi)\ R_2) \\
\mathsf{subst}\ (t,\pi)\ (\forall r'{::}t.\ R) & = \pi\ (\forall r'{::}t.\ \mathsf{subst}\ (t,\pi)\ R) \\
\mathsf{subst}\ (t,\pi)\ (\exists r'{::}t.\ R) & = \pi\ (\exists r'{::}t.\ \mathsf{subst}\ (t,\pi)\ R)
\end{array}$$

and if t' and t are not syntactically equal:

$$\begin{array}{l}
\mathsf{subst}\ (t,\pi)\ (\forall r'{::}t'.\ R) = (\forall r'{::}t'.\ \mathsf{subst}\ (t,\pi)\ R) \\
\mathsf{subst}\ (t,\pi)\ (\exists r'{::}t'.\ R) = (\forall r'{::}t'.\ \mathsf{subst}\ (t,\pi)\ R)
\end{array}$$

In Figure 1 we list the usual equations for constructing the wp of $\mathcal{L}ingu$'s table instructions. We assume here that the target table has no constraint.

When the target table has integrity constraints calculating wp becomes more complicated. First, let us take the convention that $\mathcal{L}ingu$ instructions will simply perform a skip if it would otherwise cause a constraint on the target table to be violated. For each table instruction S and a given target table t, we can formalize what we call a *safety condition*, a condition under which S can be executed safely without violating any constraint on t. Notice that for sophisticated constraints, the formalization as a safety condition can be complicated and thus expensive to

Definition 2.

$$\begin{aligned}
&\mathsf{wp}\ (t := \mathsf{select}\ (r) \to e\ \mathsf{from}\ u\ \mathsf{where}\ P)\ Q \quad = \quad \mathsf{subst}\ (t, \pi_S)\ Q \\
&\pi_S\ (\forall r'{::}t.\ R) = (\forall r{::}u.\ P \Rightarrow R[e/r']) \\
&\pi_S\ (\exists r'{::}t.\ R) = (\exists r{::}u.\ P \wedge R[e/r'])
\end{aligned}$$

Definition 3.

$$\begin{aligned}
&\mathsf{wp}\ (t := \mathsf{select}\ (r, s) \to e\ \mathsf{from}\ u \times v\ \mathsf{where}\ P)\ Q \quad = \quad \mathsf{subst}\ (t, \pi_{S2})\ Q \\
&\pi_{S2}\ (\forall r'{::}t.\ R) = (\forall r{::}u.\ (\forall s{::}v.\ P \Rightarrow R[e/r'])) \\
&\pi_{S2}\ (\exists r'{::}t.\ R) = (\exists r{::}u.\ (\exists s{::}v.\ P \wedge [e/r'])
\end{aligned}$$

Definition 4.

$$\begin{aligned}
&\mathsf{wp}\ (\mathsf{insert}\ r\ \mathsf{to}\ t)\ Q \quad = \quad \mathsf{subst}\ (t, \pi_I)\ Q \\
&\pi_I\ (\forall r'{::}t.\ R) = R[r/r'] \wedge (\forall r'{::}t.\ R) \\
&\pi_I\ (\exists r'{::}t.\ R) = R[r/r'] \vee (\exists r'{::}t.\ R)
\end{aligned}$$

Definition 5.

$$\begin{aligned}
&\mathsf{wp}\ (\mathsf{delete}\ r\ \mathsf{from}\ t\ \mathsf{where}\ P)\ Q \quad = \quad \mathsf{subst}\ (t, \pi_D)\ Q \\
&\pi_D\ (\forall r'{::}t.\ R) = (\forall r'{::}t.\ \neg P[r'/r] \Rightarrow R) \\
&\pi_D\ (\exists r'{::}t.\ R) = (\exists r'{::}t.\ \neg P[r'/r] \wedge R)
\end{aligned}$$

Definition 6.

$$\begin{aligned}
&\mathsf{wp}\ (\mathsf{update}\ r\ \mathsf{in}\ t\ \mathsf{to}\ e\ \mathsf{where}\ P)\ Q \quad = \quad \mathsf{subst}\ (t, \pi_U)\ Q \\
&\pi_U\ (\forall r'{::}t.\ R) = (\forall r{::}t.\ \mathsf{if}\ P\ \mathsf{then}\ R[e/r']\ \mathsf{else}\ R[r/r']) \\
&\pi_U\ (\exists r'{::}t.\ R) = (\exists r{::}t.\ (P \wedge R[e/r']) \vee (\neg P \wedge R[r/r']))
\end{aligned}$$

Fig. 1. wp *of the table instructions when the target table has no constraint.*

verify. In Section 7 we will discuss some conditions that can be checked statically and from which we can infer whether a safety condition can be dropped, or at least simplified.

In this paper we will only consider one sort of constraint: we will allow tables to have a single attribute as its primary key. If K is an attribute of a table t, we write K key t to mean that K is the primary key of t. Formally:

$$K\ \mathsf{key}\ t \quad = \quad (\forall r, r'{::}T.\ r.K = r'.K \ \Rightarrow\ r = r') \tag{1}$$

Note that this notion of primary key allows duplicate rows. It is necessarily to catch the possible error of duplicate ID in the input data which happen in the case study.

For every table t with primary key K which is manipulated by a script U, the pre-condition in the specification of U should be strengthened with K key t.

Only the delete instruction will not violate a primary key constraint, so we do not have to change its wp equation. The other table instructions may violate such a constraint if they don't skip. We alter their wp equations, as shown in Figure 2. The equations in the figure assumes that the target table t has K as its primary key.

Definition 7.

$$\begin{array}{l}
\mathsf{wp}\ (t := \mathsf{select}\ r \to e\ \mathsf{from}\ u\ \mathsf{where}\ P)\ Q \quad = \quad \mathsf{if}\ \ Safe_S \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\ \ \mathsf{subst}\ (t, \pi_S)\ Q \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ Q \\
\textit{where} \\
\quad Safe_S \quad = \quad (\forall r, r'{::}u.\ P \wedge P[r'/r] \wedge (e.K = e[r'/r].K) \ \Rightarrow\ e = e[r'/r]))
\end{array}$$

Definition 8.

$$\begin{array}{l}
\mathsf{wp}\ (t := \mathsf{select}\ (r, s) \to e\ \mathsf{from}\ u \times v\ \mathsf{where}\ P)\ Q \quad = \quad \mathsf{if}\ \ Safe_{S2} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\ \ \mathsf{subst}\ (t, \pi_{S2})\ Q \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ Q \\
\textit{where} \\
\quad Safe_{S2} \quad = \quad (\forall r, r'{::}u.\ (\forall s, s'{::}v.\ \ P \wedge P[r'/r] \wedge (e.K = e[r'/r, s'/s].K) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad e = e[r'/r, s'/s]))
\end{array}$$

Definition 9.

$$\begin{array}{l}
\mathsf{wp}\ (\mathsf{insert}\ r\ \mathsf{to}\ t)\ Q \quad = \quad \mathsf{if}\ \ Safe_I\ \ \mathsf{then}\ \ \mathsf{subst}\ (t, \pi_I)\ Q\ \ \mathsf{else}\ \ Q \\
\textit{where} \\
\quad Safe_I \ = \ (\forall r'{::}t.\ r'.K \neq r.K)
\end{array}$$

Definition 10.

$$\begin{array}{l}
\mathsf{wp}\ (\mathsf{update}\ r\ \mathsf{in}\ t\ \mathsf{to}\ e\ \mathsf{where}\ P)\ Q \quad = \quad \mathsf{if}\ \ Safe_U\ \ \mathsf{then}\ \ \mathsf{subst}\ (t, \pi_U)\ Q\ \ \mathsf{else}\ \ Q \\
\textit{where} \\
\quad Safe_U \quad = \quad (\forall r, r'{::}t.\ \ (P \wedge P[r'/r] \wedge (e.K = e[r'/r].K) \ \Rightarrow\ e = e[r'/r]) \\
\qquad\qquad\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad\qquad\qquad (P \wedge \neg P[r'/r] \wedge (e.K = r'.K) \ \Rightarrow\ e = r') \\
\qquad\qquad\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad\qquad\qquad (\neg P \wedge P[r'/r] \wedge (r.K = e[r'/r].K) \ \Rightarrow\ r = e[r'/r]) \\
\qquad\qquad\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad\qquad\qquad (\neg P \wedge \neg P[r'/r] \wedge (r.K = r'.K) \ \Rightarrow\ r = r'))
\end{array}$$

Fig. 2. wp *when the target table has a primary key constraint.*

5 Semantics

We will use sets of records to express the semantic of tables. Consequently, the semantics of quantified table predicates in terms of sets is straightforward:

$$|[(\forall r{::}t.\ P)]| \ = \ (\forall r.\ r \in t \Rightarrow P) \qquad (2)$$

$$|[(\exists r{::}t.\ P)]| \ = \ (\exists r.\ r \in t \wedge P) \qquad (3)$$

The semantics of the $\mathcal{L}ingu$ instructions will be expressed in terms of a standard simple command language (without table instructions).

It is sufficient to give the semantics of the $\mathcal{L}ingu$ table instructions, which are given below. For a target table t that has primary key K, the definitions of $Safe_S$, $Safe_I$, and $Safe_U$ can be found in Figure 2. If t has no primary key, the $Safe$ predicates are always true.

$$\begin{array}{l} |[\,t := \mathsf{select}\ r \rightarrow e\ \mathsf{from}\ u\ \mathsf{where}\ P\,]| \\ = \\ \mathsf{if}\ |[Safe_S]|\ \mathsf{then}\ t := \{e \mid r \in u \wedge P\}\ \mathsf{else\ skip} \end{array} \qquad (4)$$

$$|[\mathsf{insert}\ r\ \mathsf{to}\ t]| \ = \ \mathsf{if}\ |[Safe_I]|\ \mathsf{then}\ t := t \cup \{r\}\ \mathsf{else\ skip} \qquad (5)$$

$$|[\mathsf{delete}\ r\ \mathsf{from}\ t\ \mathsf{where}\ P]| \ = \ t := \{r \mid r \in t \wedge \neg P\} \qquad (6)$$

$$\begin{array}{l} |[\,\mathsf{update}\ r\ \mathsf{in}\ t\ \mathsf{to}\ e\ \mathsf{where}\ P\,]| \\ = \\ \mathsf{if}\ |[Safe_U]|\ \mathsf{then}\ t := \{\mathsf{if}\ P\ \mathsf{then}\ e\ \mathsf{else}\ r \ \mid \ r \in t\}\ \mathsf{else\ skip} \end{array} \qquad (7)$$

6 Soundness and Decidability

Theorem 1. *$\mathcal{L}ingu$ Logic is Sound.*

Proof: We only need to focus on the wp equations of the table instructions, since these are non-standard. These equations can be quite straight forwardly verified against the given semantic of $\mathcal{L}ingu$.

Theorem 2. *$\mathcal{L}ingu$ specifications are decidable.*

Proof: Notice that the wp equations of $\mathcal{L}ingu$ in Figure 2 produce formulas which are within the syntax of *Tpred*. Formulas from *Tpred* are first order formulas over atoms of primitive types or record types. *Tpred* only allows records to be compared for equality, which can be reduced to equalities on atoms of primitive types. Atoms of string types can only be compared for equalities, which is decidable. Atoms of numeric types can be compared using standard numeric relations like $<$ and $\leq$. These are decidable if we do not include multiplication of variables. It explains why we restrict the expression on table expression as mentioned in Section 4.

7 Optimizations

Below we give some conditions that will allow the $Safe$ predicates in the wp of the table instructions to be dropped. The optimizations are limited to work on a primary key constraint. More precisely, we assume the target table t has only one constraint, namely one attribute K as its primary key.

1. $\mathsf{update}\ r\ \mathsf{in}\ t\ \mathsf{to}\ e\ \mathsf{where}\ P$ is safe if e does not modify K.
2. $\mathsf{update}\ r\ \mathsf{in}\ t\ \mathsf{to}\ e\ \mathsf{where}\ r = r'$ is a single update. It affects only r' (if it is in t). This instruction is safe if $\forall r{::}t.\ r \neq r' \Rightarrow r.K \neq e.K$.
3. $t := \mathsf{select}\ r \rightarrow e\ \mathsf{from}\ u\ \mathsf{where}\ P$ is safe if K is also a primary key of u and if e does not change its value.
4. $t := \mathsf{select}\ (r, s) \rightarrow e\ \mathsf{from}\ u \times v\ \mathsf{where}\ P$ such that e copies the value of K from u. The instruction is safe if K is also a primary key of u. The situation if e copies K from v is analogous.

8 Case Study

We will describe a case study to show how our ideas can be applied to a real database application. We use the Student Entrance Test (SET) application as the case study for $\mathcal{L}ingu$ development. It is a critical application that involves vast collections of data, and thus heavily relies on database processing. And because of its nature of importance, especially the examinees, then it is mandatory that this SET application processes are fully verified.

SET in Indonesia is named SPMB[3], but for further discussion we will use the term SET to avoid confusion. Computerized SET processing in Indonesia has been held for about 45 years started from the early 60s. It has been an important national annual event. For most of the high school students, this test result will determine their future. Either they will be accepted to go to their preferred prestigious university or get rejected. Lately, the number of forms to be evaluated is on the magnitude of 300.000. The evaluation process takes approximately one month until all result can be delivered. The evaluation result will be ranked and placed on universities available seats.

During the evaluation process, a lot of things can happen with the forms that will affect the outcome of its result. For example, it can happen that the examinee writes the incorrect identity number or name on the form, or the examinee forgot to submit all set of answer forms. When these things happen, it can cause the test result to fail even though the answer is mostly correct. Worse scenarios occur when the examinee writes a wrong identity number on its form that turns out to be the identity number of another examinee. Both candidates could then be automatically excluded from the evaluation. Cases like these are the kind that we are trying to avoid by identifying it sooner and handle it accordingly.

We believe that verifying the described SET application with $\mathcal{L}ingu$ can significantly reduce the kind of errors as stated above. Verification of the application involves writing a specification that lays down the rules that the application should obey. By giving strict specification, we can anticipate and investigate potential problems that could degrade the credibility of SET's results.

Currently, we have identified 7 potential problems that could arise in this SET case study. We will use the term critical modules to refer to these potential problem areas. It is by no means that the set of critical modules identified are complete, but it is sufficient enough to show how far can *𝓛ingu* automatically verify this case study. The critical modules identified are: (1) answer evaluation, (2) unknown id check, (3) double id check, (4) safeness check, (5) fairness check, (6) completeness check, and (7) consistency check. In this paper we will show the work on (4) safeness check as an example of verification on the critical modules. The code can be seen in Example 1.

9 Embedding on HOL

We use our *𝓛ingu* embedding in the HOL theorem prover [2] to help us verify the *𝓛ingu* code. The HOL code for check safeness critical module of Example 1 is shown below.

Example 2. HOL code of *Safe* Module

```
safe ( HealthyAFormTab : AnswerFormTable set,
     SolutionsTab : SolutionTable set,
     MasterTab : RegistrationTable set,
     PassTab : RegistrationTable set,
     val : bool
    )
    (dummy)
     =

     let val1 = forall PassTab
                      (\r. (exists MasterTab (\t. r.ID = t.ID))
                      /\
                     (exists HealthyAFormTab (\y. r.ID = y.ID)))
     in
     val ::= val1
```

Safe takes 5 arguments, HealthyAFormTab, SolutionsTab, MasterTab, PassTab, and val. For all records on PassTab, if there exist a record on MasterTab that have the same ID, and if there exist a record on HealthyAFormTab that have the same ID, then the it will return a true value. This value is returned to be compared with existing specification for this critical module.

The specification of check safeness critical module is shown below.

Example 3. Specification

```
(* pre:    *)    (PassTab = {}) /\ (val = F),

(* script: *)    safe(HealthyAFormTab,SolutionsTab,MasterTab,
                      PassTab,val)
                     (dummy),

(* post:   *)    val = forall PassTab
                         (\r. exists MasterTab (\t. r.ID = t.ID)
                          /\
                          exists HealthyAFormTab (\y. r.ID = y.ID))
```

This specification is a Hoare logic triplet, where the process is verified by the precondition and postcondition defined by the user. The precondition of this critical module is that the PassTab is in empty state, and the `val` value is false. Then after the process is run, the expected postcondition is that the value of val is true if for every records in PassTab has its clone on MasterTab and HealthyAFormTab.

The specification above then will be solved by using HOL theorem prover. The proof code for check safeness critical module is shown below.

Example 4. Proof Script

```
val cert_safe = certify (spec_safe, NORMALIZE_TAC [I_THM]
                                    THEN PROVE_TAC []     );
```

The verification above uses normalization tactic with identity theorem which basically unfolds some definitions and applies standard reduction. The resulted formulas are solved by HOL theorem prover automatic theorem solver. The above script generates a theorem which can be used as the certificate of the proven module.

We have successfully test the other six critical modules with the same step as sample above. The specifications are unique for each critical module, and the verification steps are done with little or no modification from the verification code shown on the sample. It is caused by the fact that we only use small and simplified rules for the critical modules. Simplification is necessary to achieve our current objective: showing how far $\mathcal{L}ingu$ verification can be done in an automatically fashion.

10 Concluding remarks

10.1 Related Works

One of the earliest formal approaches to database application verification was done by Barros [4]. It gives a complete description of a general method for the specification of a relational database using Z. It also gives a description of a set of rules to translate such a Z specification into a relational DBMS called DBPL implementation.

The work of Laleau and Mammar [5, 6] provides an approach that enables the elaboration of an automatic prover dedicated to the refinement of B specification into a relational database implementation. A set of generic refinement rules are described that take into account both data and operations. The approach is based on the strategy of proof reuse and on the specific characteristic of such application.

The work of Locuratolo and her collaborators [7] deals with refinement for object oriented database design.

10.2 Future Work

The logic as it is, only supports a very simple kind of table constraints. While it is possible to formalize more sophisticated kinds of constraints, we believe it is more useful to focus on identifying conditions, as in Section 7 that can lead to the elimination, or at least simplification, of the corresponding safety conditions.

The certification as shown in Example 4 can be used for verification of procedure call as part of modular verification. The snapshot version of the HOL code (which is available by the authors) is capable to verify script with procedure call. Complete description and reasoning on the matter is still in progress.

Validation will be an important concept in $\mathcal{L}ingu$. A $\mathcal{L}ingu$ method can be equipped with validation scripts. As the name suggests, these are scripts used to test the class. It is a combination of formal verification and testing. The test requires a built-in feature to generate random inputs and to report the test results. The work on validation and random input generator are still in progress.

There is no $\mathcal{L}ingu$ compiler nor $\mathcal{L}ingu$ interpreter yet. The embedded code of $\mathcal{L}ingu$ in the HOL environment can also be mechanized together with the compiler which may bring an important topic of embedded verification in compilation.

References

1. Azurat, A., Prasetya, I.: A survey on embedding programming logics in a theorem prover. Technical Report UU-CS-2002-007, Inst. of Information and Comp. Science, Utrecht Univ. (2002) Download: `www.cs.uu.nl/staff/wishnu.html`.
2. Gordon, M.J., Melham, T.F.: Introduction to HOL. Cambridge University Press (1993)
3. UI, P.I.K.: Seleksi penerimaan mahasiswa baru (spmb): Modul seleksi nasional. (2002)
4. Barros, R.: Deriving relational database programs from formal specifications. In: Proceedings of 2nd Int. Conf. FME'94, Springer-Verlag, LNSC 873, Barcelona, Spain, Oct 94. (1994)
5. Laleau, R., Mammar, A.: A generic process to refine a b specification into a relational database implementation. In: Proceedings of Int. Conf. ZB2000, Springer-Verlag, LNCS 1878, York. 2000. (2000)

6. Mammar, A., Laleau, R.: Design of an automatic prover dedicated to the refinement of database applications. In Araki, K., Gnesi, S., Mandrioli, D., eds.: Proceedings of Int. Conf. FME 2003, Springer-Verlag, LNCS 2805, France. 2000. (2003) 835–853
7. Locuratolo, E., Matthews, B.M.: Formal development of databases in asso and b. In J.Wing, Woodcock, Davies, J., eds.: Proceedings of FM 99 Formal Methods,LNCS 1708. (1999) 388–410

Quantitative Temporal Logic Mechanized in HOL

Orieta Celiku

Åbo Akademi University and Turku Centre for Computer Science
Lemminkäisenkatu 14 A, 20520 Turku, Finland

Abstract. The paper describes an implementation in the HOL theorem prover of the *quantitative Temporal Logic* (*qTL*) of Morgan and McIver [18, 14]. *qTL* is a branching-time temporal logic suitable for reasoning about probabilistic nondeterministic systems. The interaction between probabilities and nondeterminism, which is generally very difficult to reason about, is handled by interpreting the logic over real- rather than boolean-valued functions. In the *qTL* framework many laws of standard branching-time temporal logic generalize nicely giving access to a number of logical tools for reasoning about quantitative aspects of randomized algorithms.

1 Introduction

Randomization is very useful for improving algorithms' efficiency and solving problems where standard methods fail, but reasoning about randomized algorithms is notoriously difficult. As a result the interest in the computer-aided verification of randomized algorithms has been increasing, both in the model-checking as well as in the theorem-proving communities. Recent work on using theorem provers for such verifications includes Hurd et al.'s [9] mechanization in HOL [5] of Morgan's *probabilistic Guarded Command Language* (*pGCL*) [19] and its associated program logic [21]. We extend this work with the mechanization of the *quantitative Temporal Logic* (*qTL*) — the probabilistic generalization of temporal logic — and its associated algebra [18, 14].

Our interest in the mechanization of *qTL* is several-fold. To start with, *pGCL* and *qTL* provide a unified framework in which to model, specify temporal properties of, and reason about probabilistic systems. The properties that can be specified and verified are *quantitative* and thus very general. For example, one can reason about "the probability that a walker eventually reaches a position on the number line"; more generally one can reason about the expected value of a random variable of interest when certain strategies for deciding whether to continue executing the program are applied. That *nondeterminism* — the mathematical notion underlying abstraction and refinement — is retained in the framework makes it possible to work at various abstraction levels, including at the level of program code. Moreover, nondeterminism's ability to abstract over probability distributions enables switching from quantitative to *qualitative* analyses, when one is interested in properties that hold with probability 1 [13]. Another useful application of qTL is provided by the relationship of its operators to McIver's operators for reasoning about performance-related aspects of probabilistic programs [16].

Our newly-mechanized qTL is suitable for both high-level analyses of properties of probabilistic programs, which may make use of the many algebraic laws we verify for qTL, as well as for concrete verifications, which are supported by the HOL interactive correctness tools built for $pGCL$ programs [9].

In this paper we describe how qTL is implemented in HOL. Although we briefly summarize qTL itself, we refer the reader to [14] for a thorough discussion of it. In Sec. 2 we describe the probabilistic semantics and HOL theories [9] on which the implementation of qTL builds. In Sec. 3 the syntax and semantics of qTL is given. We then continue in Sec. 4 with showing the algebra of qTL which supplies many useful properties for verification. A non-trivial result for (non-homogeneous) random walkers satisfying certain properties is verified in Sec. 5. Finally, we conclude with some remarks on the present state of computer-aided verification for probabilistic programs in Sec. 6.

Notation: We use "." for function application. $x \mathrel{\hat{=}} t$ denotes that x is defined as t. We write α for a (fixed) underlying state space, and $\overline{\alpha}$ for the set of *discrete probability distributions* over α, where a probability distribution is a function from α to the interval $[0,1]$ which is normalized to 1. Functions from α to the non-negative reals are called *expectations*; they are ordered by lifting pointwise the order $\leq$ on the reals — we denote the order on expectations by $\Rrightarrow$. Given two expectations A and B, they are equivalent, denoted $A \equiv B$, exactly when $A \Rrightarrow B$ and $B \Rrightarrow A$. Operations on expectations are pointwise liftings of those on the reals. We write $\underline{c}$ for the constant function returning c for all states. cA denotes $\underline{c} \times A$, where A is an expectation. If f is a probability distribution and A is a measurable function then $\int_f A$ denotes the expected value of A with respect to f. When f is in $\overline{\alpha}$ and A is an expectation, this reduces to $\sum_{s\in\alpha}(f.s) \times (A.s)$ (if defined). If pred is a predicate, then we write $[\mathsf{pred}]$ for the *characteristic function* which takes states satisfying pred to 1, and to 0 otherwise.

2 Probabilistic Semantics

In this section we briefly describe the *quantitative program logic* [21, 14] — the probabilistic semantics that has inspired the choice of semantics for qTL.

Unlike standard programs, probabilistic programs do not produce definite final states — although any *single* execution of such a program will result in the production of some specific state, which one in particular might well be impossible to predict (if its computation is governed by some random event). However over many executions the relative *frequencies* with which final states occur will be correlated with the program's known underlying random behavior. For example executing the *probabilistic choice*

$$b := \mathsf{T} \ {}_{2/3}\!\oplus\ b := \mathsf{F} \ , \tag{1}$$

a large number of times results in roughly $2/3$ of the executions setting b to T.

The language *pGCL* [19] — the *Guarded Command Language* [3] augmented with the probabilistic choice construct mentioned above — and its associated *quantitative logic* [14] were developed to express such programs and to derive their probabilistic properties by extending the classical assertional style of programming [20]. Programs in *pGCL* are modeled (operationally) as functions (or transitions) which map *initial states* in α to (sets of) probability distributions over *final states* — the program at (1) for instance operates over a state space of size 2, and has a single transition which maps any initial state to a (single) final distribution; we represent that distribution as a normalized function d, evaluating to $2/3$ when $b = \mathsf{T}$ and to $1/3$ when $b = \mathsf{F}$.

Since properties now involve numeric frequencies they are expressed via a logic of (non-negative) *real-valued functions*, or *expectations.* For example the property "the final value of b is T with probability $2/3$" can be expressed as $\int_d [b = \mathsf{T}] \equiv \underline{2/3}$ — the *expected value* of the function $[b = \mathsf{T}]$ with respect to d evaluates to $2/3 \times 1 + 1/3 \times 0 = 2/3$. However, direct appeal to the operational semantics is often unwieldy — better is the equivalent transformer-style semantics which is obtained by rationalizing the above calculation in terms of expectations rather than transitions. The expectation $[b = \mathsf{T}]$ has been transformed to the expectation $\underline{2/3}$ by the program (1) above so that they are in the relation "$\underline{2/3}$ is the expected value of $[b = \mathsf{T}]$ with respect to the program's result distribution".

More generally having access to real-valued functions makes it possible to express many properties as "random variables" of interest, which for us are synonymous with expectations. Then given a program P, an expectation A and a state $s \in \alpha$, we define $\mathsf{wp}.P.A.s$ to be the expected value of A with respect to the result distribution of program P if executed initially from state s [14]. We say that $\mathsf{wp}.P$ is the *expectation transformer* relative to P. In our example that allows us to write

$$\underline{2/3} \quad \equiv \quad \mathsf{wp}.(b := \mathsf{T} \;{}_{2/3}\!\oplus\; b := \mathsf{F}).[b = \mathsf{T}] \; . \tag{2}$$

In the case that *nondeterminism*[1] is present, execution of P results in a set of possible distributions and the definition of wp is modified to take account of this — in fact $\mathsf{wp}.P.A.s$ may be defined so that it delivers either the *least*- or *greatest*-expected value with respect to all distributions in the result set. Those choices correspond respectively to a *demonic* or *angelic* resolution of the nondeterminism — which interpretation is used depends very much on the application. Operationally such interpretations of nondeterminism can be justified by viewing the underlying computation as a *game* — the agent responsible for resolving the nondeterminism is trying to minimize/maximize the random variable of interest.

With the transformer approach it is possible to express temporal properties of systems. For example, fixing the underlying computation to be the expectation transformer relative to the program at (1), the modal primitive *next time* "$\circ$" has a natural interpretation relative to an expectation, say $[b = \mathsf{T}]$ — the

[1] Nondeterminism represents genuinely unquantifiable uncertainty in systems.

intended interpretation of $\circ[b = \mathsf{T}]$ is "$\overline{2/3}$", and expresses as Eqn. 2 does, that the probability of $(b = \mathsf{T})$'s holding after *one execution step* is 2/3. More generally, given an expectation transformer *step*, and an expectation A, $\circ A$ is the expected value of A when transformed as explained above by *step*.

Reachability properties can also be expressed, using while-loops. For example since the following loop

$$\mathsf{do}\ (b = \mathsf{F}) \quad \rightarrow \quad b := \mathsf{T}\ {}_{2/3}\!\oplus\ b := \mathsf{F}\ \mathsf{od}\ ,$$

iterates as long as $(b = \mathsf{F})$ holds, its termination probability is in fact the probability of *eventually* establishing $(b = \mathsf{T})$ by repeatedly executing the probabilistic choice. By a simple fact of probability theory this probability is 1. Although many temporal properties can be expressed and reasoned about in the *pGCL* framework, analysis of more complex temporal behavior requires the introduction of an extra logical layer.

The *quantitative Temporal Logic* (*qTL*) [18, 14] — the probabilistic extension of temporal logic — was developed to express such properties and provide a set of logical tools to reason about them. The underlying computation is viewed as an expectation transformer, as described above, and the temporal operators *eventually* ($\Diamond$), *always* ($\Box$), and *unless* ($\rhd$), are defined in terms of fixed points over expectation transformers. We will set out the formal semantics in Sec. 3; in the remainder of this section we describe the HOL theories of non-negative reals, expectations and their transformers, which are the basis of the *qTL* theory.

2.1 Formalized Expectation Transformers

Non-negative reals have been formalized in HOL by Hurd et al. [9] as a type of higher-order logic, called posreal. A nice feature of the theory is that the posreal type also includes a constant ∞, which dominates other elements of the type with respect to the natural order $\leq$ on posreal. The usual arithmetic operations are defined over this type.

Expectations, formalized in HOL by Hurd et al., are functions of type:

$$(\alpha)\mathsf{expect} \mathrel{\hat{=}} \alpha \rightarrow \mathsf{posreal}\ .$$

where α, the type of the state space, is polymorphic and can be instantiated to any type of higher-order logic. The order and operations on posreal are lifted pointwise to operations on expectations. Since all expectations are bounded by $\underline{\infty}$ the expectations form a complete partial order. In fact, the space of expectations bounded by any constant expectation $\underline{c}$ forms a complete partial order, and fixed points are well-defined for any monotonic function on such expectations.

We define some extra operations on expectations which are needed in the probabilistic context; in Fig. 1 we name a few. These operations can be viewed as generalizations of standard operations on predicates; however since we are working in the more general context of the reals there might be several suitable generalizations for each operation on predicates. For example, both "$\sqcap$" and "&" are suitable, in different contexts, as generalizations of conjunction. The first sanity check when picking the operators is whether the truth tables are as expected for standard predicates. The choice of the operators is fully-motivated in [14].

$$\begin{array}{lcll}
A \sqcap B & \hat{=} & (\lambda s \bullet A.s \ \mathsf{min}\ B.s) & \textit{minimum} \\
A \sqcup B & \hat{=} & (\lambda s \bullet A.s \ \mathsf{max}\ B.s) & \textit{maximum} \\
\neg A & \hat{=} & (\lambda s \bullet 1 - A.s) & \textit{complement} \\
A \ \&\ B & \hat{=} & (\lambda s \bullet A.s + B.s - 1) & \textit{conjunction} \\
A \mathrel{-\!\triangleright} B & \hat{=} & (\lambda s \bullet 1 - (A.s - B.s)) & \textit{implication, } \Rrightarrow\textit{-adjoint of } \& \\
P \rightrightarrows Q & \hat{=} & [P \Rightarrow Q] & \textit{``standard'' implication}
\end{array}$$

A, B range over (α)expect; P, Q over standard (boolean-valued) predicates.
$\neg$ binds tightest, whereas the order relations $\Rrightarrow, \equiv$ weakest.
We will also use more general versions of some of the above operations, with β — a scalar from posreal — substituted for 1. They can be viewed as scaling by β the corresponding operation. Such operators will be denoted by say $\&_\beta$ instead of $\&$.

Fig. 1. Some operations on expectations.

Expectation transformers are functions from expectations to expectations:

$$(\alpha)\mathsf{transformer} \mathrel{\hat{=}} (\alpha)\mathsf{expect} \to (\alpha)\mathsf{expect}\ .$$

For us the interesting expectation transformers are those that determine the wp-meaning of *pGCL* programs, that is, describe how *pGCL* commands transform *post-expectations* into *pre-expectations*. For example, assignments induce substitutions, and probabilistic choices average according to the specified probabilities. We show in Fig. 2 the definitions for the straight-line commands; the wp-semantics for the complete *pGCL* (including Boolean choice, and while-loops) has been formalized in HOL by Hurd et al. [9].

We do not discuss the full semantics of *pGCL* here because the structure of the wp-operator is not of importance in the development of the *qTL* theory.[2,3] More important are the properties characterizing the wp-transformers, which generalize Dijkstra's *healthiness conditions* characterizing standard programs. The healthiness conditions for demonic probabilistic transformers are shown in Fig. 3; they are part of the HOL expectations theory formalized in HOL by Hurd et al. [9]. Monotonicity, for example, ensures that the fixed-points $(\mu X \bullet t.X)$ (least), and $(\nu X \bullet t.X)$ (greatest) are well defined for a transformer t.

[2] It is however imperative when verifying concrete algorithms. For such verifications correctness tools [9] can be used to reduce reasoning about goals of the form "$A \Rrightarrow \mathsf{wp}.\mathit{Prog}.B$" to reasoning about relationships between ordinary expectations.

[3] We will not make any assumptions about how the state space is implemented, which increases the utility of the mechanized theory.

skip	$\mathsf{wp}.\mathsf{skip}.A \mathrel{\hat{=}} A$,
assignment	$\mathsf{wp}.(x := E).A \mathrel{\hat{=}} A[x := E]$,
sequential composition	$\mathsf{wp}.(r; r').A \mathrel{\hat{=}} \mathsf{wp}.r.(\mathsf{wp}.r'.A)$,
probabilistic choice	$\mathsf{wp}.(r\ {}_p\!\oplus r').A \mathrel{\hat{=}} p \times \mathsf{wp}.r.A + (\underline{1} - p) \times \mathsf{wp}.r'.A$,
multi-way prob. choice	$\mathsf{wp}.(r_0 @ p_0 \mid \ldots \mid r_n @ p_n).A \mathrel{\hat{=}} p_0 \times \mathsf{wp}.r_0.A + \cdots + p_n \times \mathsf{wp}.r_n.A$,
nondeterm. choice	$\mathsf{wp}.(r \mathbin{[\!]} r').A \mathrel{\hat{=}} \mathsf{wp}.r.A \ \sqcap\ \mathsf{wp}.r'.A$.

The state space here is instantiated to $\mathsf{string} \to \mathbb{Z}$.
E is an integer-valued state function, p is an expectation bounded by $\underline{1}$, $p_0 \ldots p_n$ sum to $\underline{1}$.
Nondeterminism is interpreted demonically, that is minimal-seeking.

Fig. 2. Structural definitions of wp for straight-line *pGCL*.

$$
\begin{array}{lcl}
\mathsf{feasible}\ t & \hat{=} & \forall A, c \bullet A \Rrightarrow \underline{c} \quad \Rightarrow \quad t.A \Rrightarrow \underline{c} \\
\mathsf{monotonic}\ t & \hat{=} & \forall A, B \bullet A \Rrightarrow B \quad \Rightarrow \quad t.A \Rrightarrow t.B \\
\mathsf{sublinear}\ t & \hat{=} & \forall A, B, c, c_1, c_2 \bullet t.(c_1 A + c_2 B - \underline{c}) \Rrightarrow c_1(t.A) + c_2(t.B) - \underline{c}
\end{array}
$$

t ranges over (α)transformer, A, B over (α)expect, and c, c_1, c_2 over posreal.
Feasibility implies $t.A \Rrightarrow \underline{\sqcup A}$, so $t.\underline{0} \equiv \underline{0}$.
Sublinearity generalizes *conjunctivity* of standard predicate transformers.
Monotonicity is a consequence of sublinearity.
The wp-operator (part of which is given in Fig. 2) satisfies all three conditions [14, 9].

Fig. 3. Healthiness conditions for demonic transformers.

3 *qTL* and its Semantics

In this section we show the syntax and semantics of *qTL*, Morgan and McIver's [18, 14] probabilistic extension of temporal logic.

The syntax of *qTL* formulas, set out in Fig. 4, is defined in HOL as a new datatype called formula. Note that any expectation can be transformed into a *qTL* formula of the form Atom(A).

As hinted in the previous section, when interpreting the temporal formulas we do so with respect to a fixed expectation transformer *step*, which describes the underlying computation; the intention is that most of the time and whenever convenient *step* is wp.*Step*, where *Step* is a syntactic *pGCL* program.

The formal semantics of *qTL* is defined on the structure of the formulas and is set out in full in Fig. 5 — it essentially generalizes standard modal μ-calculus [11] from Booleans to reals, and takes the temporal subset of that [14]. We postpone explaining β's appearance in the definitions until further down in this section.

The operational interpretation of the quantitative temporal operators requires thinking in terms of games. Take $\Diamond(\mathsf{Atom}(A))$: if A is a standard expectation — there exists a predicate P such that $A \equiv [P]$ — then the interpretation of

$$
\begin{array}{rcl}
\mathsf{formula} & \hat{=} & \mathsf{Atom}(A) \\
 & | & \neg a \mid a \sqcup b \mid a \sqcap b \mid a \,\&\, b \mid a \mathrel{-\!\!\triangleright} b \mid a \rightrightarrows b \\
 & | & \circ a \mid \Diamond a \mid a \rhd b
\end{array}
$$

A is an expectation. Lower-case letters $a, b, \ldots$ range over formulas.
The second row contains state formulas. To improve readability here we use the same symbols as for the operators on expectations.
The third row describes proper temporal formulas, which are read as usual: *next time*, *eventually*, (weak) *unless*. *always* is defined in terms of *unless*:

$$\Box a \mathrel{\hat{=}} a \rhd (\mathsf{Atom}(\mathsf{false}))$$

where $\mathsf{false} \mathrel{\hat{=}} \underline{0}$.
Temporal operators associate to the right, and apart from $\neg$ bind tighter than the rest of the operators.

Fig. 4. The syntax of *qTL*.

$\Diamond(\mathsf{Atom}(A))$ is in fact the probability that P is eventually established. However, in the more general case when A is a proper expectation, thinking in terms of "establishing A" does not make sense. The interpretation of $\Diamond(\mathsf{Atom}(A))$ in the general case relies on a game-like analogy, and we quote directly from [13]:

> [I]t is the supremum, over all strategies that determine in each state whether to make another transition or to stop, of the expected value of A when the strategy says "stop"; the strategy "never stop" gives 0 by definition.

The other operators are interpreted similarly. Note for example that nontermination in the case of $\Box$ is interpreted as a success, and that the interpretation of $\rhd$ involves a mixture of minimizing and maximizing strategies.

With the game analogy in mind it is clear that there is no need to focus exclusively on expectations bounded by 1 when specifying the semantics of *qTL*, although admittedly most of the time this is sufficient (since most often than not we are interested in temporal behavior relative to standard expectations). Especially as far as the implementation in HOL is concerned, it is advantageous to parameterize the semantic operator with the bound "β". This is so because proving properties for an arbitrary bound β (although sometimes under the assumption that $\beta \neq \infty$) is no more difficult than in the 1-bounded case. Moreover, specializing to the 1-bounded expectations is trivial, and clearly we also have access to reasoning about unbounded expectations, although as expected the properties available in this case are fewer.

We conclude this section with a few words on the expressibility of *qTL*. Recall that in the wp-definition (Fig. 2) we interpreted nondeterminism demonically. With such an interpretation we can only describe *universal* versions of the temporal operators, since demonic interpretations give us guarantees on least

$$
\begin{array}{lcl}
\|\mathsf{Atom}(A)\|_{(\beta,step)} & \hat{=} & A \sqcap \underline{\beta} \\
\|\neg a\|_{(\beta,step)} & \hat{=} & \underline{\beta} - \|a\|_{(\beta,step)} \\
\|a \sqcap b\|_{(\beta,step)} & \hat{=} & \|a\|_{(\beta,step)} \sqcap \|b\|_{(\beta,step)} \\
\|a \sqcup b\|_{(\beta,step)} & \hat{=} & \|a\|_{(\beta,step)} \sqcup \|b\|_{(\beta,step)} \\
\|a \,\&\, b\|_{(\beta,step)} & \hat{=} & \|a\|_{(\beta,step)} \,\&_\beta\, \|b\|_{(\beta,step)} \\
\|a \multimapdotbothB b\|_{(\beta,step)} & \hat{=} & \|a\|_{(\beta,step)} \multimapdotbothB_\beta \|b\|_{(\beta,step)} \\
\|b \rightrightarrows b\|_{(\beta,step)} & \hat{=} & \|a\|_{(\beta,step)} \rightrightarrows_\beta \|b\|_{(\beta,step)} \\
\|\circ a\|_{(\beta,step)} & \hat{=} & step.\|a\|_{(\beta,step)} \\
\|\Diamond a\|_{(\beta,step)} & \hat{=} & (\mu A \bullet \|a\|_{(\beta,step)} \sqcup step.A) \\
\|a \rhd b\|_{(\beta,step)} & \hat{=} & (\nu A \bullet \|b\|_{(\beta,step)} \sqcup (\|a\|_{(\beta,step)} \sqcap step.A))
\end{array}
$$

step is a (α)transformer. β is a posreal scalar.
The (least μ and greatest ν) fixed points are defined over the complete partial order of expectations bounded by β. For a monotonic *step* the fixed points are well-defined.
For finite β, $\sqcup$ and $\sqcap$ are duals: $\|a \sqcup b\|_{(\beta,step)} \equiv \|\neg(\neg a \sqcap \neg b)\|_{(\beta,step)}$.

Fig. 5. The semantics of *qTL*.

expected values. For a logic that also admits *existential* versions of the operators the transformer *step* should also allow angelic interpretations. However, the price to pay is that sublinearity does not generally hold for transformers containing angelic nondeterminism[4]. Note however that for the semantics to be well-defined monotonicity is all that is needed.

4 The Algebra of *qTL*.

In this section we show some properties that hold under the main general healthiness assumptions about the transformer *step*. The properties are generalizations of Ben-Ari et al.'s [1]. They are essential logical tools when verifying algorithms since often direct appeal to the semantics is impractical. The results have been verified in HOL so we do not show the proofs here.

A useful property in concrete verifications is *feasibility*, which can be proved easily if *step* is feasible:

$$\forall \beta, step, a \bullet \; \|a\|_{(\beta,step)} \Rrightarrow \underline{\beta} \; .$$

From now on to improve readability we identify formulas with their semantics; we identify the semantic bound β when that can be ambiguous, in particular, we mark those properties that assume that the bound is finite. We also pretty-print $\mathsf{Atom}(A)$ as simply A. As far as the computation *step* is concerned

[4] Purely angelic transformers are *suplinear*

$$\mathsf{suplinear}\; t \;\hat{=}\; \forall A, B, c, c_1, c_2 \bullet c_1(t.A) + c_2(t.B) - \underline{c} \;\Rrightarrow\; t.(c_1 A + c_2 B - \underline{c}) \; .$$

we assume that the basic properties it satisfies are feasibility and monotonicity. However many of the interesting properties require *step* be sublinear (i.e. demonic). We will then show which properties rely on this assumption. Although a (purely) demonic *step* is in most situations expressive enough, there are some situations when it is more advantageous to interpret nondeterminism angelically. McIver [16], for example, showed how temporal operators working with an angelic transformer can be used to reason about the efficiency of probabilistic programs.

The properties of "$\circ$" are trivially inherited from those of the transformer *step*. In Fig. 6 we show some basic properties; they are proved directly from monotonicity of *step*.

$$\begin{array}{llll}
a \Rrightarrow b & \Rightarrow & \circ a \Rrightarrow \circ b & \textit{monotonic, next time} \\
a \Rrightarrow b & \Rightarrow & \Diamond a \Rrightarrow \Diamond b & \textit{monotonic, eventually} \\
a \Rrightarrow b & \Rightarrow & \Box a \Rrightarrow \Box b & \textit{monotonic, always} \\
a \Rrightarrow b & \Rightarrow & \neg b \Rrightarrow \neg a & \textit{antimonotonic, complement } (*)
\end{array}$$

$(*)$ assumes $\beta \neq \infty$.
Because some of the formulas, e.g. those containing $\rightarrowtriangle$, are neither "logically positive" nor "negative", a more general result for the (anti)monotonicity of positive (negative) formulas cannot be established.
For *op* any of the monotonic operators, it is easy to prove:

$$\begin{array}{lll}
op(a) \sqcup op(b) \Rrightarrow op(a \sqcup b) & & \textit{subdistributes } \sqcup \\
op(a \sqcap b) \Rrightarrow op(a) \sqcap op(b) & & \textit{supdistributes } \sqcap
\end{array}$$

Fig. 6. (Anti)Monotonicity properties.

Next we prove the basic fixed points properties for $\Diamond, \rhd, \Box$, which are shown in Fig. 7. The *greatest* property for *always*, for example, is a very useful tool for bounding from below the probability that *invariance* properties hold. Moreover the verification conditions in such cases can be discharged with the help of total correctness calculators [9].

We also note that the usual "double" laws hold in our setting, and the proofs do not require anything but the monotonicity assumption:

$$\begin{array}{ll}
\Diamond\Diamond a \equiv \Diamond a & \textit{double, eventually}\ , \\
\Box\Box a \equiv \Box a & \textit{double, always}\ .
\end{array}$$

The most challenging and interesting properties are shown in Fig. 8. As noted in [14] it can require some ingenuity to describe the intuition behind some of them. However, what is clear is that these properties allow breaking down the reasoning to more manageable pieces, and achieving modularity of proofs. The example in the next section illustrates the usefulness of such properties.

$$
\begin{array}{rll}
 & \Diamond a \equiv a \ \sqcup \ \circ\Diamond a & \textit{fixed point, eventually} \\
a \ \sqcup \ \circ b \Rrightarrow b \Rightarrow & \Diamond a \Rrightarrow b & \textit{least, eventually} \\
 & \circ a \ \sqcup \ \circ\Diamond a \Rrightarrow \Diamond a & \\
 & a \rhd b \equiv b \ \sqcup \ (a \ \sqcap \ \circ(a \rhd b)) & \textit{fixed point, unless} \\
d \Rrightarrow b \ \sqcup \ (a \ \sqcap \ \circ d) \Rightarrow & d \Rrightarrow a \rhd b & \textit{greatest, unless} \\
 & \Box a \equiv a \ \sqcap \ \circ\Box a & \textit{fixed point, always} \\
b \Rrightarrow a \ \sqcap \ \circ b \Rightarrow & b \Rrightarrow \Box a & \textit{greatest, always} \\
 & \Box a \Rrightarrow \circ a \ \sqcap \ \circ\Box a &
\end{array}
$$

The monotonicity of *step* suffices for these properties.

Fig. 7. Fixed point properties of $\Diamond, \rhd, \Box$.

$$
\begin{array}{rll}
\Box a \ \& \ \Box b \Rrightarrow & \Box(a \ \& \ b) & \textit{always subdistributes } \& \\
\Box(a \mathrel{-\!\rhd} b) \Rrightarrow & \Box a \mathrel{-\!\rhd} \Box b & \textit{always supdistributes} \mathrel{-\!\rhd} \\
a \ \& \ \Box(a \rightrightarrows \circ a) \Rrightarrow & \Box a & \\
\Box(a \mathrel{-\!\rhd} b) \ \& \ \Diamond a \Rrightarrow & \Diamond b & \\
\Box a \Rrightarrow & \neg\Diamond(\neg a) & \textit{always-eventually duality} \\
\Diamond a \ \& \ \Box(\circ a \rightrightarrows a) \Rrightarrow & a & \\
\Diamond a \ \& \ \Box b \Rrightarrow & \Diamond(a \ \& \ b) &
\end{array}
$$

The properties assume $\beta \neq \infty$, and that *step* is sublinear.

Fig. 8. Duality properties and more.

5 Example: The Jumping Bean

The verification of the example presented in this section follows very closely Morgan's presentation of the proof [17]; the same result is also proved in [14], although the presentation there is given in terms of while-loops. Throughout this section we assume that the semantic bound β is 1. Moreover, we assume that the computation step is given in terms of wp.

The Jumping Bean sits on the number line and hops some integer distance, either up or down. Informally, the Bean must move according to the following rules:

1. With some nonzero probability, it *must move* at least one unit up or down.
2. There is a *uniform maximum distance*, arbitrarily large but fixed, that it can travel in one jump.
3. Its *expected movement is never down*: on average, it either moves up or stays where it is.

For example, the *symmetric walker*:

$$
\textit{SymmetricWalker} \quad \hat{=} \quad n := n - 1 \ {}_{1/2}\oplus \ n := n + 1 \ , \tag{3}
$$

trivially satisfies the required behavior: it moves with probability 1, the maximum distance of each jump is 1, and on average it stays in place. A less trivial example of an acceptable behavior is that of this *demonic walker*:

$$\mathit{DemonicWalker} \mathrel{\hat{=}} \begin{pmatrix} n := \ n+2 \ @ \ 1/4 \\ \mathsf{skip} \qquad\quad @ \ 5/12 \\ n := \ n-1 \ @ \ 1/3 \end{pmatrix} \ [\!] \ \begin{pmatrix} n := \ n+1 \ @ \ 1/2 \\ n := \ n-1 \ @ \ 1/2 \end{pmatrix} \tag{4}$$

On each step the walker can choose to move according to the distribution specified in the left statement or the one in the right — which choice will be made cannot be predicted in advance. However, each of the alternatives satisfies the movement conditions set above, thus the *demonic walker*'s behavior is in the acceptable range.

We will prove that under the three assumptions above (which we state formally below), and given any number H on the integer line, the Bean will eventually jump over it with probability 1:

$$\Diamond[H \le n] \ \equiv \ \underline{1} \ . \tag{5}$$

Here n is the variable recording the current position of the Bean on the line. First we note that

$$\Diamond[H \le n] \ \Rrightarrow \ \underline{1} \ ,$$

from the feasibility of the formulas and $[H \le n]$'s being a standard expectation (as such it is bounded pointwise from above by 1). As expected, it is the other direction that is nontrivial to show — the least fixed point property is not helpful since it works the other way around. Moreover, since we do not know the exact probabilities involved in the jumps we cannot just solve the fixed-point equation. This is an indication that we need to check whether some *zero-one* law is at work. The next theorem shows an example of such a law for *eventually*.

Theorem 1 (Zero-one law for *eventually*) [5]*If the probability of eventually establishing a standard predicate P is everywhere bounded away from* 0 *then it is in fact* 1*:*

$$(\exists c \bullet 0 < c \ \wedge \ \underline{c} \Rrightarrow \Diamond[P]) \quad \Rightarrow \quad \Diamond[P] \ \equiv \ \underline{1}$$

Note that we are assuming that step is nondivergent.[6]

Another *zero-one* law we have available for $\Diamond$ of a standard expectation $[P]$ is a lot like the variant-rule for probabilistic loops [14]. Informally, if we can find an integer-valued function of the state — a *variant* — that

[5] The theorem works for nondeterministic programs — Morgan [17] shows the proof for the deterministic case; the proof for the nondeterministic case made use of a property of the wp-semantics which says that for any *pGCL* program *Prog* and post-expectation A, there exists a deterministic (probabilistic) program *Prog'* such that $\forall B \bullet \mathsf{wp}.\mathit{Prog}.B \Rrightarrow \mathsf{wp}.\mathit{Prog'}.B$ and $\mathsf{wp}.\mathit{Prog}.A \equiv \mathsf{wp}.\mathit{Prog'}.A$ [14].

[6] Nondivergence characterizes computation steps that do not have any effect on constant expectations, that is $(\forall c \bullet \circ\underline{c} \equiv \underline{c})$ — for *pGCL* programs this means that they *terminate with probability* 1 [14].

- is bounded from below and above while P is not established,
- and the probability of its increasing on every computation step is bounded away from 0,

then we know that with probability 1, P will eventually be established.

Theorem 2 (Variant-rule for *eventually*) *For any standard predicate P, an integer-valued* state variant V*,*

$$\begin{array}{l}(\exists c, L, H \bullet \\ \quad 0 < c \;\wedge\; \neg(P \Rightarrow L \leq V < H) \;\wedge\; (\forall N \bullet c[\neg P \wedge (V = N)] \;\Rrightarrow\; \circ[N < V])) \Rightarrow \\ \Diamond[P] \equiv \underline{1}\end{array}$$

From the informal assumptions the position of the Bean must increase with some probability, if the Bean must move on each step and its expected movement is never down. However, we cannot apply Thm. 2 directly to our initial goal because we cannot bound our variant n from below.

Instead we argue that given any (finite) position L to the left of H, if the Bean starts from a position within the interval $[L \ldots H)$ then

- it eventually escapes from that interval with probability 1,
- the probability of its escaping from the right can be bounded away from 0 — this probability depends on how low L is chosen.

The glue to all the pieces is the following lemma:

Lemma 3 [7] *For any H, L*

$$\Diamond[\neg(L \leq n < H)] \quad \& \quad [L \leq n] \triangleright [H \leq n] \quad \Rrightarrow \quad \Diamond[H \leq n]\ .$$

The part that needs some explanation is when for the chosen L the initial position of n falls to the left of the interval $[L \ldots H)$. Then trivially the first conjunct of the left-hand side is 1, however since both $[L \leq n]$ and $[H \leq n]$ are 0, by the semantics of $\triangleright$ and the definition of & we get on the left-hand side 0, which in other words means that we cannot say anything nontrivial about the probability of $(H \leq n)$'s establishment. Clearly L should be chosen such that it is lower than n's initial position (when this itself is lower than H). We in fact show that it can be chosen low enough so that the probability of escaping to the right of H is at least 1/2, which allows us to prove the following:

$$\begin{array}{lll} & \Diamond[\neg(L \leq n < H)] \quad \& \quad [L \leq n] \triangleright [H \leq n] & \\ \Lleftarrow & \underline{1} \;\;\&\;\; \underline{1/2} & \text{To show later (8,9)} \\ \equiv & \underline{1/2}\ . & \text{Definition of \&, Fig 1} \end{array}$$

[7] The lemma is an instance of a more general one:

$$\forall A, P \bullet (\Diamond(A \mathrel{-\!\!\triangleright} [P]) \;\&\; A \triangleright [P] \Rrightarrow \Diamond[P])\ .$$

If $A \equiv [Q]$ then directly from the definition of $\mathrel{-\!\!\triangleright}$, $[Q] \mathrel{-\!\!\triangleright} [P] \equiv [\neg Q \vee P]$.

Having bounded $\Diamond[H \le n]$ away from 0, by appeal to Thm. 1, we can conclude that the Bean's eventual escape to the right of H is *almost certain*, i.e. occurs with probability 1.

Now we fill in the missing steps, for which we first state the assumptions about the Jumping Bean formally in Fig. 9.

$$\begin{array}{ll} \forall L, H \bullet L \le H \Rightarrow & \\ \quad \exists c \bullet 0 < c \Rightarrow & \qquad\qquad \textit{JB1} \\ \qquad \forall N \bullet c[L \le n < H \wedge (n = N)] \;\Rrightarrow\; \circ(N < n) & \end{array}$$

JB1 encodes the requirement that the Bean must move on each step; more precisely that it must move up with some probability.[8]

$$\begin{array}{ll} \exists K \bullet 0 < K \wedge & \\ \quad \forall L, H \bullet L \le H \Rightarrow & \qquad\qquad \textit{JB2} \\ \qquad \langle L : n : H \rangle \;\Rrightarrow\; \circ\langle L : n : H + K \rangle & \end{array}$$

where

$$\langle L : n : H \rangle \;\;\hat{=}\;\; [L \le n \le H] \times (n - \underline{L}) + [H < n] \times (\underline{H - L}) \; .$$

K in *JB2* is the maximum distance the Bean can travel in one step; recall that in our informal assumption we required that such a distance be bounded. *JB2* also encodes the requirement that the movement never be down: take $L + 1 = H = N$, where N is the Bean's initial position (in state s). Then $\langle L : n : L + 1 \rangle$ in s is 1, and the only way for $1 \le \circ\langle L : n : L + 1 + K \rangle.s$ is if finally $N \le n$.
We also assume that the jump is nondivergent.

Fig. 9. Jumping Bean assumptions.

Appealing to Thm. 2 with state-variant n we directly conclude from *JB1* that:

$$\forall L, H \bullet L \le H \quad \Rightarrow \quad \Diamond[\neg(L \le n < H)] \equiv \underline{1} \; . \tag{8}$$

[8] Note that a stronger (and simpler-looking) alternative for *JB1* would be:

$$\exists c \bullet 0 < c \wedge (\forall N \bullet c[n = N] \Rrightarrow \circ[N < n]) \; . \tag{6}$$

Here is an example of a jump that satisfies *JB1* but not the assumption at (6):

$$\textit{LazyBean} \quad \hat{=} \quad \begin{array}{ll} n := \; n + 1 & @ \; \frac{1}{2(|n| \;\mathsf{max}\; 1)} \\ \mathsf{skip} & @ \; 1 - \frac{1}{(|n| \;\mathsf{max}\; 1)} \\ n := \; n - 1 & @ \; \frac{1}{2(|n| \;\mathsf{max}\; 1)} \end{array} \tag{7}$$

Clearly for any interval $[L \ldots H)$, it is possible to bound away from 0 the probability of jumping up but a more general bound cannot be established, as the farther the Bean goes from 0 the lazier it gets.

Next we prove that under assumption *JB2* the bound L can be chosen such that:

$$\underline{1/2} \quad \Rrightarrow \quad [L \leq n] \rhd [H \leq n] \ . \tag{9}$$

First we note that given an initial position N of the Bean, if $H \leq N$ then the right-hand side trivially evaluates to 1, and thus it is enough to choose any $L \leq H$ (for which also (8) is true). So we can safely assume that $N < H$ initially.

Using the least-fixed point property for *unless*, and some transformations of *JB2* which we do not show here because the details are rather unenlightening, we can prove:

$$\frac{\langle L : n : H + K\rangle}{H + K - L} \Rrightarrow [L \leq n] \rhd [H \leq n] \ .$$

For any initial position of the Bean N (to the left of H), we can see that choosing $L = 2N - (H + K)$, makes the left-hand side 1/2.

We have thus proved the claim we set about to:

Theorem 4 (Jumping Bean) *Given any number H on the number line, a Jumping Bean satisfying the assumptions JB1 and JB2 (see Fig. 9) will eventually jump over it with probability 1:*

$$\Diamond[H \leq n] \equiv \underline{1} \ .$$

This result is more general than similar ones found in probability textbooks: the actual transition probabilities may vary from place to place (see *LazyBean* (7)) so our walkers may be non-homogeneous. Allowing nondeterministic moves is a further generalization to traditional such theorems — the transition probabilities may vary not only on different places of the number line but also on different visits to the same place (see *DemonicWalker* (4)). We also note that the state space is infinite, and the theorem is parameterized on the limit on the jump.

To give an idea of the mechanization effort involved in the verification, the proofs of this section (including those for the *zero-one* laws) required around 2000 lines of proof script. However, proving that the *SymmetricWalker, DemonicWalker* and *LazyBean* satisfy the assumptions is by comparison much easier, and the proofs were around 100, 150, 300 lines respectively. The latter verifications can also be supported by correctness tools built for *pGCL* [9], cutting the verification effort further.

Verifications in a theorem prover are most useful if properties about classes of probabilistic programs can be proved — this is possible either as in our example, by stating the assumptions as generally as possible, or by specifying the syntactic *pGCL* program as abstractly as possible and using refinement to derive the property for concrete programs. The refinement approach is justified by theorems such as the following:

$$\forall \beta, step, step' \bullet \ step \sqsubseteq step' \quad \Rightarrow \quad \|a\|_{(\beta,step)} \Rrightarrow \|a\|_{(\beta,step')} \ ,$$

where $step \sqsubseteq step'$ denotes that $step'$ *refines* $step$, i.e. $\forall A \bullet \mathsf{wp}.Step.A \Rrightarrow \mathsf{wp}.Step'.A$, and a is a positive formula.

6 Conclusions and Related Work

The importance of having methods suitable for reasoning about randomized algorithms is already widely recognized. A recent survey on a number of approaches used to verify the correctness of such algorithms is presented by Norman [22]. As far as computer-aided verification is concerned much work has been done in the area of model checking, with PRISM [12, 23], ETMCC [4], Ymer [24], RAPTURE [10], and ProbVerus [6] being some of the model-checking tools for probabilistic systems. They differ in whether they model-check purely probabilistic systems (ETMCC, Ymer, ProbVerus), or probabilistic nondeterministic systems (PRISM, RAPTURE); the temporal logics used to express the properties to be model-checked also vary, although they mainly fall into two categories: logics that keep the formulas standard and associate probabilities with computation paths, and those that have formulas denote probabilities rather than truth values. *qTL* falls into the latter category.

Although impressive work has been done to tackle the problems inherent in model-checking approaches, such as state explosion, model-state finiteness, and the inability to parameterize systems, as concluded in [22] most of the probabilistic model-checking tools are not yet sufficiently developed compared to their non-probabilistic peers; as such they still are applicable to only complete, finite-state models.

Theorem provers can deal with infinite-state systems, and parameterization — our Jumping Bean theorem is one such example. However, verification using theorem provers is still labor-intensive. Compared to model-checking efforts not much work has been done on providing theorem prover support for verifying probabilistic systems. To our knowledge the first work on verifying probabilistic programs using a theorem prover is that of Hurd [8], who formalized probability theory in HOL and implemented tools for reasoning about correctness of (purely) probabilistic algorithms. More recently work on formalizing *pGCL* and its quantitative logic in HOL has appeared [9, 2], and it is this work that we extended. Similarly, support for probabilistic reasoning in the *pGCL* style is being incorporated in the B method [7, 15].

Acknowledgments

I thank Annabelle McIver for answering my questions with regard to *qTL* and commenting on a draft of this paper. I also thank Carroll Morgan for making available the nice Jumping Bean slides.

References

1. M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
2. O. Celiku and A. McIver. Cost-based analysis of probabilistic programs mechanised in HOL. *Nordic Journal of Computing*, 11(2):102–128, 2004.

3. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
4. Erlangen-Twente Markov Chain Checker. *http://www.informatik.uni-erlangen.de/etmcc/*.
5. M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.
6. V. Hartonas-Garmhausen, S. V. A. Campos, and E. M. Clarke. Probverus: Probabilistic symbolic model checking. In *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, Proceedings*, volume 1601 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 1999.
7. T. S. Hoang, Z. Jin, K. Robinson, A. McIver, and C. Morgan. Probabilistic invariants for probabilistic machines. In *ZB 2003: Formal Specification and Development in Z and B,Proceedings*, volume 2651 of *Lecture Notes in Computer Science*, pages 240–259. Springer, 2003.
8. J. Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.
9. J. Hurd, A. McIver, and C. Morgan. Probabilistic guarded commands mechanized in HOL. In *Proc. of QAPL 2004*, Mar. 2004.
10. B. Jeannet, P. D'Argenio, and K. Larsen. RAPTURE: A tool for verifying Markov Decision Processes. In I. Cerna, editor, *Tools Day'02,* Brno, Czech Republic, Technical Report. Faculty of Informatics, Masaryk University Brno, 2002.
11. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
12. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Proceedings of TOOLS 2002*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204. Springer, Apr. 2002.
13. A. McIver and C. Morgan. Almost-certain eventualities and abstract probabilities in the quantitative temporal logic qTL. *Theoretical Computer Science*, 293(3):507–534, 2003.
14. A. McIver and C. Morgan. *Abstraction, refinement and proof for probabilistic systems*. Springer, 2004.
15. A. McIver, C. Morgan, and T. S. Hoang. Probabilistic termination in B. In *ZB 2003: Formal Specification and Development in Z and B, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*, pages 216–239. Springer, 2003.
16. A. K. McIver. Quantitative program logic and expected time bounds in probabilistic distributed algorithms. *Theoretical Computer Science*, 282:191–219, 2002.
17. C. Morgan. Probabilistic temporal logic: qTL. *Lectures on Probabilistic Formal Methods for the 2004 RSISE Logic Summer School*. Slides available at *http://www.cse.unsw.edu.au/∼carrollm/canberra04/*.
18. C. Morgan and A. McIver. An expectation-transformer model for probabilistic temporal logic. *Logic Journal of the IGPL*, 7(6):779–804, 1999.
19. C. Morgan and A. McIver. pGCL: Formal reasoning for random algorithms. *South African Computer Journal*, 22:14—27, 1999.
20. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
21. C. C. Morgan, A. K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.
22. G. Norman. Analysing randomized distributed algorithms. In *Validation of Stochastic Systems*, volume 2925 of *Lecture Notes in Computer Science*, pages 384–418. Springer-Verlag, 2004.
23. Probabilistic Symbolic Model Checker. *http://www.cs.bham.ac.uk/dxp/prism/*.
24. Ymer. *http://www.cs.cmu.edu/∼lorens/ymer.html.*

Embedding a fair CCS in Isabelle/HOL

Michael Compton

Computer Laboratory,
University of Cambridge,
JJ Thomson Avenue,
Cambridge CB3 0FD, UK
Michael.Compton@cl.cam.ac.uk

Abstract. This paper presents an embedding of Milner's CCS, modified to include fairness constraints. The embedding is of interest due to the use of Higher Order Abstract Syntax (HOAS) to represent the CCS terms. This removes some of the burden of explicit syntax manipulation, but requires well-formedness predicates, to remove exotic terms, and some extra machinery for writing recursive functions over the structure of terms. The main focus of the paper is on the derivation of a fair version of CCS. I modify the fair CCS calculus originally presented by Costa and Stirling [2], to produce a CCS theory in which I can distinguish the strongly- and weakly-fair traces. The theory is derived as a definitional extension of Isabelle/HOL.

1 Introduction

Milner's process calculus CCS [6] is generally presented without any notion of fairness. This is not a problem for most abstract modelling. However, if one wishes to derive temporal properties, especially progress properties, some notion of fairness is required.

As noted by Puhakka and Valmari [11], fairness has received less attention in process algebra than in other areas such as classical model checking or temporal logic. However, there are a number of approaches to fairness in process algebraic verification. For example Puhakka and Valmari have developed a system where fairness constraints can be expressed as temporal logic formulae [11]. Also the proof rules of μCRL incorporate fairness notions [5], which have been used in verification of protocols [3].

Here I will investigate a method for achieving fairness based on the fair CCS semantics presented by Costa and Stirling [2]. Their goal was to define an operational semantics in which only the fair execution sequences could be derived. Their fair semantics was based on a labelled CCS calculus. In the labelled calculus, one can determine the *live processes* in any CCS term: live processes are those which are ready to contribute to an action. Costa and Stirling gave an operational semantics for the labelled calculus with two levels; the first (the familiar CCS semantics) gave semantics for single steps, the second level filtered out unfair execution sequences by inspecting the set of live processes at each step in the sequence. Thus in second level semantics only the fair execution sequences were derivable.

The approach is not without difficulties. The most salient is that it seems impossible to derive the strongly fair sequences without, in some way, enforcing choices on the execution [4]. I will avoid this unnatural construction by altering the theory.

Costa and Stirling's fair calculus reappeared in Francez's book *Fairness* [4]. The mechanisation in this paper will begin by following Francez's presentation. However, my theory differs in that I do not develop a fair operational semantics. Instead, I give a definition of traces and use the theory to rule out the unfair traces outside the semantics. This choice removes the problem of generating the strongly fair execution sequences.

Fairness. Here fairness may be viewed as ruling out the unreasonable behaviors of CCS processes. For example, an unreasonable execution is one where one component of a distributed system is simply ignored and is never given the opportunity to execute.

This paper will consider two common forms of fairness, *weak* and *strong* fairness. Intuitively, weak fairness states that no action will remain possible indefinitely. Strong fairness states that no action can be possible infinitely often. Clearly all the strongly fair traces are also weakly fair.

CCS. Milner's CCS is a calculus for the abstract description of the interaction (communication) between distributed and concurrent processes [6]. I present a theory of value passing CCS, which includes mutually recursive functions. Mechanisations of the basic theory have appeared before; perhaps the most closely related is Monica Nesi's HOL embedding [7,8,9], though the treatment of recursion differs significantly. I follow Röckl et al. [12] in defining predicates for well-formed processes, which rule out exotic terms.

Isabelle/HOL. This theory is presented as a definitional extension of Isabelle/HOL [10]. Isabelle has powerful tools for generating readable formal theories. All definitions and theorems in this documents are imported directly and automatically by Isabelle from the formal theory, so you can be sure that Isabelle accepts them as theorems. Isabelle's LaTeX translations have been used to make the theory look, as far as possible, like ordinary semantics, so I give no introduction to Isabelle. The symbol $\longrightarrow$ is logical implication (I use a simlar symbol, $t \xrightarrow{l} t'$, in the semantics but there should be no confusion). Note that free variables in theorems are implicitly universally quantified.

Paper Outline. Section 2 presents the syntax (Section 2.1) and semantics (Section 2.3) of the CCS embedding. I also describe efforts in writing recursive functions over the CCS terms (Section 2.2). Labelled CCS is introduced (Section 3) and I show how to determine the live processes (Section 3.2). Using the definition of live processes I show how to define the fair traces of CCS terms (Section 3.3). Next I give predicates describing well-formed processes (Section 4.1) and derive some correctness properties of the embedding (Section 4.2). Finally I conclude with a brief discussion of this work (Section 5).

2 CCS Language and Semantics

Assume some fixed value domain, ranged over by x, y and z, which acts as a parameter to the theory.

Assume a set of value carrying actions which are partitioned into two sets, those which may synchronise and those that don't require synchronisation. For the synchronising actions write $c?x$ for input on channel c, $c!x$ for output, and $c_\tau x$ for the synchronisation of the two. For the non-synchronising actions write c for simple actions, $c\ x$ for actions with values, and also include the τ action. Let a, a_1, a', etc. range over actions and c, d, etc. range over channel names.

The function *sync* determines if two actions can synchronise, there are only two successful cases.

$$\begin{aligned} \mathit{sync}\ c!x\ c?x &= \mathit{Some}\ c_\tau x \\ \mathit{sync}\ c?x\ c!x &= \mathit{Some}\ c_\tau x \end{aligned}$$

2.1 Syntax of CCS terms

The terms of the CCS embedding are defined by the following grammar:

$$\begin{aligned} t = \mathsf{nil} \mid a \cdot t \mid c!x \cdot t \mid c\ ?\ t \mid t_1 + t_2 \mid t_1 \parallel t_1 \mid \\ t \backslash L \mid\ X \mid t\ x \mid\ t_f \mid \mathsf{fix}\ X\ .\ \mathit{recs} \end{aligned}$$

The terms nil, *sum* $t_1 + t_2$, *parallel* (or *product*) $t_1 \parallel t_1$ and *restriction* $t \backslash L$ appear in most presentations of CCS. Restriction is used to force processes to synchronise. In the *prefix* term $a \cdot t$ the action a may be any of the non-synchronising actions. The *send* term $c!x \cdot t$ can perform action $c!x$ and then continue as the process term t.

The *abstraction* term t_f is a function from values to processes. It is used to allow Isabelle to handle substitution at the meta level, rather than encoding it explicitly. This sort of abstraction is called Higher Order Abstract Syntax (HOAS) and is a well known way of handling issues of syntax and substitution. The *application* term, $t\ x$, applies value x to term t. Obviously application only makes sense if t is an abstraction. Abstraction and application work together in handling the *input* term, written $c\ ?\ t$. Here we expect t to be an abstraction and this term may receive a value, say x, on channel c (emitting the action $c?x$) and then continue as the process $t\ x$.

The *process variable* X and *recursive definitions* $\mathit{fix}\ X\ .\ \mathit{recs}$ allow the creation of mutually recursive terms. These terms, unlike abstractions, can produce infinite behaviour. Here *recs* is a family of recursive terms: in Isabelle it is a list of pairs of process variables and terms. Essentially the variable may be viewed as the name of a function and the term viewed as the function code. For example, the term

$$\mathit{fix}\ X\ .\ [\ (X,\ c\ ?\ (\lambda\ x.\ \tau\cdot(Y\ x))),\ (Y,\ (\lambda\ x.\ d!x\cdot(\mathit{nil}\ + X)))],$$

can input a value on channel c, output the same value on d, and then either terminate or recurse over the same behaviour. As Milner notes [6], it is theoretically more convenient to have recursions defined as above; however, they are easier to read if written as

$$\begin{array}{lcl} X & \equiv & c\ ?\ \ (\lambda x.\ \tau\ \cdot\ \ Y\ x) \\ Y\ x & \equiv & d!x\ \cdot\ \mathit{nil}\ +\ \ X. \end{array}$$

I will only write mutually recursive terms using the $\mathit{fix}\ X\ .\ \mathit{recs}$ syntax.

Using HOAS has relieved the burden of dealing with substitution or environments for input terms. However, this abstraction raises problems of its own. Firstly the terms above include many so called exotic terms, which could not be created in CCS. For example the terms
$P \equiv\ (\lambda x.\ \mathit{if}\ x = y\ \mathit{then}\ c!x\ \cdot\ t_1\ \mathit{else}\ \tau\ \cdot\ t_2)$ and
$Q \equiv c\ ?\ \ (\lambda x.\ x!x\ \cdot\ t_1)$
are not able to be expressed in the usual presentation of CCS. P is not expressible in CCS because it is built from the HOL conditional statement, which is not part of CCS. Q behaves more like a π-calculus term than a CCS term. The second difficulty with HOAS is that, writing functions over the terms has become more difficult, care needs to be taken if we ever need to peek inside an abstraction.

I discuss exotic terms further in Section 4.1. Next I show how to write functions over the CCS terms.

2.2 Writing recursive functions over CCS terms

In Higher Order Logic all functions must terminate for all inputs, but the termination of functions written over the syntax above might not be obvious. In fact, Isabelle already accepts primitive recursive functions where the abstraction case is of the form $f\ (t_f) = (\lambda\ x.\ f\ (t_f\ x))$. However, for total recursive functions (Isabelle's *recdef* functions) the user is required to establish the termination. Here we must establish that recursions like $f\ (f\ (f\ (t_f))))\ldots$ eventually terminate.

For functions defined over the structure of terms the usual method is to show that the size of the terms decreases on each recursive call. Here this is not appropriate as it is not clear how to define a size function for abstractions. Instead, I create an ordering on terms, show that the ordering is well-founded and then use the well-founded ordering as the measure that terms decrease with each recursive call.

The ordering turns out to be quite natural. Begin by defining a predecessor relation over terms. Most of the cases are obvious, for example, *nil* is a predecessor of everything except itself, t is a predecessor of $c!x \cdot t$, terms t_1 and t_2 are predecessors of $t_1 + t_2$. Take the predecessors of the abstraction t_f as all terms in the set $\bigcup_x t_f\ x$. The transitive closure of this relation yields a well-founded ordering on terms. The well-foundedness proof proceeds by simultaneous induction over terms, lists of terms and process variable-term pairs.

Proofs over the syntax of terms require the simultaneous induction mentioned above, this is due to the recursive terms which are defined using lists and pairs. However, using the ordering on terms we can derive a principle of well-founded induction which is easier to use.

With this machinery in place Isabelle will admit the definition of functions such as substitution of mutually recursive definitions.

$$\begin{array}{lcl}
\mathit{subst}\ (\mathsf{nil},\ S,\ \mathit{MRL}) & = & \mathsf{nil} \\
\mathit{subst}\ (l \cdot t,\ S,\ \mathit{MRL}) & = & l \cdot \mathit{subst}\ (t,\ S,\ \mathit{MRL}) \\
\mathit{subst}\ (c!a \cdot t,\ S,\ \mathit{MRL}) & = & c!a \cdot \mathit{subst}\ (t,\ S,\ \mathit{MRL}) \\
\mathit{subst}\ (c\ ?\ t,\ S,\ \mathit{MRL}) & = & c\ ?\ \mathit{subst}\ (t,\ S,\ \mathit{MRL}) \\
\mathit{subst}\ (t_1 + t_2,\ S,\ \mathit{MRL}) & = & \mathit{subst}\ (t_1,\ S,\ \mathit{MRL}) + \mathit{subst}\ (t_2,\ S,\ \mathit{MRL}) \\
\mathit{subst}\ (t_1 \parallel t_2,\ S,\ \mathit{MRL}) & = & \mathit{subst}\ (t_1,\ S,\ \mathit{MRL}) \parallel \mathit{subst}\ (t_2,\ S,\ \mathit{MRL}) \\
\mathit{subst}\ (t\backslash L,\ S,\ \mathit{MRL}) & = & \mathit{subst}\ (t,\ S,\ \mathit{MRL})\backslash L \\
\mathit{subst}\ (\ X,\ S,\ \mathit{MRL}) & = & \\
\multicolumn{3}{c}{\mathsf{if}\ X \in S \wedge \mathit{fp\text{-}mem}\ X\ \mathit{MRL}\ \mathsf{then}\ \mathsf{fix}\ X\ .\ \mathit{MRL}\ \mathsf{else}\ \ X} \\
\mathit{subst}\ (t\ a,\ S,\ \mathit{MRL}) & = & \mathit{subst}\ (t,\ S,\ \mathit{MRL})\ a \\
\mathit{subst}\ (\ t_f,\ S,\ \mathit{MRL}) & = & \ (\lambda x.\ \mathit{subst}\ (t_f\ x,\ S,\ \mathit{MRL})) \\
\mathit{subst}\ (\mathsf{fix}\ X\ .\ \mathit{recs},\ S,\ \mathit{MRL}) & = & \\
\multicolumn{3}{c}{\mathsf{fix}\ X\ .\ \mathit{map}\ (\lambda(Y,\ t_Y).\ (Y,\ \mathit{subst}\ (t_Y,\ S - \mathit{list\text{-}fst}\ \mathit{recs},\ \mathit{MRL})))\ \mathit{recs}}
\end{array}$$

subst is a somewhat lazy definition of substitution: in the recursive case I make no attempt to avoid variable capture. However, this doesn't turn out to be a problem as for the bulk of this theory I only consider closed terms, in which there are no free variables to capture.

2.3 CCS LTS semantics

The labelled transition system semantics presented in Figure 1 is a standard rendering of the semantics for CCS. Each step in the reduction is labelled with the action that was performed.

Note that while input is handled in HOAS, I explicitly deal with substitutions for the recursive terms. This is a bit incongruous but it seems that explicit substitution for recursive terms is the correct choice. In the first instance, fix terms are actually required to create infinite behaviour: Section 2.2 shows that abstraction terms can not create infinite behaviour. Further substitution is handled explicitly for these terms as they require a very precise treatment in order to make any sense. Section 5 evaluates the use of HOAS.

$$a \cdot t \xrightarrow{a} t \qquad c!x \cdot t \xrightarrow{c!x} t \qquad c\ ?\ t_f \xrightarrow{c?x} t_f\ x \qquad \frac{t_f\ x \xrightarrow{l} t'}{t_f\ x \xrightarrow{l} t'}\textsc{App}$$

$$\frac{t_1 \xrightarrow{l} {t_1}'}{t_1 + t_2 \xrightarrow{l} {t_1}'}\textsc{SumL} \qquad \frac{t_2 \xrightarrow{l} {t_2}'}{t_1 + t_2 \xrightarrow{l} {t_2}'}\textsc{SumR} \qquad \frac{t \xrightarrow{l} t' \qquad l \notin L}{t\backslash L \xrightarrow{l} t'\backslash L}\textsc{Res}$$

$$\frac{t_1 \xrightarrow{l} {t_1}'}{t_1 \parallel t_2 \xrightarrow{l} {t_1}' \parallel t_2}\textsc{ParL} \qquad \frac{t_2 \xrightarrow{l} {t_2}'}{t_1 \parallel t_2 \xrightarrow{l} t_1 \parallel {t_2}'}\textsc{ParR}$$

$$\frac{t_1 \xrightarrow{l_1} {t_1}' \qquad t_2 \xrightarrow{l_2} {t_2}' \qquad \mathit{sync}\ l_1\ l_2 = \mathit{Some}\ l_s}{t_1 \parallel t_2 \xrightarrow{l_s} {t_1}' \parallel {t_2}'}\textsc{ParS}$$

$$\frac{[(a,\ b){\in}\mathit{recs}\ .\ a = X] = [(X,\ t_X)] \qquad \mathit{subst}\ (t_X,\ \mathit{list\text{-}fst\ recs},\ \mathit{recs}) \xrightarrow{l} {t_X}'}{\mathsf{fix}\ X\ .\ \mathit{recs} \xrightarrow{l} {t_X}'}\textsc{MRec}$$

$$\frac{[(a,\ b){\in}\mathit{recs}\ .\ a = X] = [(X,\ \ t_f)] \qquad \mathit{subst}\ (\ t_f,\ \mathit{list\text{-}fst\ recs},\ \mathit{recs})\ x \xrightarrow{l} {t_f}'}{\mathsf{fix}\ X\ .\ \mathit{recs}\ x \xrightarrow{l} {t_f}'}\textsc{MRecApp}$$

Fig. 1. Isabelle rules for the CCS semantics.

3 A Fair CCS

The semantics just presented does not respect any notion of fairness. For example, consider the process

$$\mathsf{fix}\ X\ .\ [(X,\ a\cdot\ \ X)] \parallel \mathsf{fix}\ Y\ .\ [(Y,\ b\cdot\ \ Y)]$$

which performs sequences of a and b actions. There are many valid traces of such a term, most will interleave a and b actions. However, one valid trace is a, a, a, a, a, a, ... which only performs actions in the term on the left. Clearly this behaviour isn't fair to the term on the right: in a fair execution we would expect that a b would eventually be performed.

I now define *process fairness* for CCS terms. It is not the only possible choice of fairness for CCS. Apt et al. [1] lists 6 possible notions of fairness for CSP, similar notions could be considered for CCS. For example one could study notions like *channel fairness* where we require that actions on certain channels would not be indefinitely delayed.

To remove any confusion, in this paper I will use *term* to describe a CCS term and *process* to describe the notion of a process within a CCS term (this concept will be formally defined shortly). Process fairness requires each individual process within a term to eventually proceed if it is able to do so.[1]

[1] Note that the model of fairness presented here is not *non-deterministically fair*: the process

$$\mathsf{fix}\ X\ .\ [(X,\ b\cdot\ \ X + a\cdot\ \ X)]$$

The development now follows the ideas for labelled processes and live processes presented by Frances [4], which differs slightly from the original presentation by Costa and Stirling [2].

3.1 Labelling Processes

In any CCS term there may be many constituent terms which are *live* – capable of proceeding by performing some action. Here I will talk of the *processes* and *live processes* of a CCS term, by which, essentially, I mean the sub-terms obtained by breaking apart sum and parallel. The immediate problem in achieving process fairness for CCS is to determine which processes are live. The solution presented here is to label every process with a unique label. At each step in an execution sequence we can inspect the structure of the whole term and determine which of these labelled processes are live.

I define a new grammar for labelled processes

$$\begin{array}{l} t = \mathsf{nil}_l \mid a_l \cdot t \mid (c!x)_l \cdot t \mid c?_l\ t_f \mid t_1 +_l t_2 \mid t_1 \parallel t_1 \\ \quad t \setminus L \mid X_l \mid t\ x \mid \ \ t_f \mid \mathsf{fix}\ X_l.\ recs \end{array}$$

The terms are similar to the original grammar except some are labelled with natural numbers. Immediately the processes of a term can be defined.

$$\begin{array}{lcllcl} procs\ \mathsf{nil}_l & = & \{l\} & procs\ t_1 \parallel t_2 & = & procs\ t_1 \cup procs\ t_2 \\ procs\ a_l \cdot t & = & \{l\} & procs\ t \setminus L & = & procs\ t \\ procs\ (c!x)_l \cdot t & = & \{l\} & procs\ X_l & = & \{\} \\ procs\ c?_l\ t & = & \{l\} & procs\ t\ x & = & procs\ t \\ procs\ t_1 +_l t_2 & = & \{l\} & procs\ (\ t_f) & = & \bigcup_x procs\ (t_f\ x) \\ & & \multicolumn{4}{l}{\mathsf{if}\ \exists\, t_X.\ [(a,\ b){\in}recs\ .\ a = X] = [(X,\ t_X)]} \\ procs\ \mathsf{fix}\ X_l.\ recs & = & \multicolumn{4}{l}{\mathsf{then}\ procs\ (snd\ (hd\ [(a,\ b){\in}recs\ .\ a = X]))} \\ & & \multicolumn{4}{l}{\mathsf{else}\ \{\}} \end{array}$$

Some restrictions will be required of the labelling. For example, if all labels are, say 10, then we will be unable to distinguish individual processes. I will require labelled terms to be well-labelled, essentially this requires a term to be labelled like a binary tree. Figure 2 explains what it means for a term to be well-labelled, giving an example with the term $a_1 \cdot t_1 +_{11} b_{112} \cdot t_2 \parallel t_3$.

Labelling terms as binary trees is not difficult, *label* takes an unlabelled term and labels it. I also define the function *relabel* which takes a labelled process and relabels it using exactly the same scheme.

will be able to perform an infinite sequence of a's or an infinite sequence of b's.

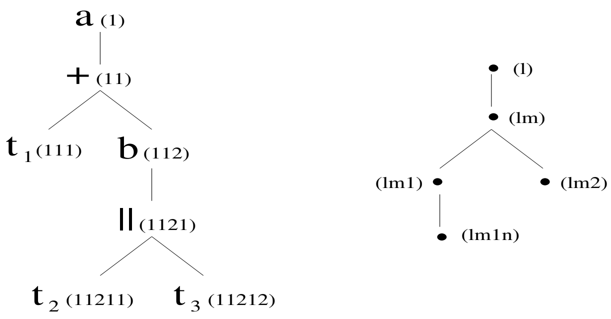


Fig. 2. Well-labelled terms: The term shown as a tree on the left is well labelled. However we needn't be so strict in fact all we require is that labels higher in the tree are prefixes of lower labels, of course ensuring that the tree structure is maintained with 1's and 2's, as shown in the right hand tree (assume that l, m and n are arbitrary numbers).

$$\begin{array}{lll}
\mathit{label}\ (\mathsf{nil},\ l) & = & \mathsf{nil}_l \\
\mathit{label}\ (a \cdot t,\ l) & = & a_l \cdot \mathit{label}\ (t,\ l * 10 + 1) \\
\mathit{label}\ (c!a \cdot t,\ l) & = & (c!a)_l \cdot \mathit{label}\ (t,\ l * 10 + 1) \\
\mathit{label}\ (c\ ?\ t_f,\ l) & = & c?_l\ \mathit{label}\ (t_f,\ l * 10 + 1) \\
\mathit{label}\ (t_1 + t_2,\ l) & = & \mathit{label}\ (t_1,\ l * 10 + 1) +_l \mathit{label}\ (t_2,\ l * 10 + 2) \\
\mathit{label}\ (t_1 \parallel t_2,\ l) & = & \mathit{label}\ (t_1,\ l * 10 + 1) \parallel \mathit{label}\ (t_2,\ l * 10 + 2) \\
\mathit{label}\ (t \backslash L,\ l) & = & \mathit{label}\ (t,\ l) \setminus L \\
\mathit{label}\ (\ X,\ l) & = & X_l \\
\mathit{label}\ (t\ a,\ l) & = & \mathit{label}\ (t,\ l)\ a \\
\mathit{label}\ (\ t_f,\ l) & = & (\lambda x.\ \mathit{label}\ (t_f\ x,\ l)) \\
\mathit{label}\ (\mathsf{fix}\ X\ .\ \mathit{recs},\ l) & = & \\
\multicolumn{3}{l}{\qquad \mathsf{fix}\ X_l.\ \mathit{map}\ (\lambda(Y,\ t_Y).\ (Y,\ \mathit{label}\ (t_Y,\ l * 10 + 1)))\ \mathit{recs}}
\end{array}$$

Using a well-founded induction for labelled terms I can prove that *label* and *relabel* result in a well-labelled terms.

Lemma 1. *well-labelled* (*label* $(t,\ l)$) l

Lemma 2. *well-labelled* (*relabel* $(t,\ l)$) l

I also define an *unlabel* function, which simply strips labels from labelled terms. The two functions *label* and *unlabel* satisfy the obvious identity.

Lemma 3. *unlabel* (*label* $(t,\ l)) = t$

An important property of well-labelled terms is that no two processes are given the same label.

Lemma 4. If *well-labelled* $t\ l$ then *unique-labelling* $(t,\ \{\})$.

Note that *label* gives each term in the *recs* list of *fix* X_l. *recs* the same labelling. So the entire list doesn't have a unique labelling. However, the only term that gets executed is the one corresponding to X, see Figure 1 (the other terms will be relabelled before execution, see below). So I only require that its labelling be unique.

Substitution can be defined for labelled terms in the same way as for unlabelled terms. The only difference between the two definitions is in the case of process variables where the terms are relabelled after substitution.

```
$substl\ (X_l,\ S,\ MRL) =$
$(\mathit{if}\ X \in S \wedge fp\text{-}mem\ X\ MRL$
 $\mathit{then}\ \mathit{fix}\ X_l.\ map\ (\lambda(Xa,\ F).\ (Xa,\ relabel\ (F,\ l * 10 + 1)))\ MRL$
 $\mathit{else}\ X_l)$
```

The reason terms are relabelled is so that each recursive invocation has different labels, and is thus considered as a different process. This means that we are fair to a term every time it occurs not just the first time it occurs.

Substitution in a well-labelled term results in a well-labelled term.

Lemma 5. If *well-labelled* $t\ l$ then *well-labelled* $(substl\ (t,\ S,\ recs))\ l$.

A LTS semantics for labelled processes, where reductions are written $t \overset{a}{\Longrightarrow} t'$, is defined in the same manner as Figure 1: the labels are simply ignored in the labelled semantics. By induction over the relevant transition system I can formally show that the two semantics coincide.

Lemma 6. If $t \xrightarrow{a} t'$ then $\exists\, t''.\ label\ (t,\ l) \overset{a}{\Longrightarrow} t''$.

Lemma 7. If $t \overset{a}{\Longrightarrow} t'$ then *unlabel* $t \xrightarrow{a}$ *unlabel* t'.

Finally well-labelling is preserved by reduction in the semantics.

Lemma 8. If $t \overset{a}{\Longrightarrow} t'$ and *well-labelled* $t\ l$ then *well-labelled* $t'\ l$.

As further example of labelling take the term

$$A \equiv d!x \cdot \mathit{nil} \parallel c!y \cdot a \cdot \ \ Y + \tau \cdot \ \ X$$

and label it to get

$$label(A,\ 1) = (d!x)_{11} \cdot \mathit{nil}_{111} \parallel (c!y)_{121} \cdot a_{1211} \cdot Y_{12111} +_{12} \tau_{122} \cdot X_{1221}.$$

3.2 Determining the live processes

Now that the process terms have been labelled, the live processes can be determined. The live processes are those which can contribute to an action. Capturing this idea is complicated by the interaction of parallel and restriction. Consider the term

$$(c!x)_l \cdot t_1 \parallel c?_{l'}\ t_2 \setminus L$$

If L restricts $c?x$ and $c!x$ actions, then each side of this parallel is unable to move without help from the other. Thus both l and l' are live: they can contribute to an $c_\tau x$ action, but not to $c?x$ or $c!x$ actions. From this we can see that in determining the live processes of a term, both the restrictions and the possible actions of both sides of parallel terms need to be considered.

The function *next-acts* computes all possible next actions (ignoring synchronisations) of a term (only some cases shown).

$$\begin{array}{lcl}
\textit{next-acts}\ a_l \cdot t & = & \{a\} \\
\textit{next-acts}\ (c!x)_l \cdot t & = & \textit{all-values}\ c!x \\
\textit{next-acts}\ c?_l\ t & = & \textit{all-values}\ c?\textit{arbitrary} \\
\textit{next-acts}\ t_1 +_l t_2 & = & \textit{next-acts}\ t_1 \cup \textit{next-acts}\ t_2 \\
\textit{next-acts}\ t_1 \parallel t_2 & = & \textit{next-acts}\ t_1 \cup \textit{next-acts}\ t_2 \\
\textit{next-acts}\ t \setminus L & = & \textit{next-acts}\ t - L \\
\textit{next-acts}\ \ t_f\ x & = & \textit{next-acts}\ (t_f\ x)
\end{array}$$

Take the *co-actions* of action a to be $\{a' \mid \exists\, a_s.\ \textit{sync}\ a\ a' = \textit{Some}\ a_s\}$, this notion can be extended to the co-actions of a set of actions in the obvious way. The function *live-procs* determines the set of live processes under a restriction set. Note how passing under a restriction increases the restriction set, while the co-actions of parallel processes can decrease the restriction set.

$$\begin{array}{lcl}
\textit{live-procs}\ (a_l \cdot t,\ A) & = & \mathsf{if}\ a \notin A\ \mathsf{then}\ \{l\}\ \mathsf{else}\ \{\} \\
\textit{live-procs}\ ((c!x)_l \cdot t,\ A) & = & \mathsf{if}\ c!x \notin A\ \mathsf{then}\ \{l\}\ \mathsf{else}\ \{\} \\
\textit{live-procs}\ (c?_l\ t,\ A) & = & \mathsf{if}\ \textit{all-values}\ c?\textit{arbitrary} \cap A = \{\}\ \mathsf{then}\ \{l\}\ \mathsf{else}\ \{\} \\
\textit{live-procs}\ (t_1 +_l t_2,\ A) & = & \begin{array}{l}\mathsf{if}\ \textit{live-procs}\ (t_1,\ A) \cup \textit{live-procs}\ (t_2,\ A) \neq \{\} \\ \mathsf{then}\ \{l\}\ \mathsf{else}\ \{\}\end{array} \\
\textit{live-procs}\ (t_1 \parallel t_2,\ A) & = & \begin{array}{l}\textit{live-procs}\ (t_1,\ A - \textit{co-actions}\ (\textit{next-acts}\ t_2)) \cup \\ \textit{live-procs}\ (t_2,\ A - \textit{co-actions}\ (\textit{next-acts}\ t_1))\end{array} \\
\textit{live-procs}\ (t \setminus L,\ A) & = & \textit{live-procs}\ (t,\ A \cup L) \\
\textit{live-procs}\ (\ t_f\ x,\ A) & = & \textit{live-procs}\ (t_f\ x,\ A) \\
\textit{live-procs}\ (\mathsf{fix}\ X_l.\ \textit{recs},\ A) & = & \begin{array}{l}\mathsf{if}\ \exists\, t_X.\ [(a,\ b){\in}\textit{recs}\ .\ a = X] = [(X,\ t_X)] \\ \mathsf{then}\ \textit{live-procs}\ (\textit{snd}\ (\textit{hd}\ [(a,\ b){\in}\textit{recs}\ .\ a = X]),\ A) \\ \mathsf{else}\ \{\}\end{array}
\end{array}$$

The live processes of a term represents exactly the set of processes that can perform some action. This claim is formally verified in section 4.

Above we have already computed the labelling for the process A as

$$\textit{label}(A,\ \mathit{1}) = (d!x)_{11} \cdot \mathsf{nil}_{\mathit{111}} \parallel (c!y)_{121} \cdot a_{1211} \cdot Y_{12111} +_{12} \tau_{122} \cdot X_{1221}.$$

Now we can also compute the live processes of A.

$$\textit{live-procs}(\textit{label}(A,\ \mathit{1}),\ \{\})\ =\ \{\mathit{12},\ \mathit{11}\}$$

Of course the live processes are a subset of all processes.

Lemma 9. *live-procs* $(t,\ A) \subseteq$ *procs* t

3.3 Weak and Strong fairness

Using the above definitions I define weak and strong fairness on the traces of labelled-CCS terms. I define traces in a standard way. An action trace is an infinite sequence $a_1,\ a_2,\ a_3\ \ldots$ of actions, while a term trace is an infinite sequence $t_1,\ t_2,\ t_3\ \ldots$ of terms. I represent both in Isabelle with a generic type of infinite traces.

$$\alpha\ \mathit{trace} :: \mathit{nat} \Rightarrow \alpha$$

A trace of a labelled term is a pair of an action trace and a labelled term trace.

$$\mathit{labelled\text{-}ccs\text{-}trace} :: \mathit{labelled\text{-}ccs\text{-}term\ trace} \times \mathit{action\ trace}$$

For a trace *tr* to represent an execution in the semantics of term *t* I require the following condition to hold

$$\begin{array}{l}
\mathit{is\text{-}ccs\text{-}labelled\text{-}trace}\ t\ \mathit{tr}\ \ = \\
(\mathit{labelled\text{-}terms}\ \mathit{tr})\ \mathit{0} = t\ \wedge \\
(\forall\, i.\ (\mathit{labelled\text{-}terms}\ \mathit{tr})\ i \xrightarrow{\mathit{actions}\ \ \mathit{tr}\ \ i} (\mathit{labelled\text{-}terms}\ \mathit{tr})\ (i\ +\ \mathit{1})\ \vee \\
\qquad (\mathit{labelled\text{-}terms}\ \mathit{tr})\ i = (\mathit{labelled\text{-}terms}\ \mathit{tr})\ (i\ +\ \mathit{1})\ \wedge\ \mathit{actions}\ \mathit{tr}\ i = \tau)
\end{array}$$

labelled-terms and *actions* project out the term trace and the action trace respectively. I add stuttering, the extra τ steps, to ensure that all traces are infinite.

The definitions of weak and strong fairness are now immediate.[2] A trace satisfies weak fairness if no process becomes live and then remains live forever, i.e. all processes eventually lose their liveness. A trace *tr* satisfies weak fairness iff

$$\forall\, p\ i.\ \exists\, k.\ i < k\ \wedge\ p \notin \mathit{live\text{-}procs}\ ((\mathit{labelled\text{-}terms}\ \mathit{tr})\ k,\ \{\}).$$

A trace is strongly fair if no process is live infinitely often, i.e. no process can become live, lose its liveness, become live again, etc. forever. Formally we require that for every process there be some point beyond which it never again becomes live. A trace *tr* satisfies strong fairness iff

$$\forall\, p.\ \exists\, i.\ \forall\, k.\ i \leq k \longrightarrow p \notin \mathit{live\text{-}procs}\ ((\mathit{labelled\text{-}terms}\ \mathit{tr})\ k,\ \{\}).$$

[2] Similarly the concept of *minimal progress* could be defined. A trace satisfies minimal progress if whenever something can happen then eventually something does. This means that we never idle indefinetely with τ actions when some useful action could be performed. Formally we would require

$$\forall\, i\ a\ t'.\ (\mathit{labelled\text{-}terms}\ \mathit{tr})\ i \xrightarrow{a} t' \longrightarrow (\exists\, j.\ i \leq j\ \wedge\ (\mathit{labelled\text{-}terms}\ \mathit{tr})\ j \xrightarrow{\mathit{actions}\ \ \mathit{tr}\ \ j} (\mathit{labelled\text{-}terms}\ \mathit{tr})\ (j\ +\ \mathit{1})).$$

Note that, in both cases, processes are not required to perform an action, just to lose their liveness. A live process that performs an action is immediately removed from both the processes and the live processes. However, a process may remain in the set of processes but no longer be live if, for example, a communication partner is removed by some other action.

4 Correctness of fairness definitions

I now show that the definitions given above satisfy some properties that we would expect. However first I will restrict the set of terms. Too many terms are admitted by the definition given in Section 3.1. In this section only *consistent* terms are considered. A consistent term is

- well-formed (Section 4.1),
- well-labelled (Section 3.1 and Figure 2),
- guarded,
- and has well-formed recursive terms.

These are quite natural restrictions, in fact, they are meant to denote the set of terms that most CCS theories rely on. In guarded terms recursive variables only occur below some prefixing action. For example, $(c!x)_l \cdot X_{l'}$ is guarded but $X_{l'} +_l t$ is not. Most CCS theories require guarded terms. A term has well-formed recursions if all recursive sub-terms are closed, i.e. for all sub-terms $\mathit{fix}\ X_l.\ \mathit{recs}$ we have $\forall\ t_X.\ t_X \in \mathit{set}\ (\mathit{snd}\ \mathit{recs}) \longrightarrow \mathit{fv}\ t_X \subseteq \mathit{set}\ (\mathit{fst}\ \mathit{recs})$ – this restriction is, in fact, required by the definition of *subst* which doesn't avoid variable capture; however, requiring closed terms is a also sensible choice.

Well-formed terms are those which do not have exotic terms introduced by HOAS, I explain how to ensure such terms in the next section.

4.1 Removing exotic terms

It may not be immediately clear that the terms defined in Sections 2.1 and 3.1 represent the usual definition of CCS terms. Indeed, the use of HOAS has introduced many terms which are not terms in CCS. These exotic terms can be a problem when trying to prove properties that should be true of all CCS terms: the exotic terms may not satisfy the same properties as regular CCS terms. Further, if the terms do not exactly represent CCS terms then one could question if this is indeed a theory of CCS. In this section I will rule out exotic terms to recover a more natural set of CCS terms.

I introduce predicates, based on similar predicates for the π-calculus defined by Röckl et al. [12], to rule out exotic terms. The predicate ***wf***, see Figure 3, defines the set of well formed process terms, while the second order predicate ***wfa*** defines well formed process abstractions.

Though I introduced consistent terms to represent the usual definition of CCS terms, I make no attempts to prove this relationship. The essential property of well-formed abstractions is that, for any value the set of processes remains the same.

Lemma 10. If $\boldsymbol{wfa}\ t_f$ then $\mathit{procs}\ (t_f\ x) = \mathit{procs}\ (t_f\ y)$.

The relationship between the two predicates is revealed by the following lemma.

Lemma 11. If $\boldsymbol{wfa}\ t_f$ then $\boldsymbol{wf}\ (t_f\ x)$.

$$\boldsymbol{wf}\ \mathsf{nil}_l \qquad \frac{\boldsymbol{wf}\ t}{\boldsymbol{wf}\ (c!x)_l \cdot t} \qquad \frac{\boldsymbol{wfa}\ t_f}{\boldsymbol{wf}\ c?_l\ \ t_f}$$

$$\frac{\boldsymbol{wf}\ t_1 \qquad \boldsymbol{wf}\ t_2}{\boldsymbol{wf}\ t_1 +_l t_2} \qquad \frac{\boldsymbol{wf}\ t_1 \qquad \boldsymbol{wf}\ t_2}{\boldsymbol{wf}\ t_1 \parallel t_2} \qquad \frac{\boldsymbol{wfa}\ t_f}{\boldsymbol{wf}\ (\ t_f)}$$

$$\boldsymbol{wfa}\ (\lambda x.\ \mathsf{nil}_l) \qquad \frac{\boldsymbol{wfa}\ t_f}{\boldsymbol{wfa}\ (\lambda x.\ (c!f\ x)_l \cdot t_f\ x)}$$

$$\frac{\boldsymbol{wfa}\ t_{f1} \qquad \boldsymbol{wfa}\ t_{f2}}{\boldsymbol{wfa}\ (\lambda x.\ t_{f1}\ x +_l t_{f2}\ x)} \qquad \frac{\boldsymbol{wfa}\ t_{f1} \qquad \boldsymbol{wfa}\ t_{f2}}{\boldsymbol{wfa}\ (\lambda x.\ t_{f1}\ x \parallel t_{f2}\ x)}$$

$$\frac{\forall x.\ \boldsymbol{wfa}\ (t_f\ x) \qquad \forall x.\ \boldsymbol{wfa}\ (\lambda y.\ t_f\ y\ x)}{\boldsymbol{wfa}\ (\lambda x.\ \ (t_f\ x))}$$

Fig. 3. Part of the definitions of well-formed terms, ***wf***, and well-formed process abstractions, ***wfa***.

4.2 Francez's correctness lemmas

Francez proposes three key properties that go some way to showing the correctness of this development [4, pg. 248, lemmas i, ii & iii]

Theorem 1. If *consistent* t and $t \overset{a}{\Longrightarrow} t'$ and $p \in \mathit{procs}\ t - \mathit{live\text{-}procs}\ (t, \{\})$ then $p \in \mathit{procs}\ t'$.

Theorem 2. If *consistent* t and $p \in \mathit{live\text{-}procs}\ (t, \{\})$ then $\exists\, a\ t'.\ t \overset{a}{\Longrightarrow} t' \wedge p \notin \mathit{procs}\ t'$.

Theorem 3. If *consistent* t and $t \overset{a}{\Longrightarrow} t'$ then $\exists\, p.\ p \in \mathit{live\text{-}procs}\ (t, \{\}) \wedge p \notin \mathit{procs}\ t'$.

Theorem 1 says that processes that aren't live don't participate in transitions. Theorem 2 shows that live process must be able to participate in a transition. Theorem 3 shows that a every transition is performed by some live process. These rules show that the live processes match with the intuition of a live process, i.e. that a live process is a subterm that is ready to perform an action.

Theorems 3 and 1 are proved by induction over the labelled semantics, while Theorem 2 is proved by well founded induction over labelled terms. The proof of Theorem 2 is the most difficult. Essentially this is because after induction over term t in each case the inductive hypothesis claims that Theorem 2 holds for all smaller terms, but in the case for recursive terms I am required to argue about terms after substitution (see the rules MRec and MRecApp in Figure 1), which are larger. So first a number of key lemmas relating terms to their larger substituted counterparts need to be derived.

In fact none of the lemmas are provable in their current form. Each contains a reference to a constant, the empty set {}. Inductive proofs often require one to first abstract away such constants. For example, in order to prove Theorem 3 I must first prove the more abstract property

If $\mathit{consistent}\ t$ and $t \overset{a}{\Longrightarrow} t'$ and $\mathit{wf\text{-}restriction}\ A$ and $a \notin A$ then $\exists\, p.\ p \in \mathit{live\text{-}procs}\ (t,\ A) \wedge p \notin \mathit{procs}\ t'$.

Clearly taking $A = \{\}$ produces theorem 3 above. Similar abstractions need to be found before proving Theorems 1 and 2.

5 Discussion

In this paper I have shown how to embed a fair CCS in Higher Order Logic. I embedded the syntax and semantics for value passing CCS with recursive functions. Following an earlier theory I extended the basic definition of CCS by defining a labelled calculus and showed how to define live processes in this calculus. I used the definition of live processes to encode a definition of the fair traces of the labelled CCS terms. As I have proved a correspondence between the transitions systems of labelled and unlabelled terms (lemmas 7 and 6) I can also talk of the fair executions of ordinary CCS terms, not just the labelled ones.

An interesting feature of the embedding is the attempt to handle substitution and function application at the meta level. The introduction of mutually recursive terms (and the requirement to define substitution) creates a strange marriage of HOAS and substitution. HOAS saved some hassle in dealing with syntax but introduced a few extra issues. The effort associated with dealing with HOAS terms wasn't excessive. After derivation of a well-founded ordering and induction principle they essentially had little impact except the requirement to prove lemmas 10 and 11.

Another approach would be to avoid the syntactic approach and define instead some sort of environment, treating variables as having values in the environment rather than performing substitution. Though undoubtedly such an approach would have it's own drawbacks.

As future work I intend to embed a temporal logic in which I can reason a about fair CCS processes.

References

1. K. R. Apt, N. Francez, and S. Katz. Appraising fairness in distributed languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 189–198, New York, NY, USA, 1987. ACM Press.
2. Gerardo Costa and Colin Stirling. A fair calculus of communicating systems. *Acta Inf.*, 21:417–441, 1984.
3. Wan Fokkink, Jan Friso Groote, Jun Pang, Bahareh Badban, and Jaco van de Pol. Verifying a sliding window protocol in μ-CRL. In *10th Conference on Algebraic Methodology and Software Technology, Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 148–163. Springer-Verlag, 2004. Springer-Verlag.
4. Nissim Francez. *Fairness.* Springer-Verlag New York, Inc., New York, NY, USA, 1986.
5. J. F. Groote and M. A. Reniers. Algebraic process verification. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 17. Elsevier, 2001.
6. R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
7. M. Nesi. Value-passing CCS in HOL. Number 780 in Lecture Notes in Computer Science, pages 352–365. Springer, 1993.
8. M. Nesi. Mechanising a modal logic for value-passing agents in hol. Number 5 in ENTCS, http://www.elsevier.nl/locate/entcs/volume5.html, 1997. Elsevier.
9. M. Nesi. Formalising a value-passing calculus in HOL. *Formal Aspects of Computing*, 11:160–199, 1999.
10. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
11. Antti Puhakka and Antti Valmari. Liveness and fairness in process-algebraic verification. In *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*, pages 202–217, London, UK, 2001. Springer-Verlag.
12. C. Röckl, D. Hirschkoff, and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the π-calculus and mechanizing the theory of contexts. In F. Honsell and M. Miculan, editors, *Proc. FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2001.

What can be Learned from Failed Proofs of Non-Theorems?

Louise A. Dennis and Pablo Nogueira

School of Computer Science and Informaton Technology, University of Nottingham,
Nottingham, NG8 1BB
lad@cs.nott.ac.uk
pni@cs.nott.ac.uk

Abstract. This paper reports an investigation into the link between failed proofs and non-theorems. It seeks to answer the question of whether anything more can be learned from a failed proof attempt than can be discovered from a counter-example. We suggest that the branch of the proof in which failure occurs can be mapped back to the segments of code that are the culprit, helping to locate the error. This process of tracing provides finer grained isolation of the offending code fragments than is possible from the inspection of counter-examples. We also discuss ideas for how such a process could be automated.

The use of mathematical proof to show that a computer program meets its specification has a long history in Computer Science (e.g. [13, 12]). However the techniques and tools are only used in very specialised situations in industry where programmers generally rely on testing and bug reports from users to assess the extent to which a program meets its specification. One of the many reasons to which this poor uptake is attributed is that the final proof will tell you if the program is correct, but failing to find a proof does not, on immediate inspection, help in locating errors. This problem can be particularly severe when using automated proof techniques which generally produce no proof trace in the case of failure. However cases have been reported where the process of attempting a proof by hand has highlighted an error. Anecdotal evidence suggests that errors are located by examining and reflecting on the process of the failed proof attempt.

It is worth noting the comparative success of model checking techniques (e.g. [9]), part of which has been attributed to the fact that model-checkers return counterexamples when they fail. As a result, there has been a great deal of interest recently in the detection of counter-examples for software programs and protocols (e.g. [17]). The purpose of the work reported here was to see if the process of attempting and then failing to find a proof could tell a user anything further about software errors than could be extracted from a counter-example.

We start by outlining our methodology (§1) and give a broad categorisation of the errors we encountered (§2), we then examine some specific examples of buggy programs, the failed proof attempts they generated, how these attempts can be used to assign “blame” to particular segments of the original source code and in some cases to suggest patches(§3). Lastly we discuss ideas for automating this process and some challenges that such a system would face (§4).

1 Methodology

We have opted to study programs produced by novice programmers, specifically undergraduates working on functional programming modules. There were three main reasons why we selected this domain: it is relatively easy to acquire a large number of such programs; they are likely to be well suited to attempts at correctness proofs and easy to translate into the object language of theorem provers; and they tend to be relatively small so the associated correctness proofs can be generated comparatively rapidly allowing us to focus on surveying errors. There are clearly also some problems with selecting such a domain, in particular that the errors produced by novices may be dissimilar to those produced by professional programmers. It is also the case that conclusions drawn about the utility of failed proof attempts in functional programming may not tell us anything about their utility in other paradigms.

We gathered a test bed of such programs and used standard testing to identify those which were incorrect. This process also generated counter-examples for those programs. We then attempted hand proofs in the Isabelle/HOL theorem prover [15, 14] to see if anything further could be learned about the nature of the error above that already learned from the counter-examples.

Our choice of Isabelle was dictated by an interest in extending this work to an automated system to produce program error diagnoses based on failed proof attempts. We have already performed some initial investigations in this area using the proof planning paradigm [5]. IsaPlanner [6] is a proof planning system built on top of Isabelle so this made the combination of Isabelle/IsaPlanner an attractive option. Furthermore our test database contained, almost exclusively, recursive programs suitable for proof by mathematical induction, an area in which the IsaPlanner system specialises.

The proof planning paradigm [2] works by capturing patterns of proof across families of similar problems. The work reported here discusses hand proof attempts in Isabelle performed with the intention of defining patterns of failure which could then be expressed within a proof planning system.

Our initial investigations involved a set of ML and Haskell Programs generated by students at the Universities of Edinburgh[1] and Nottingham[2] respectively. We performed a naive shallow embedding by hand of these programs into Isabelle/HOL. We assumed that all datatypes mapped directly to the corresponding Isabelle datatype (i.e. Isabelle lists behave in the same way as ML lists etc.) and that built-in functions (such as list append, list membership etc.) could also be directly transferred. In some cases we had to edit definitions so they used

[1] The original author of these exercises is unknown. They were set as part of the Computer Science 2 module.

[2] Exercise authored by Dr. Graham Hutton for the Functional Programming module.

structural recursion – for instance when writing functions on the natural numbers students often used `n` and `n - 1` when defining the recursive case, rather than `n + 1` and `n`. However, given the similarity between the Isabelle/HOL theories and ML these assumptions are relatively safe and certainly adequate for an initial investigation. The Haskell programs provided greater challenges. We had issues with the representation of type classes and lazy evaluation and we also found that the match between the functions in the Haskell prelude and Isabelle's existing theories was often insufficient[3]. This reduced our confidence in the conclusions we could draw from the Haskell examples. In this paper we shall only examine ML examples however the Haskell examples do broadly support our conclusions.

In what follows we shall use the `typewriter font family` to represent actual fragments of program code from our corpus and *latex math environment* to represent Isabelle definitions and goals so it is clear when we are talking about actual ML and when we are discussing our translation into Isabelle/HOL.

2 Categorisation of Errors

2.1 Errors in the Basis Case of a Recursive Program

Our previous work investigated the errors that occur in the basis cases of recursive programs [5] and identified two common classes of error in novice programs: that one or more base cases are omitted or that a base case is incorrect. These are distinguished in proofs by reaching a proof goal in the base case of an inductive proof which contains a "user-defined" function which can not be simplified or alternatively by the derivation of $False$. These observations were largely confirmed by this study. In some cases we gained extra information from Isabelle's recursive definition mechanisms. For instance `primrec` (Isabelle's mechanism for defining primitive recursive functions) will issue a warning that there is no equation for a particular constructor if a case is omitted. However, often functions with missing cases required the use of the more general `recdef` definition mechanism[4] which did not issue such warnings. We also found that there were instances where a student would ensure that a sub-function was never called on particular cases and therefore the fact that it was only partially defined was not an issue for the overall correctness of the program.

2.2 Errors in the Recursive Case of a Recursive Program

Initially we observed three different kinds of error in recursive cases. Firstly, the recursive case contained insufficient information (it was embeddable within a

[3] We also had problems with an exercise specification which under-specified the type of bits to integers although it was clear the functions were only to be tested on 0 and 1.

[4] These errors tended to arise when the student was attempting a two step recursive scheme which `primrec` would not allow.

"correct" recursive case), secondly the recursive case contained too much information (a "correct" recursive case was embeddable within it), and thirdly there was no embeddability relation with the correct recursive case. We had hoped that these three errors would map to three different styles of failure observed in our proofs. However experimentation with a number of artificially created examples revealed that this was not the case. Furthermore experimentation with our actual corpus revealed that finer distinctions could be drawn (eg. indicating that a particular sub-expression in the recursive case was to blame). This made us realise that an approach based on a broad categorisation of types of error and then an attempt to sort programs into categories was misguided. Instead, where an error is attributed to the recursive case of a program, the trace of the proof attempt can be used to further localise the error to a sub-expression.

2.3 Specification Errors

Additionally some errors in the corpus could be broadly classified into programs which simply omitted to satisfy some part of the specification. In this case it is unclear that proof had much to offer over and above any sort of counter-example generation. A counter-example could quickly show that this part of the specification was not met and, usually, a major redesign of the program was required in order to accommodate the omitted feature so identifying a culpable program fragment was not really an issue.

3 Analysis of Some Specific Examples

In this paper we focus on three ML list processing exercises:

1. *Write a function* `removeAll x l` *which removes all occurrences of the item* `x` *in the list* `l`.
2. *Write a function* `onceOnly l` *which returns a list containing exactly one copy of every item that appears in* `l`.
3. *Write a function* `insertEverywhere x l` *which returns the list of all lists obtained from* `l` *by inserting* `x` *somewhere inside.*

We created several auxiliary functions in Isabelle for expressing the specifications of these exercises. In particular we used *count_list* and *sub_list* shown below. *count_list* counts the number of appearances of an element in a list and *sub_list* only evaluates to true if one list is contained in another and if the elements of that list appear in the same order as the super-list. Our Isabelle definitions of our specification functions are[5]:

[5] We have preserved much Isabelle syntax in this presentation. For instance the uncurried format; the use of # for list concatenation and the use of *Suc* to indicate the successor function on natural numbers. However in some cases with have used standard mathematical notation instead. Most notably ∈ for list membership (the Isabelle function *mem*) and ≠ for inequality.

$$\begin{aligned}
\mathit{count\_list}\, a\, [] &= 0, && (1)\\
\mathit{count\_list}\, a\, (h\#t) &= \textbf{if}\ \ a = h\, \textbf{then}\ \ \mathit{Suc}(\mathit{count\_list}\, a\, t) && (2)\\
&\qquad\qquad\quad\ \textbf{else}\ \ \mathit{count\_list}\, a\, t.
\end{aligned}$$

$$\begin{aligned}
\mathit{sub\_list}\, l\, [] &= \textbf{if}\ \ l = []\ \ \textbf{then}\ \ True\ \ \textbf{else}\ \ False, && (3)\\
\mathit{sub\_list}\, l\, h\#t &= \textbf{if}\ \ l = []\ \ \textbf{then}\ \ True\ \ \textbf{else}\\
&\quad\ \textbf{if}\ \ hd\, l = h\, \textbf{then}\ \ \mathit{sub\_list}\, (tl\, l)\, t && (4)\\
&\qquad\qquad\quad\ \textbf{else}\ \ \mathit{sub\_list}\, l\, t.
\end{aligned}$$

The full specification we used for each exercise can be found in Appendix A.

Following proof planning literature we will refer, in what follows, to the use of the induction hypothesis in an inductive proof as *fertilisation.*

3.1 Case 1: Fertilisation Fails

This example occurs in the `removeAll` exercise. It is important to note here that in a previous exercise the student had been asked to create a function `removeOne` which removed one occurrence of the item `x` from the list `l`. The student has defined `removeAll` as follows:

```
fun removeAll _ [] = []
 |  removeAll x (h::t) = if x = h then removeAll x t
                                  else h::removeOne x t;
```

Presumably the student is programming by analogy, starting with their `removeOne` function. They have forgotten to change the recursive step to `removeAll`.

This program generates rather obscure counter-examples. All the following calls succeed:

```
> removeAll 1 [];
val it = [] : int list
> removeAll 1 [1, 1, 1];
val it = [] : int list
> removeAll 1 [1, 2, 1];
val it = [2] : int list
```

Our first counter-example was:

```
> removeAll 1 [1, 1, 2, 3, 4, 1, 1];
val it = [2, 3, 4, 1] : int list
```

careful investigation with additional counter-examples reveals that the problem arises when the item to be removed occurs more than once in the list *after* an element that is not to be removed.

We translated the student code into Isabelle as:

$$\begin{aligned} removeAll\, x\, [] &= [], & (5)\\ removeAll\, x\, (h\#t) &= \textbf{if}\ \ x = h\ \textbf{then}\ \ removeAll\, x\, t & (6)\\ &\qquad\qquad\qquad\ \ \textbf{else}\ \ h\#removeOne\, x\, t. \end{aligned}$$

The attempted proof against the first part of the specification,

$$\neg(x \in removeAll(x, l)),$$

gets blocked at an unsuccessful fertilisation attempt,

$$\neg x \in removeAll\, x\, l\ \wedge x \neq a \Rightarrow \neg x \in removeOne\, x\, l.$$

It is easy to see from this goal that the "blame" lies with the use of the function `removeOne` and it is also possible to work out the correction that is needed to make the proof complete successfully. Indeed an automated technique such as difference unification [1] would probably be able to generate a patch.

In this case the proof trace much more directly localises the error in the program than the counter-example did.

3.2 Case 2: Fertilisation succeeds

This is a case where the student has been asked to write the `insertEverywhere` function. The attempt is

```
fun insertEverywhere x [] = [[x]]
  | insertEverywhere x (x1 :: xs) = (x :: x1 :: xs)
                                        :: insertEverywhere x (xs);
```

and they have, in fact, added the comment

> "this function only returns the list of lists given by inserting the value x before all the elements in the list. It does not take into account the values before which x has already been inserted. For example 1[2,3]; would return[[1,2,3],[1,3],[1]] not sure how to implement the function to include the previous values in the list"

indicating that they are well aware of the bug in their program. The problem can be solved by mapping `\l. x1::l` over all the lists produced by `insertEverywhere`[6].

As part of verifying this we attempted to prove

$$l \in insertEverywhere\, x\, l_1 \rightarrow count\_list\, x\, l = Suc(count\_list\, x\, l_1).$$

[6] Although the correct student programs generally used a sub-function with an accumulator argument to achieve the same effect.

The step case is

$$\begin{aligned} &\forall x_a . x_a \in \mathit{insertEverywhere}\; x\; \mathit{list} \rightarrow \mathit{count\_list}\; x\; x_a = \mathit{Suc}(\mathit{count\_list}\; x\; \mathit{list}) \\ &\quad \Rightarrow \forall x_a . x_a \in \mathit{insertEverywhere}\; x\; (a\#\mathit{list}) \rightarrow \\ &\quad \mathit{count\_list}\; x\; x_a = \mathit{Suc}(\mathit{count\_list}\; x\; (a\#\mathit{list})). \end{aligned}$$

This simplifies to

$$\begin{aligned} &\forall x_a . x_a \in \mathit{insertEverywhere}\; x\; \mathit{list} \rightarrow \mathit{count\_list}\; x\; x_a = \mathit{Suc}(\mathit{count\_list}\; x\; \mathit{list}) \\ &\Rightarrow x = a \rightarrow (\forall x.(a\#a\#\mathit{list} = x \rightarrow \mathit{count\_list}\; a\; x = \mathit{Suc}(\mathit{Suc}(\mathit{count\_list}\; a\; \mathit{list}))) \wedge \\ &\qquad (a\#a\#\mathit{list} \neq x \rightarrow \\ &x \in \mathit{insertEverywhere}\; a\; \mathit{list} \rightarrow \mathit{count\_list}\; a\; x = \mathit{Suc}(\mathit{Suc}(\mathit{count\_list}\; a\; \mathit{list})))). \end{aligned}$$

NB. This shows that Isabelle has automatically proved a branch where $x \neq a$. Repeated use of introduction rules then case splits this into two goals depending on whether $x_a = a :: a :: \mathit{list}$ or whether $x_a \in \mathit{insertEverywhere}\; a\; \mathit{list}$. These two branches can be mapped to sub-expressions of the original recursive case of the function definition: `(x :: x1 :: xs)` and `insertEverywhere x (xs)` respectively. The first of these goals is true and we can derive false from the second with the following sequence of steps:

$$\begin{aligned} &\forall x_a . x_a \in \mathit{insertEverywhere}\; a\; \mathit{list} \rightarrow \mathit{count\_list}\; a\; x_a = \mathit{Suc}(\mathit{count\_list}\; a\; \mathit{list}) \\ &\quad \wedge x = a \wedge a\#a\#\mathit{list} \neq x_a \wedge x_a \in \mathit{insertEverywhere}\; a\; \mathit{list} \\ &\quad \Rightarrow \mathit{count\_list}\; a\; x_a = \mathit{Suc}(\mathit{Suc}(\mathit{count\_list}\; a\; \mathit{list})). \end{aligned}$$

Fertilisation occurs

$$\begin{aligned} &\mathit{count\_list}\; a\; x_a = \mathit{Suc}(\mathit{count\_list}\; a\; \mathit{list}) \wedge \\ &x = a \wedge a\#a\#\mathit{list} \neq x_a \wedge x_a \in \mathit{insertEverywhere}\; a\; \mathit{list} \\ &\Rightarrow \mathit{count\_list}\; a\; x_a = \mathit{Suc}(\mathit{Suc}(\mathit{count\_list}\; a\; \mathit{list})). \end{aligned}$$

and we now have a contradiction between the first hypothesis and the conclusion.

In order to prevent this contradiction arising it is necessary to prevent the unification of $x_a \in \mathit{insertEverywhere}\; a\; \mathit{list}$ with the antecedent of $x_a \in \mathit{insertEverywhere}\; a\; \mathit{list} \rightarrow \mathit{count\_list}\; a\; x_a = \mathit{Suc}(\mathit{count\_list}\; a\; \mathit{list})$. The goal, $x_a \in \mathit{insertEverywhere}\; a\; \mathit{list}$, was obtained by rewriting the formula $x_a \in \mathit{insertEverywhere}\; a\; (a\#\mathit{list})$ with the definition of *insertEverywhere* and then taking the tail of the resulting term. Clearly this tail needed some extra structure somewhere to prevent immediate fertilisation. In this case the extra structure was needed around *insertEverywhere a list* but conceivably it could have required extra structure in one of the argument positions.

It is not clear that we know anything more at this point than we did from the counter-examples detailed by the student (and the ones we generated ourselves in testing) since these clearly indicate that successive calls to `insertEverywhere` are losing necessary information. It is, however, possible to see how an automated mechanism could isolate the responsible sub-expression using a proof trace more easily, perhaps, than it could from analysis of a counter-example alone.

It is also possible that some form of deductive synthesis [3] or corrective predicate construction [11] at this point might allow the correct sub-expression to be synthesized.

Case 3: Fertilisation not expected

In this case another student, also attempting to write `insertEverywhere`, has met problems. They have defined a subsidiary function, `ie`:

```
fun ie n R [] = [R@[n]]
 | ie n R (h::t) = (n::h::t)::[R]@(ie n (R@[h]) t);
```

Again they are aware that the function does not work correctly *"I have had massive problems trying to concatenate the list R and the tail"*. There are several problems here. `R` needs to appear within the head of the list of lists; and it should appear *before* and not after `(n::h::t)`.

This program produces an odd set of counter-examples:

```
> insertEverywhere 0 [1, 2];
val it = [[0, 1, 2], [], [0, 2], [1], [1, 2, 0]] : int list list
> insertEverywhere 0 [1, 2, 3];
val it = [[0, 1, 2, 3], [], [0, 2, 3], [1], [0, 3], [1, 2],
          [1, 2, 3, 0]] : int list list
```

However `insertEverywhere []` produces the correct answer which together with the counter-examples strongly suggests that the problem lies with the recursive rather than the basis case of the functions. So the question is whether proof can isolate the problem further.

In our Isabelle development it was easy to establish that

$$\mathit{insertEverywhere}\, x\, l = \mathit{ie}\, n\, []\, l$$

We then attempted to establish a generalised version of our specification including

$$l \in (\mathit{ie}\, x\, l_1\, l_2) \Rightarrow \mathit{count\_list}\, x\, l = \mathit{Suc}(\mathit{count\_list}\, x\, l_2) + \mathit{count\_list}\, x\, l_1$$

The proof follows a similar pattern to that in the previous example. We perform induction on l_2 considering $a\#list$ in the step case and then case split on whether $l = x\#a\#list$. In this instance the first branch of the case split causes problems resulting in the goal

$$\begin{gathered}\forall x_a x_b . x_b \in \mathit{ie}\, x\, x_a\, \mathit{list} \rightarrow \\ \mathit{count\_list}\, x\, x_b = \mathit{Suc}(\mathit{count\_list}\, x\, \mathit{list}) + \mathit{count\_list}\, x\, x_a \\ \wedge\, l = x\#a\#\mathit{list} \Rightarrow \\ \mathit{count\_list}\, x\, (x\#a\#\mathit{list}) = \mathit{Suc}(\mathit{count\_list}\, x\, (a\#\mathit{list})) + \mathit{count\_list}\, x\, l_1,\end{gathered}$$

which rewrites to

$$\begin{gathered}\forall x_a x_b . x_b \in \mathit{ie}\, x\, x_a\, \mathit{list} \rightarrow \\ \mathit{count\_list}\, x\, x_b = \mathit{Suc}(\mathit{count\_list}\, x\, \mathit{list}) + \mathit{count\_list}\, x\, x_a \\ \wedge\, l = x\#a\#\mathit{list} \Rightarrow \\ \mathit{Suc}(\mathit{count\_list}\, x\, (a\#\mathit{list})) = \mathit{Suc}(\mathit{count\_list}\, x\, (a\#\mathit{list})) + \mathit{count\_list}\, x\, l_1,\end{gathered}$$

which simplifies to

$$\begin{gathered}\forall x_a x_b . x_b \in \mathit{ie}\; x\; x_a\; \mathit{list} \rightarrow \\ \mathit{count\_list}\; x\; x_b = \mathit{Suc}(\mathit{count\_list}\; x\; \mathit{list}) + \mathit{count\_list}\; x\; x_a \\ \wedge\; l = x\#a\#\mathit{list} \;\Rightarrow\; \mathit{count\_list}\; x\; l_1 = 0\end{gathered}$$

which is satisfiable but not always true.

In this case we wouldn't have expected fertilisation to be possible because of the structure of the case splits. However this has still shown there is a problem with `(n::h::t)`. In fact exploration of the remaining branches of the proof further suggests that `[R]` is also a problem while `(ie n (R@[h]) t)` is not. In this case the proof trace once again appears to have provided more information than the counter-example. For instance the fact that the actual recursive call itself *is* correctly formed is not at all obvious from the counter-examples.

3.3 Case 4: Combined Problems

We conclude by examining an example where there are a combination of errors in the program. In this case in the basis cases. This is an example where the student has been asked to write the `onceOnly` function. They have created a complex set of sub-functions, not required by the program specification, in order to achieve this:

```
fun insert x [] = []
  | insert x(h::t) =
          if x <= h then x :: h :: t
                    else h :: insert x t;

fun sort [] = []
  | sort (x :: xs) = insert x (sort xs);


fun Once [] = []
  | Once (x1 :: x2 :: xs) =
        if x1 = x2 then Once (x2 :: xs)
                   else x1 :: x2 :: Once xs;

fun onceOnly [] = []
  | onceOnly (x :: xs) = Once (sort (x :: xs));
```

There are two errors here. Firstly the basis case for `insert` should be `insert x [] = [x]` and secondly we are missing a basis case for `Once` where there is a singleton list. Our counter-examples indicate that all calls to `onceOnly` evaluate to `[]`.

It is fairly simple to prove that

$$\mathit{onceOnly}\; l = \mathit{Once}(\mathit{sort}\; l).$$

We then needed to establish a number of theorems about `sort` and `insert`. We introduced two new functions *min_list* and *less_min_list* (which evaluate tot he minimum element in a list, and a list less its minimum element respectively):

$$min\_list(a\#[]) = a, \tag{7}$$
$$min\_list(h\#t) = min\, h\; (min\_list\, t). \tag{8}$$

$$less\_min\_list\; [] = [], \tag{9}$$
$$less\_min\_list\; (h\#t) = \textbf{if}\; h = min\_list(h\#t)\;\; \begin{array}{l}\textbf{then}\; t \\ \textbf{else}\; h\#(less\_min\_list\, t).\end{array} \tag{10}$$

and attempted to prove

$$l \neq [] \Rightarrow (sort\, l) = (min\_list\, l)\#sort(less\_min\_list\, l).$$

The step case of this proof introduces the goal:

$$\begin{array}{c} list \neq [] \Rightarrow sort\ list = (min\_list\ list)\#(sort\ (less\_min\_list\ list)) \\ a\#list \neq [] \Rightarrow sort\ a\#list = (min\_list\ a\#list)\#(sort\ (less\_min\_list\ a\#list)) \end{array}$$

which triggers a case split on whether $list = []$. In the case where this is true the induction hypothesis evaluates to true and rewriting the induction conclusion reaches the goal

$$insert\ a\ [] = [a]$$

which rewrites to

$$[] = [a]$$

and then we derive *False*. From this it is obvious that the appropriate fix is to edit the basis case of `insert` to `insert x [] = [x]`. It is important to note here that the lemma we have chosen is dependent our intuitions about the way `insert` and `sort` should behave. If the student had named these functions less informatively the process of locating the `insert` error would have been considerably complicated.

It isn't possible to detect the additional problem with `Once` even continuing from this point with enthusiastic use of Isabelle's `sorry` command in order to establish theorems that can't be proved. `Once` is only ever applied to expressions of the form `sort l` which, because of the bug in `insert`, all evaluate to `[]` so the missing case in `Once` is undetectable.

In other examples we were able to identify combinations of problems where they caused the proof to break down in different branches of the trace. However this example illustrates some limitations of the proof approach in terms of detecting all the errors within a program.

4 Discussion and Related Work

We have picked some representative examples of problems in our corpus of study. Unsurprisingly these illustrate that the structure of a program correctness proof is related to the structure of the underlying program. It is important to realise that the structure of the proof is also affected by the structure of the specification functions and so a direct mapping between branches of a proof and cases within a program is not always possible. Also of note is the fact that there may not be an explicit `if ... then ... else` structure in a program and yet the structure of the code can still induce a case split on the proof allowing us to focus our attention on particular sub-expressions. In our examples by the interaction of Isabelle's list membership function and the head::tail structure of lists.

It is also possible to see that the proof traces provide hints about how a proof can be patched sometimes directly providing the correct evaluation of an expression (the `insert` example) and sometimes highlighting where information may be missing, etc.

Our examples raise a number of interesting questions:

1. How could the proof traces shown here be produced automatically. In particular how should such a system decide when it has reached an informative failure? and how can a programmer's intended behaviour for sub-functions be determined?
2. Given a trace how can useful information be extracted and presented to a user?
3. How might patches for problem areas of code be constructed?

Automating Proof Tracing Although in many cases the programs we are studying are only a few lines long the proofs we have produced raise a number of challenges for automation, even when the programs are correct. For instance we frequently needed to generalise our goals to accommodate the presence of accumulators in sub-functions and occasionally needed to speculate new functions and lemmas entirely (e.g. the need to provide a rule for expressing *sort l* in terms of the head and tail of a list). Proof planning already has an account of how new lemmas and generalisations can be found [8] but our examples present considerable challenges to the state-of-the-art in this area.

Leaving aside the issue of appropriate lemma speculation we also found that in order to extract the useful information from the failed proofs we had to bypass Isabelle's simplifier to step through a number of rewrite steps and other simplifications by hand before we reached a goal that was "informative". In general we used the simplifier to narrow the investigation to particular branch of the proof, but then found we needed to retract the simplification to gain finer grained information about exactly which case was causing problems and the steps that led to an unprovable goal. This process would need to be controlled carefully in any automated system. As a related issue an important part of this process was identifying goals that were satisfiable but not always true, existing counter-example discovery technology clearly has a role to play here. Possibly a call to `quickcheck` or similar should be employed each time a proof attempt branches in order to ascertain whether a system should attempt to prove that branch or gather error information.

Lastly there is the issue of programmer introduced sub-functions. In an ideal world a programmer would specify the behaviour of all sub-functions as well as the main program however there are many situations where this will not be the case, for instance if an error diagnosis system were used as a marking aid rather than as a program construction aid for students. In some cases it may be possible to use ontology matching and repair methods [10] to deduce the intended behaviour. In case 4 the fact that the function was named `insert` would allow us to compare its behaviour to a correct list insertion function[7]. It might also be possible, in some cases, to get a programmer to provide sample inputs and outputs and use conjecture forming technology [4] to deduce appropriate lemmas.

Extracting Information from Proof Traces Once a proof trace has been produced there is then the question of how useful information can be extracted from it. There would appear to be a number of ways in which this could work, some general and some related to specific forms of failure. For instance, when fertilisation has failed it seems plausible to attempt difference unification [1] of the induction conclusion and the induction hypothesis in order to highlight differences and suggest patches. Similarly where we are attempting to prove an equality, it may be possible to compare the LHS and RHS in order to adapt a function's output.

It is also often possible to extract generalised counter-examples or counter-example classes from a failed proof branch which may provide more focused information than individual counter-examples. For instance, in the `insert` example, it is possible to deduce there is a problem with all one element lists.

Patching Problem Code Monroy [11] has already used proof planning to examine faulty conjectures. He follows work by Franova and Kodratoff [7] and Protzen [16] and attempts to synthesize a *corrective predicate* in the course of proof. The idea is that the corrective predicate will represent the theorem that the user intended to prove. This predicate is represented by a meta-variable, P, such that $P \rightarrow G$ where G is the original (non)theorem. This process can correctly fill in missing base cases[8] but the approach would need modification if it were to remove a piece of faulty code and then replace it with a different correct fragment.

An alternative approach might be to look to deductive synthesis technology [3]. Deductive synthesis uses meta-variables in existence proofs to synthesise a program that meets its specification. Once an offending program fragment has been identified it should be possible to replace it with a meta-variable and attempt to use similar techniques to instantiate this variable. Similarly where premature fertilisation has occurred indicating missing structure, the possible lo-

[7] Our thanks to Fiona McNeill for this suggestion.

[8] Monroy and Dennis, *Fault Diagnosis*, Edinburgh Dream Group Blue Book Note 1485.

cations for this structure could be represented by meta-variables and deductive synthesis used to instantiate these.

5 Conclusion and Further Work

In this paper we have reported the results of of a case study in the use of proof to locate program errors. We have shown that the structure of the proof can be used to narrow the focus of attention to specific parts of program and made some suggestions about how such a trace could, in some situations, also be used to suggest appropriate patches. We have compared the information we could extract from our failed proof attempts with the information deducible from counter-examples and concluded that it is generally, although not always, possible to narrow the focus to the culpable piece of code better using a proof trace than it is using the counter-example.

We now intend to construct an automated system based on proof planning to produce these proof traces and then use this system to automatically generate diagnoses and patches.

Acknowledgements

This research was funded by EPSRC grant GR/S01771/01 and Nottingham NLF grant 3051.

References

1. D. A. Basin and T. Walsh. Difference unification. In R. Bajcsy, editor, *Proceedings of IJCAI-93*, pages 116–122. Morgan Kaufmann, 1993.
2. A. Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.
3. A. Bundy, L. Dixon, J. Gow, and J. D. Fleuriot. Constructing induction rules for deductive synthesis proofs. In *Constructive Logic for Automated Software Engineering*, ENTCS. Elsevier, 2005. To Appear.
4. S. Colton. The HR program for theorem generation. In A. Voronkov, editor, *18th International Conference on Automated Deduction*, volume 2392 of *LNCS*, pages 285–289. Springer, 2002.
5. L. A. Dennis. The use of proof planning critics to diagnose errors in the base cases of recursive programs. In W. Ahrendt, P. Baumgartner, and H. de Nivelle, editors, *IJCAR 2004 Workshop on Disproving: Non-Theorems, Non-Validity, Non-Provability*, pages 47–58, 2004.
6. L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In F. Baader, editor, *19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2003.
7. M. Franova and Y. Kodratoff. Predicate synthesis from formal specification. In B. Neumann, editor, *10th European Conference on Artificial Intelligence*, pages 97–91. John Wiley and Sons, 1992.

8. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.
9. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publisher, 1993.
10. F. McNeill, A. Bundy, and C. Walton. Diagnosing and repairing ontological mismatches. In *Starting AI Researchers' Symposium*, 2004. Also available as Edinburgh Informatics Report EDI-INF-RR-0251.
11. R. Monroy. Predicate synthesis for correcting faulty conjectures: The proof planning paradigm. *Automated Software Engineering*, 10(3):247–269, 2003.
12. F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6:139–143, 1984.
13. P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.
14. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
15. L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer, 1994.
16. M. Protzen. Patching faulty conjectures. In M. A. McRobbie and J. K. Slaney, editors, *13th Conference on Automated Deduction*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 77–91. Springer, 1996.
17. G. Steel, A. Bundy, and E. Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. *AISB Journal*, 1(2):169–182, 2002.

Appendix A: Isabelle Specifications Used

1.1 `removeAll`

removeAll_spec1: $\neg(x \in \mathit{removeAll}(x, l))$
removeAll_spec2: $x \neq a \Rightarrow \mathit{count\_list}(x, \mathit{removeAll}(a, l)) = \mathit{count\_list}(x, l)$
removeAll_spec3: $\neg(x \in l) \Rightarrow \mathit{removeAll}(x, l) = l$

1.2 `onceOnly`

onceOnly_spec1: $\neg(x \in l) \Rightarrow \mathit{count\_list}\ x\ (\mathit{onceOnly}\ l) = 0$
onceOnly_spec2: $(x \in l) \Rightarrow \mathit{count\_list}\ x\ (\mathit{onceOnly}\ l) = 1$

1.3 `insertEverywhere`

insertEverywhere_spec1: $l_1 \in \mathit{insertEverywhere}\ x\ l \Rightarrow \mathit{count\_list}\ x\ l_1 = \mathit{Suc}(\mathit{count\_list}\ x\ l)$
insertEverywhere_spec2: $l_1 \in \mathit{insertEverywhere}\ x\ l \Rightarrow \mathit{sub\_list}\ l\ l_1$
insertEverywhere_spec3: $l_1 \in \mathit{insertEverywhere}\ x\ l \wedge x_1 \neq x \Rightarrow \mathit{count\_list}\ x_1\ l_1 = \mathit{count\_list}\ x_1\ l$
insertEverywhere_spec4: $(\mathit{count\_list}\ x\ l_1 = \mathit{Suc}(\mathit{count\_list}\ x\ l) \wedge \mathit{sub\_list}\ l\ l_1 \wedge \forall x_1.\ x_1 \neq x \rightarrow \mathit{count\_list}\ x_1\ l_1 = \mathit{count\_list}\ x_1\ l) \Rightarrow l_1 \in (\mathit{insertEverywhere}\ x\ l)$

LAST REVISED June 2, 2005

A Proof-Producing Hardware Compiler for a Subset of Higher Order Logic

Mike Gordon, Juliano Iyoda
University of Cambridge
Computer Laboratory
William Gates Building
JJ Thomson Avenue
Cambridge CB3 0FD, UK

Scott Owens, Konrad Slind
University of Utah
School of Computing
50 South Central Campus Drive
Salt Lake City
Utah UT84112, USA

(*authors listed in alphabetical order*)

Abstract. Higher order logic (HOL) is a modelling language suitable for specifying behaviour at many levels of abstraction. We describe a compiler from a 'synthesisable subset' of HOL function definitions to correct-by-construction clocked synchronous hardware. The compiler works by theorem proving in the HOL4 system and goes through several phases, each deductively refining the specification to a more concrete form, until a representation that corresponds to hardware is deduced. It also produces a proof that the generated hardware implements the HOL functions constituting the specification. Synthesised designs can be translated to Verilog HDL, simulated and then input to standard design automation tools. Users can modify the theorem proving scripts that perform compilation. A simple example is adding rewrites for peephole optimisation, but all the theorem-proving infrastructure in HOL4 is available for tuning the compilation. Users can also extend the synthesisable subset. For example, the core system can only compile tail-recursions, but a 'third-party' tool `linRec` is being developed to automatically generate tail recursive definitions to implement linear recursions, thereby extending the synthesisable subset of HOL to include linear recursion.

1 Introduction

In the HOL4 proof system for higher order logic, a function f satisfying an equation $f(x) = e$, which may be recursive, is defined by executing:

`Define` '$f(x) = e$'

We describe an extension to `Define`, called `cirDefine`, that generates hardware implementations for a 'synthesisable subset' of higher order logic. Executing:

`cirDefine` '$f(x) = e$'

first defines f (i.e. invokes `Define`) and then automatically generates an implementation of f as a circuit suitable for executing in hardware.

The synthesised circuit is generated by proof and is thus correct by construction. A correctness theorem is proved by `cirDefine` for each circuit generated.

Our system is implemented in HOL4, but the ideas could be realised in other programmable proof systems.

In the next section we walk through a pedagogical example to illustrate the flow from higher order logic to Verilog HDL. We then outline how the proof-producing compiler works. It is a specialised theorem prover and we describe how deductions are used to generate hardware, and the sense in which synthesised circuits implement higher order logic functions. Next we discuss related work. Finally we outline future plans. An appendix contains technical details, including the formal description of the constructors used to build circuits and the four-phase handshake protocol that they implement.

2 Compiling higher order logic definitions to circuits

Let's dive in: `cirDefine` is an extension of HOL4's standard command `Define` for defining functions. Before using `cirDefine` one needs to load appropriate modules and start a new theory in which to store definitions and theorems. Assume this is done, and also assume that the `word32` library [6] is loaded, so that arithmetic operations like `+` and `-` default to 32-bit versions. Numerals like `0w` and `1w` denote the appropriate 32-bit word values, and `w2n:word32->num` converts a 32-bit word to a natural number. The HOL4 top level is Standard ML (SML); consider the following declaration:

```
 - val (Mult32Iter_def,Mult32Iter_ind,Mult32Iter_cir) =
     cirDefine
      `(Mult32Iter(m,n,acc) = if m = 0w then (0w,n,acc) else Mult32Iter(m-1w,n,n+acc))
       measuring (w2n o FST)`;
```

The right hand side of the declaration applies the SML function `cirDefine` to an argument of the form ‘*equation* `measuring` *measure-function*‘. The term *equation* is a recursive definition of a function `Mult32Iter` that takes a triple of 32-words and returns another triple of the same type. The measure function[1] `w2n o FST` maps a triple of 32-bit words to the natural number denoted by the first member of the triple (it is used to show termination).

The result of `cirDefine` is a triple of theorems in higher order logic. The first component of the triple, which is bound to `Mult32Iter_def` in the declaration above, is the theorem resulting from applying the HOL4 function definition tool (`Define`), using the measure function to aid proof of termination. A side effect of this definition is to define `Mult32Iter` as a constant and prove the appropriate 'definitional' theorem, which is the theorem returned. Thus the first output from the input shown above would be:

```
 > val Mult32Iter_def =
     |- Mult32Iter(m,n,acc) = (if m = 0w then (0w,n,acc) else Mult32Iter(m-1w,n,n+acc))
```

The second output is a theorem that is bound to `Mult32Iter_ind`. This is a custom induction principle for the constant `Mult32Iter` that can be used for proofs about it. We show this for completeness, but we do not discuss any proofs.

```
  val Mult32Iter_ind =
    |- ∀P. (∀m n acc. (¬(m = 0w) ==> P(m-1w,n,n+acc)) ==> P(m,n,acc)) ==> ∀v v1 v2. P(v,v1,v2)
```

[1] Measure functions are not always necessary as they can be inferred using heuristics by the termination prover. We expect future releases of the compiler to figure out the measure function automatically for simple recursions like the one here.

The final component of the triple of theorems returned by `cirDefine` is the result of compiling the definition of `Mult32Iter` to a circuit. We show this now, and then follow it by some explanation.

```
val Mult32Iter_cir =
  |- InfRise clk
     ==>
     (∃ v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18
         v19 v20 v21 v22 v23 v24 v25 v26 v27 v28 v29 v30 v31 v32 v33 v34
         v35 v36 v37 v38 v39 v40 v41 v42 v43 v44 v45 v46 v47 v48 v49 v50
         v51 v52 v53 v54 v55 v56 v57.
        DtypeT (clk,load,v21) ∧ NOT (v21,v20) ∧ AND (v20,load,v19) ∧
        Dtype (clk,done,v18) ∧ AND (v19,v18,v17) ∧ OR (v17,v16,v11) ∧
        DtypeT (clk,v15,v23) ∧ NOT (v23,v22) ∧ AND (v22,v15,v16) ∧
        MUX (v16,v14,inp1,v3) ∧ MUX (v16,v13,inp2,v2) ∧
        MUX (v16,v12,inp3,v1) ∧ DtypeT (clk,v11,v26) ∧ NOT (v26,v25) ∧
        AND (v25,v11,v24) ∧ MUX (v24,v3,v27,v10) ∧
        Dtype (clk,v10,v27) ∧ DtypeT (clk,v11,v30) ∧ NOT (v30,v29) ∧
        AND (v29,v11,v28) ∧ MUX (v28,v2,v31,v9) ∧ Dtype (clk,v9,v31) ∧
        DtypeT (clk,v11,v34) ∧ NOT (v34,v33) ∧ AND (v33,v11,v32) ∧
        MUX (v32,v1,v35,v8) ∧ Dtype (clk,v8,v35) ∧
        DtypeT (clk,v11,v39) ∧ NOT (v39,v38) ∧ AND (v38,v11,v37) ∧
        NOT (v37,v7) ∧ CONSTANT 0w v40 ∧ EQ32 (v3,v40,v36) ∧
        Dtype (clk,v36,v6) ∧ DtypeT (clk,v7,v44) ∧ NOT (v44,v43) ∧
        AND (v43,v7,v42) ∧ AND (v42,v6,v5) ∧ NOT (v6,v41) ∧
        AND (v41,v42,v4) ∧ DtypeT (clk,v5,v48) ∧ NOT (v48,v47) ∧
        AND (v47,v5,v46) ∧ NOT (v46,v0) ∧ CONSTANT 0w v45 ∧
        Dtype (clk,v45,out1) ∧ Dtype (clk,v9,out2) ∧
        Dtype (clk,v8,out3) ∧ DtypeT (clk,v4,v53) ∧ NOT (v53,v52) ∧
        AND (v52,v4,v51) ∧ NOT (v51,v15) ∧ CONSTANT 1w v54 ∧
        SUB32 (v10,v54,v50) ∧ ADD32 (v9,v8,v49) ∧ Dtype (clk,v50,v14) ∧
        Dtype (clk,v9,v13) ∧ Dtype (clk,v49,v12) ∧
        Dtype (clk,v15,v56) ∧ AND (v15,v56,v55) ∧ AND (v0,v7,v57) ∧
        AND (v57,v55,done))
     ==>
     DEV Mult32Iter
         (load at clk, (inp1<>inp2<>inp3) at clk, done at clk, (out1<>out2<>out3) at clk)
```

This theorem, which is bound to the ML name `Mult32Iter_cir`, has the form:

`|- InfRise clk ==>` *circuit* `==>` *device specification*

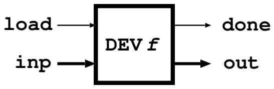
The variable `clk` represents the clock and is modelled by a function from time (natural numbers) to clock values (Booleans). The term `InfRise clk` asserts that `clock` has an infinite number of rising edges. This is a standard precondition for temporal abstraction [10] and is needed because of the use of the `at`-operator (explained below) in the device specification.

The *circuit* is a standard representation as a conjunction of component instances with internal lines existentially quantified (ibid). The components used here are described in Section 2.1. Circuits in this form are the lowest level of formal representation we generate. However they are easily converted to HDL and then simulated or input to other tools. We have written a ‘pretty-printer’ that generates Verilog HDL and have used several simulators and the Quartus II FPGA synthesis tool to run examples (including `Mult32Iter`) on FPGAs.

The *device specification* uses a HOL predicate `DEV` that specifies how a HOL function f is computed using a four-phase handshake. The general form is:

`DEV` f `(load at clk, inp at clk, done at clk, out at clk)`

where a term of the form σ `at clk` denotes the signal consisting of the sequence of values of σ at successive rising edges of `clk`. This is a standard operation of temporal abstraction (sometimes called clock projection and denoted by σ`@clk`). Temporal abstraction (projection) converts a signal σ representing the behavior of a wire (or bundle of wires) at the clocked circuit level to a signal σ `at clk` representing the behavior at the 'cycle level' – i.e. to an abstracted signal in which successive values represent the values during successive stable states of the circuit. The predicate `DEV` relates the values of signals at the abstracted level. A term `DEV` f `(load,inp,done,out)` specifies a handshaking device computing f where the signals `load`, `inp`, `done` and `out` are the handshake request line, the data input bus, the handshake acknowledge line and data output bus, respectively.

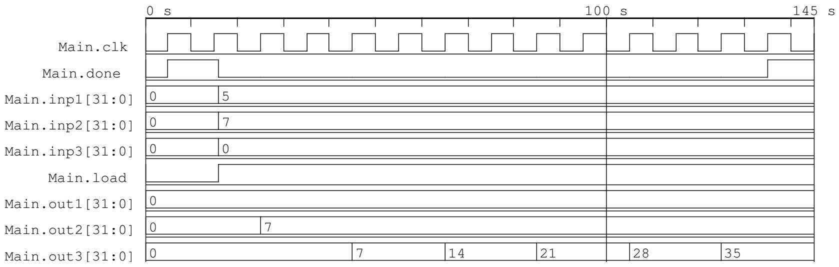


The behavior of such a handshaking device is formalised in the HOL definition of `DEV`, which says that if a value v is input on `inp` when a request is made on `load` then eventually $f(v)$ will be output on `out`, and when this occurs `T` is signalled on `done`. A formal specification of the handshake protocol is given in the Appendix. At the start of a transaction (say at time t) the device must be outputting `T` on `done` (to indicate it is ready) and the environment must be asserting `F` on `load`, i.e. in a state such that a positive edge on `load` can be generated. A transaction is initiated by asserting (at time t+1) the value `T` on `load`, i.e. `load` has a positive edge at time t+1. This causes the device to read the value, v say, being input on `inp` (at time t+1) and to set `done` to `F`. The device then becomes insensitive to inputs until `T` is next asserted on `done`, at which time (say time $t' > t$+1) the value $f(v)$ computed will be output on `out`. In the implementation generated by our compiler, `load`, `inp`, `done` and `out` are only sampled on rising edges of a clock `clk`, hence the behavior is specified by:

`DEV` f `(load at clk, inp at clk, done at clk, out at clk)`.

In `Mult32Iter_cir`, the lines `inp` and `out` carry triples of 32-bit words, which are represented by `inp1<>inp2<>inp3` and `out1<>out2<>out3` where `inp1`, `inp2`, `inp3`, `out1`, `out2`, `out3` are 32-bit busses and `<>` denotes word concatenation.

If we simulate our implementation of `Mult32Iter` with inputs $(5, 7, 0)$ using the Icarus Verilog simulator (http://www.icarus.com) and view the result with the GTKWave waveform viewer (http://home.nc.rr.com/gtkwave), the result is:



`load` is asserted at time 15 and `done` is `T` then, but `done` immediately drops to `F` in response to `load` being asserted. At the same time as `load` is asserted the values 5, 7 and 0 are put on lines inp1, inp2 and inp3, respectively. At time 135 done rises to T again, and by then the values on out1, out2 and out3 are 0, 7 and 35, respectively, thus Mult32Iter(5,7,0) = (0,7,35), which is correct.

2.1 Primitive components

The compiler generates circuits using components from a predefined library, which can be changed to correspond to the targeted technology (the default target is Altera FPGAs synthesised using Quartus II).

The components used in Mult32Iter_cir are NOT, AND, OR (logic gates), EQ32 (32-bit equality test), MUX (multiplexer), DtypeT (Boolean D-type register that powers up into an initial state storing the value T), Dtype (D-type register with unspecified initial state), CONSTANT (read-only register with a predefined value: 0w or 1w are used in Mult32Iter_cir), ADD32 (32-bit adder) and 32-bit SUB32 (32-bit subtractor). Each of these components is defined in a standard style (ibid) in higher order logic. For example, NOT is defined by:

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

and the corresponding Verilog module definition that the compiler generates is

```
// Verilog module implementing HOL unary operator
// $~ :bool -> bool
//
// Automatically generated definition of NOT
module NOT (inp,out);
 parameter inpsize = 0;
 parameter outsize = 0;
 input   [inpsize:0] inp;
 output  [outsize:0] out;

 assign out = ! inp;

endmodule
```

An instance of a NOT-gate occurring in Mult32Iter_cir is NOT(v51,v15), which is ‘pretty-printed’ as a module instance with unique name NOT_12:

```
/* NOT ((v51 :num -> bool),(v15 :num -> bool)) */
NOT        NOT_12 (v51,v15);
  defparam NOT_12.inpsize = 0;
  defparam NOT_12.outsize = 0;
```

Notice that comments are automatically generated in the Verilog showing the corresponding HOL source. This is so that manual inspection can be used to check that the Verilog is correct (a formal check is impossible, as there is no formal semantics of Verilog).

NOT is typical of all the combinational components. The two sequential components, Dtype and DtypeT, are registers that are triggered on the rising edge (posedge) of a clock and their definitions use the predicate Rise defined by:

$$\mathtt{Rise}\ s\ t\ =\ \neg s(t)\ \wedge\ s(t{+}1)$$

and then Dtype and DtypeT are defined by:

$$\begin{aligned}\mathtt{Dtype}\ (clk,d,q) &= \ \forall t.\ q(t{+}1) = \mathtt{if\ Rise}\ clk\ t\ \mathtt{then}\ d\ t\ \mathtt{else}\ q\ t\\ \mathtt{DtypeT}(clk,d,q) &= \ (q\ 0 = \mathtt{T})\ \wedge\ \mathtt{Dtype}(clk,d,q)\end{aligned}$$

which are coded in Verilog as:

```
// Positive edge triggered Dtype register
// Dtype(clk,d,q) = !t. q(t+1) = if Rise clk t then d t else q t
module Dtype (clk,d,q);
 parameter size = 31;
 input clk;
 input  [size:0] d;
 output [size:0] q;
 reg    [size:0] q;

 initial q = 0;

 always @(posedge clk) q <= d;

endmodule

// Boolean positive edge triggered flip-flop starting in state 1
// DtypeT(clk,d,q) = (q 0 = T) /\ Dtype(clk,d,q)
module DtypeT (clk,d,q);
 input clk,d;
 output q;
 reg q;

 initial q = 1;

 always @(posedge clk) q <= d;

endmodule
```

The reason for `initial q = 0` in the definition of the `Dtype` module is explained in Section 7. Since our proofs are valid for any initial value of `q`, the Verilog module `Dtype` is a valid implementation of the model in higher order logic.

Terms `Dtype(clk,v50,v14)` and `DtypeT(clk,v4,v53)` in `Mult32Iter_cir` generate named instances of these modules:

```
 /* Dtype ((clk :num -> bool),(v50 :num -> word32),(v14 :num -> word32)) */
 Dtype      Dtype_8 (clk,v50,v14);
   defparam Dtype_8.size = 31;

 /* DtypeT ((clk :num -> bool),(v4 :num -> bool),(v53 :num -> bool)) */
 DtypeT     DtypeT_8 (clk,v4,v53);
```

The automatically generated comments show the HOL source, to aid checking that the Verilog is correct.

2.2 Compiled components

After compiling `Mult32Iter`, its implementation is added to the library of components, so one can use it in subsequent compilations. For example, a multiplier using `Mult32Iter` could be defined by:

```
(******************************************************************************)
(* Create an implementation of a multiplier from Mult32Iter                   *)
(******************************************************************************)
val (Mult32,_,Mult32_cir) =
 cirDefine
  `Mult32(m,n) = SND(SND(Mult32Iter(m,n,0w)))`;
```

where `SND(SND(m,n,acc))` evaluates to `acc`. The compiler finds hardware implementing `Mult32Iter` in the component library and uses that to generate an implementation of `Mult32` (abbreviated to *Mult32Circuit* below):

```
|- InfRise clk
    ==> Mult32Circuit
    ==> DEV Mult32 (load at clk, (inp1<>inp2) at clk, done at clk, out at clk)
```

Mult32 is then added to the component library and can be used in subsequent compilations, for example:

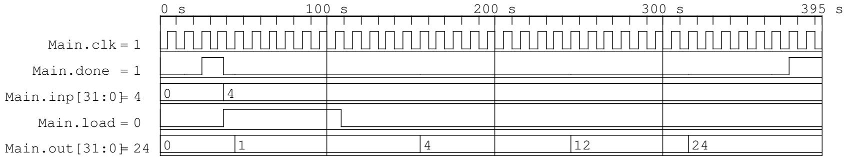
```
(*****************************************************************************)
(* Implement iterative function as a step to implementing factorial         *)
(*****************************************************************************)
val (Fact32Iter,Fact32Iter_ind,Fact32Iter_cir) =
 cirDefine
  `(Fact32Iter(n,acc) =
     if n = 0w then (n,acc) else Fact32Iter(n-1w, Mult32(n,acc)))
   measuring (w2n o FST)`;

(*****************************************************************************)
(* Implement a function Fact32 to compute SND(Fact32Iter (n,1))             *)
(*****************************************************************************)
val (Fact32,_,Fact32_cir) =
 cirDefine
  `Fact32 n = SND(Fact32Iter (n,1w))`;
```

This generates a circuit *Fact32Circuit* to compute the factorial function:

```
|- InfRise clk
    ==> Fact32Circuit
    ==> DEV Fact32 (load at clk, inp at clk, done at clk, out at clk)
```

The example waveform below shows that if 4 is input then 24 (i.e. 4!) is being output on Main.out when Main.done next goes high.



The 32-bit registers will overflow if one attempts to compute the factorial of n where $n > 12$. One can prove in HOL that:

$\vdash$ $\forall$n. (FACT n < 2^{32}) ==> (FACT n = w2n(Fact32(n2w n)))

FACT is the factorial function on natural numbers and the value of n2w n is the 32-bit word representing n **mod** 32. We have downloaded the Verilog version of *Fact32Circuit* onto an FPGA using Quartus II and verified that the factorial function is computed for n $\leq$ 12, and that the expected wrap-around values are computed for n > 12.

3 How the compiler works

The compiler implements functions f where $f : \sigma_1 \times \cdots \times \sigma_m \rightarrow \tau_1 \times \cdots \times \tau_n$ and $\sigma_1, \ldots, \sigma_m, \tau_1, \ldots, \tau_n$ are the types of values that can be carried on busses (e.g. n-bit words). The starting point of compilation is the definition of such a function f by an equation of the form: $f(x_1, \ldots, x_n) = e$, where any recursive calls of f in e must be tail-recursive. Applying cirDefine to such a definition (if necessary with a measure function to aid proof of termination) will first define f in higher order logic (using TFL [16]) and then prove a theorem:

```
|- InfRise clk
    ==> circuit_f
    ==> DEV f (load at clk, inputs at clk, done at clk, outputs at clk)
```

where *inputs* will be `inp1<>`$\cdots$`<>inpm`, *outputs* will be `out1<>`$\cdots$`<>outn` (with the type of `inp`i matching σ_i and the type of `out`j matching τ_j) and $circuit_f$ will be a HOL term representing a circuit with inputs `clk`, `load`, `inp1`, ..., `inpm` and outputs `done`, `out1`, ..., `outn`.

The first step (**Step 1**) in compiling $f(x_1, \ldots, x_n) = e$ encodes e as an applicative expression, e_c say, built from the operators `Seq` (compute in sequence), `Par` (compute in parallel), `Ite` (if-then-else) and `Rec` (recursion), defined by:

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

The encoding into an applicative expression built out of `Seq`, `Par`, `Ite` and `Rec` is performed by a proof script and results in a theorem $\vdash (\lambda(x_1, \ldots, x_n).\ e) = e_c$, and hence $\vdash f = e_c$. The algorithm used is straightforward and is not described here. As an example, the proof script deduces from:

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

the theorem:

$$\begin{array}{l}
\vdash\ \texttt{FactIter}\ = \\
\quad \texttt{Rec}\ (\texttt{Seq}\ (\texttt{Par}\ (\lambda(n, acc).\ n)\ (\lambda(n, acc).\ 0))\ (=)) \\
\qquad\quad (\texttt{Par}\ (\lambda(n, acc).\ n)\ (\lambda(n, acc).\ acc)) \\
\qquad\quad (\texttt{Par}\ (\texttt{Seq}\ (\texttt{Par}\ (\lambda(n, acc).\ n)\ (\lambda(n, acc).\ 1))\ (-)) \\
\qquad\qquad\quad (\texttt{Seq}\ (\texttt{Par}\ (\lambda(n, acc).\ n)\ (\lambda(n, acc).\ acc))\ (\times)))
\end{array}$$

The second step (**Step 2**) is to replace the combinators `Seq`, `Par`, `Ite` and `Rec` with corresponding circuit constructors `SEQ`, `PAR`, `ITE` and `REC` that compose handshaking devices (see the Appendix for their definitions). The key property of these constructors are the following theorems that enable us to compositionally deduce theorems of the form $\vdash Imp_C \Longrightarrow \texttt{DEV}\ f$, where Imp_C is a term constructed using the circuit constructors, and hence is a handshaking device (the long implication symbol $\Longrightarrow$ denotes implication lifted to functions – i.e. $f \Longrightarrow g\ =\ \forall x.\ f(x) \Rightarrow g(x)$):

$$\begin{array}{l}
\vdash\ \texttt{DEV}\ f\ \Longrightarrow\ \texttt{DEV}\ f \\[4pt]
\vdash\ (P_1\ \Longrightarrow\ \texttt{DEV}\ f_1)\ \wedge\ (P_2\ \Longrightarrow\ \texttt{DEV}\ f_2) \\
\quad \Rightarrow\ (\texttt{SEQ}\ P_1\ P_2\ \Longrightarrow\ \texttt{DEV}\ (\texttt{Seq}\ f_1\ f_2)) \\[4pt]
\vdash\ (P_1\ \Longrightarrow\ \texttt{DEV}\ f_1)\ \wedge\ (P_2\ \Longrightarrow\ \texttt{DEV}\ f_2) \\
\quad \Rightarrow\ (\texttt{PAR}\ P_1\ P_2\ \Longrightarrow\ \texttt{DEV}\ (\texttt{Par}\ f_1\ f_2)) \\[4pt]
\vdash\ (P_1\ \Longrightarrow\ \texttt{DEV}\ f_1)\ \wedge\ (P_2\ \Longrightarrow\ \texttt{DEV}\ f_2)\ \wedge\ (P_3\ \Longrightarrow\ \texttt{DEV}\ f_3) \\
\quad \Rightarrow\ (\texttt{ITE}\ P_1\ P_2\ P_3\ \Longrightarrow\ \texttt{DEV}\ (\texttt{Ite}\ f_1\ f_2\ f_3)) \\[4pt]
\vdash\ \texttt{Total}(f_1, f_2, f_3) \\
\quad \Rightarrow\ (P_1\ \Longrightarrow\ \texttt{DEV}\ f_1)\ \wedge\ (P_2\ \Longrightarrow\ \texttt{DEV}\ f_2)\ \wedge\ (P_3\ \Longrightarrow\ \texttt{DEV}\ f_3) \\
\quad \Rightarrow\ (\texttt{REC}\ P_1\ P_2\ P_3\ \Longrightarrow\ \texttt{DEV}\ (\texttt{Rec}\ f_1\ f_2\ f_3))
\end{array}$$

where $\texttt{Total}(f_1, f_2, f_3)$ is a predicate ensuring termination.

If e_c is an expression built using `Seq`,`Par`, `Ite` and `Rec`, then by suitably instantiating the predicate variables P_1, P_2 and P_3, these theorems allow us to construct an expression e_C built from circuit constructors SEQ, PAR, ITE and REC such that $\vdash e_C \Longrightarrow \texttt{DEV}\ e_c$. From Step 1 we have $\vdash f = e_c$, hence $\vdash e_C \Longrightarrow \texttt{DEV}\ f$

A function f which is combinational (i.e. can be implemented directly with logic gates without using registers) can be packaged as a handshaking device using a constructor ATM, which creates a simple handshake interface and satisfies the refinement theorem:

$\vdash\ \texttt{ATM}\ f\ \Longrightarrow\ \texttt{DEV}\ f$

The circuit constructor ATM is defined with the other constructors in the Appendix. To avoid a proliferation of internal handshakes, when the proof script that constructs e_C from e_c is implementing $\texttt{Seq}\ f_1\ f_2$, it checks to see whether f_1 or f_2 are compositions of combinational functions and if so introduces PRECEDE or FOLLOW instead of SEQ, using the theorems:

$\vdash\ (P\ \Longrightarrow\ \texttt{DEV}\ f_2)\ \Rightarrow\ (\texttt{PRECEDE}\ f_1\ P\ \Longrightarrow\ \texttt{DEV}\ (\texttt{Seq}\ f_1\ f_2))$
$\vdash\ (P\ \Longrightarrow\ \texttt{DEV}\ f_1)\ \Rightarrow\ (\texttt{FOLLOW}\ P\ f_2\ \Longrightarrow\ \texttt{DEV}\ (\texttt{Seq}\ f_1\ f_2))$

$\texttt{PRECEDE}\ f\ d$ processes inputs with f before sending them to d and $\texttt{FOLLOW}\ d\ f$ processes outputs of d with f. The definitions are:

$$\begin{aligned}&\texttt{PRECEDE}\ f\ d\ (load, inp, done, out) =\\ &\quad \exists v.\ \texttt{COMB}\ f\ (inp, v)\ \wedge\ d(load, v, done, out)\end{aligned}$$

$$\begin{aligned}&\texttt{FOLLOW}\ d\ f\ (load, inp, done, out) =\\ &\quad \exists v.\ d(load, inp, done, v)\ \wedge\ \texttt{COMB}\ f\ (v, out)\end{aligned}$$

$\texttt{COMB}\ f\ (v_1, v_2)$ drives v_2 with $f(v_1)$, i.e. $\texttt{COMB}\ f\ (v_1, v_2) = \forall t.\ v_2\ t = f(v_1\ t)$. The construction $\texttt{SEQ}\ d_1\ d_2$ introduces a handshake between the executions of d_1 and d_2, but $\texttt{PRECEDE}\ f\ d$ and $\texttt{FOLLOW}\ d\ f$ just 'wire' f before or after d, respectively, without introducing a handshake.

The result of Step 2 is a theorem $\vdash e_C \Longrightarrow \texttt{DEV}\ f$ where e_C is an expression built out of the circuit constructors ATM, SEQ, PAR, ITE, REC, PRECEDE and FOLLOW.

The third step (**Step 3**) is to rewrite with the definitions of these constructors (see their definitions in the Appendix) to get a circuit built out of standard kinds of gates (AND, OR, NOT and MUX), a generic combinational component $\texttt{COMB}\ g$ (where g will be a function represented as a HOL λ-expression) and Dtype registers.

The next phase of compilation converts terms of the form $\texttt{COMB}\ g\ (inp, out)$ into circuits built only out of components that it is assumed can be directly realised in hardware. Such components currently include Boolean functions (e.g. $\wedge$, $\vee$ and $\neg$), multiplexers and simple operations on n-bit words (e.g. versions of $+$, $-$ and $<$, various shifts etc.). A special purpose proof rule uses a straightforward recursive algorithm to synthesise combinational circuits. For example:

$$\begin{aligned}\vdash\ &\texttt{COMB}\ (\lambda(m,n).\ (m < n,\ \ m{+}1))\ (inp1\texttt{<>}inp2, out1\texttt{<>}out2)\ =\\ &\quad \exists v0.\ \texttt{COMB}\ (<)\ (inp1\texttt{<>}inp2, out1)\ \wedge\ \texttt{CONSTANT}\ 1\ v0\ \wedge\\ &\qquad\quad \texttt{COMB}\ (+)\ (inp1\texttt{<>}v0, out2)\end{aligned}$$

where <> is bus concatenation, $\texttt{CONSTANT}\ 1\ v0$ drives $v0$ high continuously, and $\texttt{COMB} <$ and $\texttt{COMB} +$ are assumed given components (if they were not given, then they could be implemented explicitly, but one has to stop somewhere).

The circuit resulting at the end of Step 3 uses unclocked abstract registers `DEL`, `DELT` and `DFF` that were chosen for convenience in defining `ATM`, `SEQ`, `PAR`, `ITE` and `REC` (see the Appendix). The register `DFF` is easily defined in terms of `DEL`, `DELT` and some combinational logic (details omitted).

The fourth step (**Step 4**) introduces a clock (with default name `clk`) and performs an automatic temporal abstraction as described in Melham's book [10] using the theorems:

$$\vdash \ \texttt{InfRise}\ clk \ \Rightarrow\ \forall d\ q.\ \texttt{Dtype}(clk,d,q) \ \Rightarrow\ \texttt{DEL}(d\ \texttt{at}\ clk,\ q\ \texttt{at}\ clk)$$

$$\vdash \ \texttt{InfRise}\ clk \ \Rightarrow\ \forall d\ q.\ \texttt{DtypeT}(clk,d,q) \ \Rightarrow\ \texttt{DELT}(d\ \texttt{at}\ clk,\ q\ \texttt{at}\ clk)$$

By instantiating *load*, *inp*, *done* and *out* in the theorem obtained by Step 3 to *load* `at` *clk*, *inp* `at` *clk*, *done* `at` *clk* and *out* `at` *clk*, respectively, and then performing some deductions using the above theorems and the monotonicity of existential quantification and conjunction with respect to implication, we obtain a theorem:

```
|- InfRise clk ==>
   circuit_f ==>
   DEV f (load at clk, inputs at clk, done at clk, outputs at clk)
```

4 Third party tools: `linRec`

The 'synthesisable subset' of HOL is the subset that can be automatically compiled to circuits. Currently this only includes tail-recursive function definitions. We anticipate compiling higher level specifications by using proof tools that translate into the synthsisable subset. Such tools are envisioned as 'third party' add-ons developed for particular applications. As a preliminary experiment we are implementing a tool `linRec` to translate linear recursions to tail-recursions. This would enable, for example, the automatic generation of `Mult32Iter` and `Fact32Iter` from the more natural definitions:

```
Mult32(m,n) = if m = 0w then 0w else m+Mult32(m-1w,n)
Fact32 n    = if n = 0w then 1w else n*Fact32(n-1)
```

A prototype implementation of `linRec` exists. It uses the following definition of linear and tail recursive recursion schemes:

```
linRec(x)     = if a(x) then b(x) else c (linRec(d x)) (e x)
tailRec(x,u) = if a(x) then c (b x) u else tailRec(d x, c (e x) u)
```

A linear recursion is matched against the definition of `linRec` to find values of `a`, `b`, `c`, `d`, `e` and then converted to a tail recursion by instantiating the theorem:

```
∀R a b c d e.
   WF R
   ∧ (∀x. ¬(a x) ==> R (d x) x)
   ∧ (∀p q r. c p (c q r) = c (c p q) r)
   ==>
   ∀x u. c (linRec a b c d e x) u = tailRec a b c d e (x,u)
```

where `WF R` means that `R` is well-founded. Heuristics are used to choose an appropriate witness for `R`.

5 Case studies

As part of a project to verify an ARM processor [5], a high-level model of the multiplication algorithm used by some ARM implementations was created in higher order logic. This is a Booth multiplier and we are using Fox's existing specification as an example for testing our compiler.

A more substantial example, being done at the University of Utah, is implementing the Advanced Encryption Standard (AES) [13] algorithm for private-key encryption. This specifies a multi-round algorithm with primitive computations based on finite field operations. The AES formalization includes a proof of functional correctness for the algorithm: specifically, encryption and decryption are inverse functions. Deriving the hardware from the proven specification using logical inference assures us that the hardware encrypter is the inverse of the hardware decrypter. An encryption round performs the following transformations on a 4-by-4 matrix of input bytes:

1. application of *sbox*, an invertible function from bytes to bytes, to each byte;
2. a cyclical shift of each row;
3. multiplication of each column by a fixed degree 3 polynomial, with coefficients in the 256 element finite field, GF(2^8);
4. adding a key to the matrix with exclusive OR.

We are exploring various options for generating components either as separate handshaking designs or expanding them into combinational logic. We have also explored converting our high-level recursive specification of multiplication into a table lookup. The resulting verified tables can then be stored into a RAM or ROM device. For synthesizing the tables directly into hardware, we have automated the definition of a function on bytes as a balanced `if` expression, branching on each successive bit of its input.

```
0xB ** x = if WORD_BIT 7 x then
             if WORD_BIT 6 x then
               ...
                       if WORD_BIT 0 then 0xA3 else 0xA8
               ...
           else
             if WORD_BIT 6 x then
               ...
```

Our experience so far is positive: compiling implementations by deduction provides a secure and flexible framework for creating and optimising designs.

6 Related work

Previous approaches to combine theorem provers and formal synthesis established an analogy between the goal-directed proof technique and an interactive design process. In LAMBDA, the user starts from the behavioural specification and builds the circuit incrementally by adding primitive hardware components which automatically simplify the goal [4]. Hanna *et al.* [7] introduce several *techniques* (functions) that simplify the current goal into simpler subgoals. Techniques are adaptations to hardware design of *tactics* in LCF.

Alternative approaches synthesise circuits by applying semantic-preserving transformations to their specifications. For instance, the Digital Design Derivation (DDD) transforms finite-state machines specified in terms of tail-recursive lambda abstractions into hierarchical Boolean systems [8]. Lava and Hydra are both hardware description languages embedded in Haskell whose programs consist of definitions of gates and their connections (netlists) [1, 12]. While Lava interfaces with external theorem provers to verify its circuits, Hydra designers can synthesise them via formal equational reasoning (using definitions and lemmas from functional programming). The functional languages μFP and Ruby adopt similar principles in hardware design [9, 15]. The circuits are defined in terms of primitive functions over Booleans, numbers and lists, and higher-order functions, the *combining forms*, which compose hardware blocks in different structures. Their mathematical properties provide a calculational style in design exploration.

These approaches deal with an interactive synthesis at the gate or state-machine level of abstraction only. Moreover, the synthesis and the proof of correctness require a substantial user guidance. Gropius and SAFL are two related works that address these issues.

Gropius is a hardware description language defined as a subset of HOL [2, 3]. Its algorithmic level provides control structures like if-then-else, sequential composition and while loop. The atomic commands are DFGs (data flow graphs) represented by lambda abstractions. The compiler initially combines every while loop into a single one at the outermost level of the program:

$$\texttt{PROGRAM}\ \mathit{out\_default}\ (\texttt{LOCVAR}\ \mathit{vars}\ (\texttt{WHILE}\ c\ (\texttt{PARTIALIZE}\ b)))$$

The body b of the `WHILE` loop is an acyclic DFG. The list *out_default* provides initial values for the output variables. The term `LOCVAR` declares the local variables *vars* and `PARTIALIZE` converts a non-recursive (terminating) DFG into a potentially non-terminating command. The compiler then synthesises a handshaking interface which encapsulates this program. Each of these hardware blocks are now regarded as primitive blocks or *processes* at the system level. Processes are connected via communication units (*k-processes*) which implement delay, synchronisation, duplication, splitting and joining of a process output data (actually there are 10 different k-processes [2]). Although the synthesis produces the proof of correctness of each process and k-process, the correctness of the top-level system is not generated. The reason for that is mainly because the top-level interface of a network of processes and k-processes does not match the handshaking interface pattern.

Our compilation method is partly inspired by SAFL (Statically Allocated Functional Language) [11], especially the ideas in Richard Sharp's PhD thesis [14]. SAFL is a first-order functional language whose programs consist of a sequence of tail-recursive function definitions. Its high-level of abstraction allows the exploitation of powerful program analyses and optimisations not available in traditional synthesis systems. However, the synthesis is not based on the correct-by-construction principles and the compiler has not been verified.

The novelty of our approach is the compilation of functional programs by composing especially designed and pre-verified circuit constructors. As each of these circuit constructors has the key property of implementing a device that computes precisely their corresponding combinators, the verification and the compilation of functional programs can be done automatically.

7 Current State and Future work

The compiler described here has been through several versions and now works robustly on all the examples we have tried. There were, however, some initial difficulties when we first experimented with Verilog simulation. Our formal model represents bits as Booleans (`T`, `F`), but the Verilog simulation model is multivalued (`1`, `0`, `x`, `z` etc.), so our formal model does not predict the Verilog simulation behavior in which registers are initialised to `x`. As a result, Verilog simulation was generating undefined `x`-values instead of the outputs predicted by our proofs. The behaviour of most real hardware does not correspond to Verilog simulation because in reality registers initialise to a definite value, which is `0` for the Altera FPGAs we are using. By making our Verilog model of `Dtype` initialise its state to `0` we were able to successfully simulate all our examples. Our investigation of this issue was complicated by a bug in the Verilog simulation test harness: `load` was being asserted before `done` became `T`, violating the precondition of the handshake protocol, so even after we understood the initialisation problem, simulation was giving inexplicable results. However, once we fixed the testbench, everything worked. All our examples now execute correctly both under simulation and on an Altera Excalibur FPGA board.

In the immediate future we plan to continue and complete the case studies described in Section 5.

At present all data-refinement (e.g. from numbers or enumerated types to words) must be done manually, by proof in higher order logic. The HOL4 system has some ‘boolification’ facilities that automatically translate higher level datatypes into bit-strings, and we hope to develop ‘third-party’ tools based on these that can be used for automatic data-refinement with the compiler.

We want to investigate using the compiler to generate test-bench monitors that can run in parallel simulation with designs that are not correct by construction. Thus our hardware can act as a “golden” reference against which to test other implementations.

The work described here is part of a bigger project to create hardware/software combinations by proof. We hope to investigate the option of creating software for ARM processors and linking it to hardware created by our compiler (possibly packaged as an ARM co-processor). Our emphasis is likely to be on cryptographic hardware and software, because there is a clear need for high assurance of correct implementation in this domain.

References

1. Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, January 1999.
2. Christian Blumenröhr. A formal approach to specify and synthesize at the system level. In *GI Workshop Modellierung und Verifikation von Systemen*, pages 11–20, Braunschweig, Germany, 1999. Shaker-Verlag.
3. Christian Blumenröhr and Dirk Eisenbiegler. Performing high-level synthesis via program transformations within a theorem prover. In *Proceedings of the Digital System Design Workshop at the Euromicro 98 Conference, Västeras, Sweden*, pages 34–37, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 1998. Online at: http://www.ubka.uni-karlsruhe.de/cgi-bin/psgunzip/1998/informatik/37/37.pdf.
4. Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal system design—interactive synthesis based on computer-assisted formal reasoning. In Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Volume 1*, pages 97–110, Houthalen, Belgium, November 1989. Elsevier Science Publishers, B.V. North-Holland, Amsterdam.
5. Anthony C. J. Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, The Computer Laboratory, University of Cambridge, England, November 2002.
6. Anthony C. J. Fox. HOL n-bit word Library, February 2004. Documentation available with the HOL4 system (http://hol.sourceforge.net/).
7. F.K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In L. Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 153–170. North-Holland, 1989.
8. Steven D. Johnson and Bhaskar Bose. DDD – A System for Mechanized Digital Design Derivation. Technical Report TR323, Indiana University, IU Computer Science Department, 1990. Available on the Internet at: http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR323.
9. Geraint Jones and Mary Sheeran. Circuit design in Ruby. Lecture notes on Ruby from a summer school in Lyngby, Denmark., September 1990. Online at: http://www.cs.chalmers.se/˜ms/papers.html.
10. Thomas F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, Cambridge, England, 1993. Cambridge Tracts in Theoretical Computer Science 31.
11. Alan Mycroft and Richard Sharp. Hardware/software co-design using functional languages. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 236–251, Genova, Italy, April 2001. Springer-Verlag. LNCS Vol. 2031.
12. John O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2002.
13. United States National Institute of Standards and Technology. Advanced Encryption Standard. Web: http://csrc.nist.gov/encryption/aes/, 2001.
14. Richard Sharp. *Higher-Level Hardware Synthesis*. PhD thesis, University of Cambridge, the Computer Laboratory, Cambridge, England, 2002.
15. Mary Sheeran. muFP, A language for VLSI design. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 104–112. ACM, ACM, August 1984.

16. Konrad Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, pages 381–398, Turku, Finland, August 1996. Springer-Verlag.

APPENDIX: formal specifications in higher order logic

The specification of the four-phase handshake protocol is represented by the definition of the predicate `DEV`, which uses auxiliary predicates `Posedge` and `HoldF`. A positive edge of a signal is defined as the transition of its value from low to high or, in our case, from `F` to `T`. The term `HoldF` $(t_1, t_2)\ s$ says that a signal s holds a low value `F` during a half-open interval starting at t_1 to just before t_2. The formal definitions are:

$$\begin{array}{l}
\vdash\ \texttt{Posedge}\ s\ t \quad = \ \texttt{if}\ t{=}0\ \texttt{then}\ \texttt{F}\ \texttt{else}\ (\neg\, s(t{-}1)\ \wedge\ s\ t\ )\\
\vdash\ \texttt{HoldF}\ (t_1, t_2)\ s = \ \forall t.\ t_1 \le t < t_2\ \Rightarrow\ \neg(s\ t)
\end{array}$$

The behaviour of the handshaking device computing a function f is described by the term `DEV` f $(load, inp, done, out)$ where:

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

The first conjunct in the right-hand side specifies that if the device is available and a positive edge occurs on *load*, there exists a time t' in future when *done* signals its termination and the output is produced. The value of the output at time t' is the result of applying f to the value of the input at time $t{+}1$. The signal *done* holds the value `F` during the computation. The second conjunct specifies the situation where no call is made on *load* and the device simply remains idle. Finally, the last conjunct states that if the device is busy, it will eventually finish its computation and become idle.

The circuit constructors

The following primitive components are used by the circuit constructors.

$$\begin{array}{l}
\vdash\ \texttt{AND}\ (in_1, in_2, out)\ = \forall t.\ out\ t = (in_1\ t\ \wedge\ in_2\ t)\\
\vdash\ \texttt{OR}\ (in_1, in_2, out)\ = \forall t.\ out\ t = (in_1\ t\ \vee\ in_2\ t)\\
\vdash\ \texttt{NOT}\ (inp, out)\ = \forall t.\ out\ t = \neg(inp\ t)\\
\vdash\ \texttt{MUX}(sw, in_1, in_2, out)\ = \ \forall t.\ out\ t = \texttt{if}\ sw\ t\ \texttt{then}\ in_1\ t\ \texttt{else}\ in_2\ t\\
\vdash\ \texttt{COMB}\ f\ (inp, out)\ = \forall t.\ out\ t = f(inp\ t)\\
\vdash\ \texttt{DEL}\ (inp, out)\ = \forall t.\ out(t{+}1) = inp\ t\\
\vdash\ \texttt{DELT}\ (inp, out)\ = (out\ 0 = \texttt{T})\ \wedge\ \forall t.\ out(t{+}1) = inp\ t\\
\vdash\ \texttt{DFF}(d, sel, q)\ = \ \forall t.\ q(t{+}1) = \texttt{if}\ \texttt{Posedge}\ sel\ (t{+}1)\ \texttt{then}\ d(t{+}1)\ \texttt{else}\ q\ t\\
\vdash\ \texttt{POSEDGE}(inp, out) = \exists c_0\ c_1.\ \texttt{DELT}(inp, c_0) \wedge \texttt{NOT}(c_0, c_1) \wedge \texttt{AND}(c_1, inp, out)
\end{array}$$

Atomic handshaking devices.

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

Sequential composition of handshaking devices.

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

Parallel composition of handshaking devices.

$$\begin{array}{l}
\vdash \texttt{PAR}\ f\ g\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out}) = \\
\quad \exists c_0\ c_1\ \mathit{start}\ \mathit{done}_1\ \mathit{done}_2\ \mathit{data}_1\ \mathit{data}_2\ \mathit{out}_1\ \mathit{out}_2. \\
\qquad \texttt{POSEDGE}(\mathit{load}, c_0)\ \wedge\ \texttt{DEL}(\mathit{done}, c_1)\ \wedge\ \texttt{AND}(c_0, c_1, \mathit{start})\ \wedge \\
\qquad f(\mathit{start}, \mathit{inp}, \mathit{done}_1, \mathit{data}_1)\ \wedge\ g(\mathit{start}, \mathit{inp}, \mathit{done}_2, \mathit{data}_2)\ \wedge \\
\qquad \texttt{DFF}(\mathit{data}_1, \mathit{done}_1, \mathit{out}_1)\ \wedge\ \texttt{DFF}(\mathit{data}_2, \mathit{done}_2, \mathit{out}_2)\ \wedge \\
\qquad \texttt{AND}(\mathit{done}_1, \mathit{done}_2, \mathit{done})\ \wedge\ (\mathit{out} = \lambda t.\ (\mathit{out}_1\ t, \mathit{out}_2\ t))
\end{array}$$

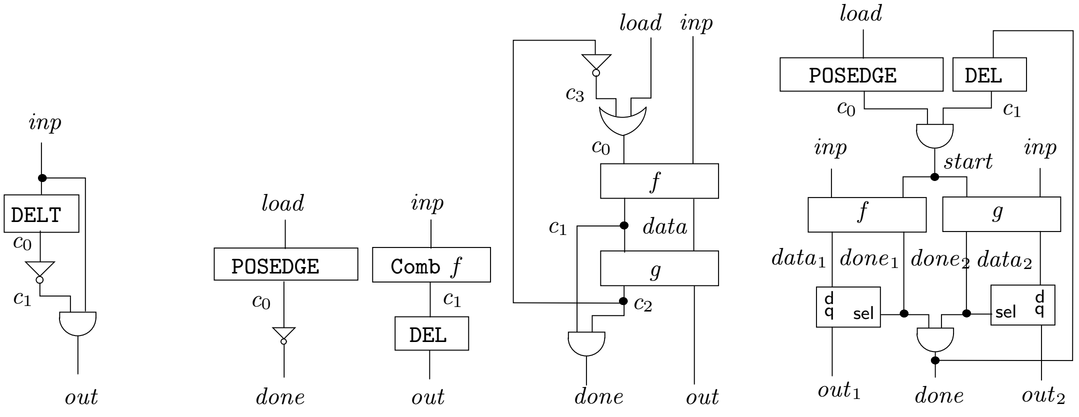
Conditional composition of handshaking devices.

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

Tail recursion constructor.

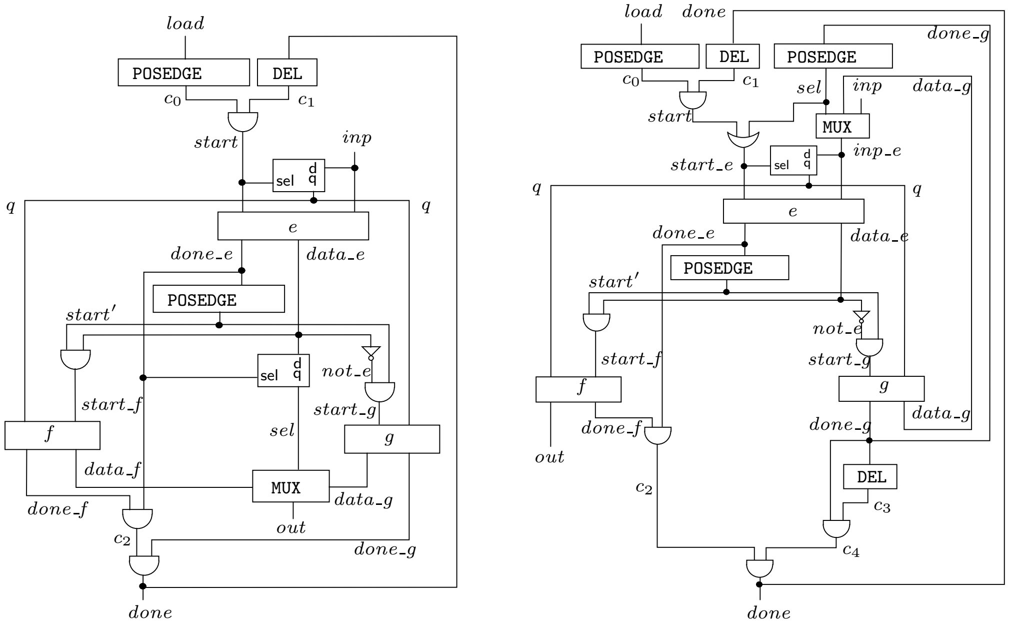
$$\begin{array}{l}
\vdash \texttt{REC}\ e\ f\ g\ (\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out}) = \\
\quad \exists \mathit{done\_g}\ \mathit{data\_g}\ \mathit{start\_e}\ q\ \mathit{done\_e}\ \mathit{data\_e}\ \mathit{start\_f}\ \mathit{start\_g}\ \mathit{inp\_e}\ \mathit{done\_f} \\
\quad\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

Circuit diagrams of the circuit constructors are shown on the following page.



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

Fig. 1. Implementation of composite devices.



(a) IF e f g (b) REC e f g

Fig. 2. The conditional and the recursive constructors.

A PVS Implementation of Stream Calculus for Signal Flow Graphs*

Hanne Gottliebsen

Department of Computer Science, Queen Mary, University of London

Abstract. We present early work on a PVS implementation of a model of simple control as signal flow graphs to enable formal verification of input/output behaviour of the control system. As has been shown by Rutten, Signal flow graphs can be described using Escardó's *coinductive stream calculus*, which includes a definition of differentiation for streams over the real numbers and the use of differential equations. The basics of coinductive stream calculus has been implemented in PVS.

1 Introduction

One of the graphical models used for control systems is signal flow graphs. Signal flow graphs were originally introduced by Mason [1] for modelling linear networks, and are now widely used in engineering to model data processing and automatic control.

To control an object means to influence its behaviour so as to achieve a desired goal. Control systems may be natural mechanisms, such as cellular regulation of genes and proteins by the gene control circuitry in DNA. They may be man-made - an early mechanical example was Watt's steam governor - but today most man-made control systems are digital, for example fighter aircraft or CD drives. Usually, in engineering we want to solve the problem of constructing a system with certain properties. Traditionally, control is treated as a mathematical phenomenon, modelled by continuous or discrete dynamical systems. Numerical computation is used to test and simulate these models, for example MATLAB is an industry standard in avionics.

The basic building blocks of signal flow graphs are branches, which allow for multiplication or delay, and nodes, which are either adding- or copying nodes. From this however, we can determine transfer functions for small graphs. Signal flow graphs can be represented using coinductive stream calculus, which gives a means of describing the effect of the various graph building block on a given input stream. Thus given a signal flow graph and an input stream we can use coinductive stream calculus to determine the output stream.

Rather than using eg. MATLAB to simulate the behaviour of our signal flow graphs, we want to use coinductive stream calculus to calculate the output streams. This allows us to have a PVS model of the signal flow graphs and

* This work is supported in part by The Nuffield Foundation.

their behaviour, thus we can reason formally about the input/output relationship for each signal flow graph by considering the various components and their compositions. We seek to use signal flow graphs and their representation using coinductive stream calculus to understand and reason about the input/output behaviour of control systems.

1.1 Structure

Section 2 gives an explanation of the main ideas and operations of coinductive stream calculus. In Section 3 we describe a basic implementation of stream calculus in PVS. Finally, in Section 4 we illustrate how the stream calculus can be used to prove equivalence between signal flow graphs and we consider some possible future work based on our implementation.

2 Signal Flow Graphs and Stream Calculus

Block diagrams are often used to represent systems with feedback graphically, for example in classical control a block diagram is a directed graph whose edges are labelled by rational functions over the complexes. They also allow more general representation of components described only by their input/output behaviour, corresponding to a more general notion of state. Signal flow graphs may be viewed as a particular type of block diagram, with the following restrictions: graph edges represent only scalar multiplication or unit delay and graph nodes either sum all their input or copy their output to several branches. However, by collapsing (parts of) signal flow graphs we may arrive at graphs with more complicated edge functions.

Figure 1 shows a simple signal flow graph. The input stream (coming from the left) is copied by the *copier*, C, and then the two resulting streams are added by the *adder*, +. The o is used to indicate composition of circuits. The result is that the output stream for this circuit is the input stream multiplied by 2.

There is also a basic circuit called an *a-multiplier*, which multiplies each element in the input stream by *a*, and a *register* circuit, which delays the stream by 1.

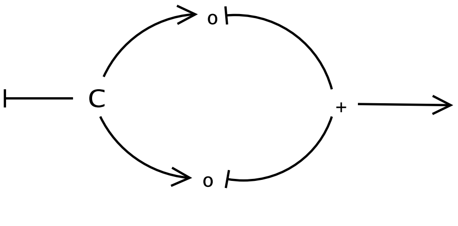


Fig. 1. Signal Flow Graph Multiplying Input Stream by 2

Pavlovic and Escardó [2] shows the connection between coinductive stream calculus and elementary calculus. Rutten [3] then describes coinductive stream calculus as a means of using streams of data to describe the behaviour of signal flow graphs, and in particular calculate the output stream based on the input stream. The coinductive stream calculus has notions of addition, multiplication, derivation and an inverse, which are all related to constructs in the signal flow graphs. Thus by modelling each part of the graphs by their input and output stream one may use compositionality to arrive at the behaviours for the complete graphs. One complication here is obviously the presence of loops in the circuits, but certain types of loops are known to give well-defined output streams. Within the stream calculus we may describe and solve linear and differential equations, which can be an easier and more direct route to finding the first elements of a stream composed from other streams, rather than calculating it directly.

2.1 Stream Calculus

Let us briefly introduce the notion of stream calculus as explained by Rutten [3]. We restrict the streams to the set $\mathbb{R}$ of real numbers. The reason for this is simply that for the signal flow graphs we only have streams over the real numbers, as opposed to some polymorphic version of streams. A stream is then a function from $\mathbb{N}$ to $\mathbb{R}$, and the set of streams over the reals are described by $\mathbb{R}^\omega$:

$$\mathbb{R}^\omega = \{\sigma | \sigma : \mathbb{N} \to \mathbb{R}\} \tag{1}$$

Rather than using the usual terminology of *head* and *tail* for a stream, we call $\sigma(0)$ the *initial value* of the stream σ is, and define the *derivative* σ' of the stream σ as

$$\sigma'(n) = \sigma(n+1) \tag{2}$$

Having this notion of a derivative allows the development of a *calculus of streams* which is fairly close to that of classical functional analysis.

We can now define addition and multiplication of streams as follows. The *sum*, $\sigma + \tau$ of streams σ and τ is element-wise, that is

$$\forall n \in \mathbb{N} : (\sigma + \tau)(n) = \sigma(n) + \tau(n) \tag{3}$$

The *convolution product*, $\sigma \times \tau$ of streams σ and τ is given by

$$\forall n \in \mathbb{N} : (\sigma \times \tau)(n) = \sum_{k=0}^{n} \sigma(k) \cdot \tau(n-k) \tag{4}$$

A particular kind of stream is $[r]$ with $r \in \mathbb{R}$. It is defined as follows:

$$[r] = (r, 0, 0, 0, \ldots) \tag{5}$$

This essentially allows us to add and multiply real numbers and streams:

$$[r] + \sigma = (r + \sigma(0), \sigma(1), \sigma(2), \ldots) \tag{6}$$

$$[r] \times \sigma = (r \cdot \sigma(0), r \cdot \sigma(1), r \cdot \sigma(2), \ldots) \tag{7}$$

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:

$$X = (0, 1, 0, 0, \ldots) \tag{8}$$

The effect of multiplying a stream by X is a delay of 1, that is:

$$X \times \sigma = (0, \sigma(0), \sigma(1), \sigma(2), \ldots) \tag{9}$$

With the above definitions of differentiation, addition and multiplication, we get the following facts about differentiation of sums and products:

$$(\sigma + \tau)' = \sigma' + \tau' \tag{10}$$
$$(\sigma \times \tau)' = ([\sigma(0)] \times \tau') + (\sigma' \times \tau) \tag{11}$$

We see that the sum behaves exactly as in classical calculus, however multiplication does not.

Eventually, we would like to be able to solve linear equations with streams, for example

$$\tau = 1 + (X \times \tau) \tag{12}$$

In order to do this in a manner similar to that which we normally use for functions, we need some definition of the *multiplicative inverse* of a stream. The property we are after is the following:

$$\frac{1}{\tau} \times \tau = [1] \tag{13}$$

This would allow us to solve Equation 12:

$$\tau = \frac{1}{1 - X} \tag{14}$$

However, since τ is a stream, it is not immediately clear what $\frac{1}{\tau}$ means. Rutten [3] defines the inverse using a *stream differential equation.*

So let us first discuss stream differential equations. Since the derivative of a stream is simply its tail, differential equations for stream are quite intuitive. Here is an example of a *higher order stream differential equation*:

derivative	initial values
$\tau'' = \tau$	$\tau(0) = 0,\ \ \tau'(0) = 1$

We can work out the value of τ from the differential equation:

$$\tau = \tau(0) : \tau' \tag{15}$$
$$= 0 : \tau'(0) : \tau'' \tag{16}$$
$$= 0 : 1 : \tau \tag{17}$$

Thus we see that $\tau = (0, 1, 0, 1, \ldots)$.

Of course, we may also have *first order* differential equations, in which case only the first derivative of the stream and one initial value is given. We can also have *systems* of differential equations, in this case two or more streams are given in terms of each other.

Now that we have defined stream differential equations, we can return to the definition of the inverse. The inverse $\frac{1}{\sigma}$ of a stream σ is defined for all streams with $\sigma(0) \neq 0$ by the following stream differential equation:

derivative	initial value
$(\frac{1}{\sigma})' = -\frac{1}{\sigma(0)} \times \sigma' \times \frac{1}{\sigma}$	$(\frac{1}{\sigma})(0) = \frac{1}{\sigma(0)}$

With this definition, we can prove the following lemmas, which are all properties we would expect the inverse to have:

$$\sigma \times \frac{1}{\sigma} = 1 \tag{18}$$

$$\frac{1}{\sigma} \times \frac{1}{\tau} = \frac{1}{\sigma \times \tau} \tag{19}$$

$$\frac{1}{\frac{1}{\sigma}} = \sigma \tag{20}$$

To summarise, we have seen what a stream is, and how we can perform basic operations - addition, multiplication, differentiation and multiplicative inverse - on streams.

3 Stream Calculus in PVS

In this section we will describe our basic implementation of stream calculus in PVS. This work is ongoing, so several interesting properties of stream calculus has not yet been implemented, and some which are included are in a somewhat crude form. We follow the structure of Section 2.

3.1 Basic Notion of Streams

PVS has a basic implementation of infinite sequences, which we use for the definition of our streams:

```
stream : TYPE = sequence[real]
```

In fact, we define our streams over a type, `A`, which is unspecified. However, for the exposition in this paper, the properties of such general streams are not needed, so we shall restrict ourselves to discussing streams over the real numbers. Next, we define the *derivative* of a stream, remember this corresponds to the tail of the stream:

```
derivative(sigma) : stream = lambda n: sigma(n+1)
```

We want to have a simple notation for the *k'th derivative* of a stream:

```
kth_derivative(sigma,k) : stream = lambda n: sigma(n+k)
```

We prove various fairly obvious properties about the (k'th) derivative, in particular the following correspondence between a stream and its derivatives:

```
Lemma1_1 : LEMMA
  sigma(n) = kth_derivative(sigma,n)(0)
```

That is, if we first take the n'th derivative and then look at the first element of the resulting stream, we get the n'th element of the original stream.

3.2 Calculating with Streams

We now define the sum and product as in Section 2:

```
sum(sigma,tau) : stream = lambda n : sigma(n) + tau(n)

prod(sigma,tau) : stream =
  lambda n :
    sigma(0,n,lambda k : IF k <= n THEN sigma(k) * tau(n-k)
                         ELSE 0
                         ENDIF)
```

For the product, we use a somewhat artificial if-statement. Since τ is defined only over the natural numbers, k should never exceed n. Obviously with the limits on the finite sum this is not going to occur, however the PVS typechecker still insists that we ensure $k <= n$. There are several ways of doing this, but so far we have found this one, while not pretty, to be the least intrusive when it comes to actually using the definition.

We then define the $[r]$ function, this is the stream that has the real number r as the first element and all other elements are 0:

```
r : VAR real
stream(r) : stream = lambda n : IF n = 0 THEN r ELSE 0 ENDIF
```

We also prove that $[r]$ behaves as expected with respect to addition and multiplication:

```
sum_r_sigma : LEMMA
  sum(stream(r),sigma) =
    lambda n : IF n = 0 THEN r + sigma(0) ELSE sigma(n) ENDIF

prod_r_sigma : LEMMA
  prod(stream(r),sigma) = lambda n : r*sigma(n)
```

Finally, we can define the negation of a stream, we simply multiply the stream by the stream generated by -1:

```
neg(sigma) : stream = prod(stream(-1),sigma)
```

This is a good point at which to convince ourselves that our specification behaves as we expected, and we do this by proving various lemma, for example that both addition and multiplication are commutative and that the distributive law holds for our streams:

```
distributive : LEMMA
  prod(sigma,sum(tau,rho)) =
    sum(prod(sigma,tau),prod(sigma,rho))
```

3.3 Polynomials

In order to simplify some notation, we define notions of

times which means to multiply a stream by n: times(sigma,n). This compares to writing $n\sigma$.

expt which means multiplying a stream with itself n times: expt(sigma,n). This compares to writing σ^n.

Defining the stream X exactly as before:

```
X : stream = lambda n : IF n = 1 THEN 1 ELSE 0 ENDIF
```

We then prove three key lemmas about X, that the following equations hold:

$$rX = \lambda n.if\ n = 1\ then\ r\ else\ 0\ endif \tag{21}$$

$$X \times \sigma = \lambda n.if\ n = 0\ then\ 0\ else\ \sigma(n-1)\ endif \tag{22}$$

$$X^k = \lambda n.if\ n = k\ then\ 1\ else\ 0\ endif \tag{23}$$

Now we run Exercise 2.4.c from [3]: Write $(1, 1, 1, 1, 1, 0, 0, 0, \ldots)$ using sum, product and X. Using the last equation above, we see that this can be done as follows:

$$X^0 + X + X^2 + X^3 + X^4 = (1, 1, 1, 1, 1, 0, 0, 0, \ldots) \tag{24}$$

3.4 Differential Equations

We can now define and try to solve stream differential equations in PVS. First we declare the type for first order stream differential equations:

```
basicFO_SCde : TYPE = [# initial : real, diff : stream #]
```

It simply holds the values for the initial value and the derivative. We also give a generic solution to the FO differential equations:

```
solve(bfode) : stream =
  lambda n : if n = 0 then bfode‘initial
             else bfode‘diff(n-1)
             endif
```

With these two definitions, we can solve for example the following differential equation:

derivative	initial value
$\tau' = \tau$	$\tau(0) = r$

In PVS, we get the following lemma:

```
example1_3a : LEMMA
  solve((# initial := r, diff := lambda n : r #)) =
    (lambda n : r)
```

We have seen how we can use stream differential equations to define streams. However, they can also be used to define functions on streams. This is an implementation of Example 1.5 from [3]. The differential equation is:

derivative	initial value
$even(\tau)' = even(\tau'')$	$even(\tau)(0) = \tau(0)$

Note that although the second derivative of τ occurs in the differential equation, it is actually still a first order differential equation, since it is defining *even* rather than τ. Indeed, the meaning of this is that for all possible streams τ, *even* should behave as given by the differential equation. Solving this differential equation, we see that

$$even(\tau) = (\tau(0), \tau(2), \tau(4), \ldots) \tag{25}$$

as we might have expected from the name of the function. In PVS, we have the following:

```
example1_5 : LEMMA
  even1 = (lambda tau :
             solve((# initial := tau(0),
                      diff := even1(kth_derivative(tau,2)) #)))
  IMPLIES
   even1 = (lambda tau : (lambda n : tau(2*n)))

even : [stream -> stream] = lambda tau : (lambda n : tau(2*n))
```

So we first prove that there is a function `even1` which solves the differential equation, and that it is on the form we calculated in (25). Next, we define the function *even* to be equal to that found solution.

At this point, this is the extent of the implementation of the theory of stream calculus in PVS. However, we have implemented several examples as well.

4 Signal Flow Graphs Represented using Streams

Let us now consider how streams can be used to describe signal flow graphs. Remember that we have the a-multiplier, Fig. 2. This works on the input stream σ to give the output stream $a\sigma$.

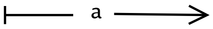


Fig. 2. a-multiplier: Multiplies the input stream with the real number a

We now consider the two circuits in Fig. 3 and want to use streams to describe each one and reason that they are equivalent (Example 3.1 in [3]). Starting from the left circuit (an a-multiplier followed by a b-multiplier):

$$
\begin{aligned}
[b] \times ([a] \times \sigma) &= ([b] \times [a]) \times \sigma && (26)\\
&= [ba] \times \sigma && (27)\\
&= [ab] \times \sigma && (28)
\end{aligned}
$$

The first two equalities comes from properties of multiplication of streams, these are both proven in in PVS. The last equality is simply commutativity of multiplication over the real numbers. Since $[ab] \times \sigma$ corresponds exactly to the effect of a ab-multiplier on σ we see that the two circuits in Fig. 3 are indeed equivalent.

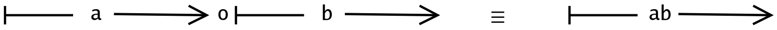


Fig. 3. a-multiplier followed by b-multiplier is equivalent to ab-multiplier

We intend to expand our implementation with a representation of signal flow graphs in PVS, linking this with the stream calculus and looking at extensions to the signal flow graphs. We also intend to provide some proof strategies for various typical properties one might want to prove, such as equivalence between circuits.

References

1. Mason, S. J.: Feedback Theory - Some properties of signal flowgraphs. Proceedings of IRE, vol.64, 1953
2. Pavlovic, D. and Escardó, M. H.: Calculus in Coinductive Form. Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science, 1998
3. Rutten, J. J. M. M.: An application of coinductive stream calculus to signal flow graphs. CWI Research Report SEN-E0305

Formal Verification of Chess Endgame Databases

Joe Hurd[⋆]

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

Abstract. Chess endgame databases store the number of moves required to force checkmate for all winning positions: with such a database it is possible to play perfect chess. This paper describes a method to construct endgame databases that are formally verified to logically follow from the laws of chess. The method employs a theorem prover to model the laws of chess and ensure that the construction is correct, and also a BDD engine to compactly represent and calculate with large sets of chess positions. An implementation using the HOL4 theorem prover and the BuDDY BDD engine is able to solve all four piece pawnless endgames.

1 Introduction

The game of chess with the modern rules came into existence in Italy towards the end of the 15th century [7]. The half millennium since then has witnessed many attempts to analyze the game, and in the last half century computers have naturally been used to extend the range of human analysis. One such approach uses computers to enumerate all possible positions of a certain type in an endgame database, working backwards from checkmate positions to determine the number of moves required to achieve checkmate from any starting position.

A survey paper by Heinz [6] cites Ströhlein's Ph.D. thesis from 1970 as the earliest publication on the algorithmic construction of endgame databases, and today endgame databases exist for all positions with five or fewer pieces on the board. Nalimov has started construction of the six piece endgames, but it is estimated that the finished database will require at least 1 terabyte of storage.

As an aside, it is still unclear whether or not access to endgame databases improves the strength of chess playing programs. However, they have found other uses by problemists in aiding the creation of endgame studies, and also by experts intepreting the computer analysis and writing instructional books for human players [10].

The attitude towards correctness of endgame databases is summed up by the following quotation in a paper comparing index schemes [9]:

> The question of data integrity always arises with results which are not self-evidently correct. Nalimov runs a separate self-consistency phase on each

[⋆] Supported by a Junior Research Fellowship at Magdalen College, Oxford.

> [endgame database] after it is generated. Both his [endgame databases] and those of Wirth yield exactly the same number of mutual zugzwangs [...] for all 2-to-5 man endgames and no errors have yet been discovered.

Applying computer theorem provers to chess endgame databases has two potential benefits:

- verifying the correctness of an endgame databases by proving that it faithfully corresponds to the rules of chess; and
- reducing the storage requirements by employing a more efficient representation than explicitly enumerating all possible positions.

For analyzing chess endgames this paper advocates the use of a higher order logic theorem prover integrated with a BDD engine. Higher order logic is very expressive, and it is possible to encode the rules of chess in a natural way, as an instance of a general class of two player games. On the other hand, BDDs can compactly represent sets of positions that have been encoded as boolean vectors, and the BDD engine can perform efficient calculation on these sets. The theorem prover ensures that the results of the BDD engine are faithfully lifted to the natural model of chess, and that all the reasoning is valid.

This methodology has been used to solve all four piece pawnless chess endgames, the product of which is a set of 'high assurance' endgame databases that provably correspond to a natural definition of chess. For example, given a chess position p in which it is Black to move, the theorem prover can fully automatically derive a theorem of the form

$$\vdash_{\mathsf{HOL+BDD}} \ \mathsf{win2\_by}\ \mathsf{chess}\ 15\ p \ \wedge\ \neg(\mathsf{win2\_by}\ \mathsf{chess}\ 14\ p)\ ,$$

which means that after any Black move from p, White can force checkmate within 15 moves but not within 14 moves. The $\vdash_{\mathsf{HOL+BDD}}$ symbol indicates that this theorem has been derived only from the inference rules of higher order logic and some BDD calculations. The only constants in this theorem are $\mathsf{win2\_by}$, which has a natural definition in the theory of two player games (see Section 2.1), and chess, which is a natural model of chess in higher order logic (see Section 2.2).

The primary contribution of this paper is a demonstration of the novel approach of verifying the correctness of an endgame database by proving its correspondence to a natural definition of chess, as opposed to testing its correspondence to another endgame database.

A secondary contribution of this paper is an investigation into the efficiency of BDDs to represent and calculate with sets of chess positions. Preliminary results in this area have already been obtained by Edelkamp, who calculated the number of BDD nodes to be 5% of the number of winning positions [3].

The structure of the paper is as follows: Section 2 presents a natural model of chess in higher order logic, which makes use of a general theory of two player games; Section 3 describes how an endgame database can be constructed in the theorem prover by rigorous proof; Section 4 presents the results; and Sections 5 and 6 conclude and look at related work.

2 Formalizing Chess in Higher Order Logic

The rules of chess are formalized in higher order logic in two phases. The first is a formalization of the theory of two player games, which is general enough to cover a large class of two player zero sum games with perfect information, including human games such as chess, checkers and go, and logic games such as Ehrenfeucht-Fraïssé pebble games.

The second phase defines the legal positions, move relations and winning positions of chess. Putting these into the two player game framework yields the crucial sets containing all positions in which White has a forced checkmate within n moves.

2.1 Two Player Games

The two players of the game are conventionally called *Player I* and *Player II.* In the general theory of two player games layed out in this section the positions have higher order logic type α, a type variable. This means that when the theory is applied to a specific game the type of positions can be instantiated to any concrete representation type.

A two player game G is modelled in higher order logic with a four tuple

$$(L, M, \overline{M}, W) \ ,$$

where L is a predicate on positions that holds if the position is legal. M is a relation between pairs of legal positions that holds if *Player I* can make a legal move from the first position to the second. Similarly, $\overline{M}$ is the move relation for *Player II.* Finally W is a predicate on legal positions that holds if the position is won for *Player I* (e.g., checkmate in chess). A game G is said to be well-formed (written $\mathsf{two\_player}\ G$) if the move relations and winning predicate are always false when given an illegal input position.

Intuitively, *Player I* wins a position if and only if it can be forcibly driven into a position satisfying W (within a finite number of moves). Given a well-formed game G, the following definitions make this intuition precise by carving out the set of legal positions that are eventually won for *Player I.* One way that *Player I* can fail to win is by reaching a non-winning position in which no moves are possible (e.g., stalemate in chess). This motivates the following two definitions:

$$\begin{aligned} \mathsf{terminal1}\ G &\equiv \{p \mid L_G(p) \ \wedge \ \forall p'.\ \neg M_G(p,p')\} \ ; \\ \mathsf{terminal2}\ G &\equiv \{p \mid L_G(p) \ \wedge \ \forall p'.\ \neg \overline{M}_G(p,p')\} \ . \end{aligned}$$

A position with *Player II* to move is won for *Player I* within zero moves if the predicate W is true of it:

$$\mathsf{win2\_by}\ G\ 0 \equiv \{p \mid W_G(p)\} \ .$$

A position with *Player I* to move is won for *Player I* within n moves if *Player I* can make a move to reach a position that is won for *Player I* within n moves:

$$\mathsf{win1\_by}\ G\ n \equiv \{p \mid \exists p'.\ M_G(p,p') \ \wedge \ p' \in \mathsf{win2\_by}\ G\ n\} \ .$$

Finally, a position with *Player II* to move is won for *Player I* within $n+1$ moves if it is won within n moves, or (i) it is not a terminal position and (ii) every move that player *Player I* makes will reach a position that is won for *Player I* within n moves:

$$\begin{array}{l} \mathsf{win2\_by}\ G\ (n+1) \equiv \\ \ \ \mathsf{win2\_by}\ G\ n\ \cup \\ \ \ \{p \mid L_G(p)\ \wedge\ \forall p'.\ \overline{M}_G(p,p')\ \Rightarrow\ p' \in \mathsf{win1\_by}\ G\ n\} - \mathsf{terminal2}\ G\ . \end{array}$$

Also of interest is the set of all positions that are eventually winning for *Player I*, which is defined separately for the cases of *Player I* to move and *Player II* to move:

$$\begin{array}{rcl} \mathsf{win1}\ G & \equiv & \{p \mid \exists n.\ p \in \mathsf{win1\_by}\ G\ n\}\ ; \\ \mathsf{win2}\ G & \equiv & \{p \mid \exists n.\ p \in \mathsf{win2\_by}\ G\ n\}\ . \end{array}$$

The preceding definitions provide all the theory of two player games that is necessary to interpret theorems resulting from a query of a verified endgame database.

2.2 Chess

The authoritative version of the laws of chess is the FIDE[1] handbook [4]. Section E.I of the handbook is entitled *Laws of Chess*, and in a series of articles describes the object of the game, the movement of the pieces and how the players should conduct themselves. For example, Article 1 is entitled *The nature and objectives of the game of chess*

> **Article 1.1.** The game of chess is played between two opponents who move their pieces alternately on a square board called a 'chessboard'. [. . .]

which confirms that chess is an instance of the general class of two player games formalized in the previous section.

The first design choice that occurs in the formalization of chess is to decide which higher order logic type will be used to represent chess positions. The results in this paper cover only pawnless endgames in which castling is forbidden, so the only information that needs to be tracked by the position type is the side to move and the location of the pieces on the board. The key types used to represent chess positions are:

$$\begin{array}{rcl} \text{side} & \equiv & \mathsf{White} \mid \mathsf{Black}\ ; \\ \text{piece} & \equiv & \mathsf{King} \mid \mathsf{Queen} \mid \mathsf{Rook} \mid \mathsf{Bishop} \mid \mathsf{Knight}\ ; \\ \text{square} & \equiv & \mathbb{N} \times \mathbb{N}\ ; \\ \text{position} & \equiv & \text{side} \times (\text{square} \to (\text{side} \times \text{piece})\ \text{option})\ . \end{array}$$

Sides and pieces simply enumerate the possibilities. In the context of the two player game of chess, this paper will follow the convention of referring to *Player I*

[1] FIDE (Fédération Internationale des Échecs) is the World Chess Federation.

as White and *Player II* as Black. Squares are pairs of natural numbers, and a position is a pair of the side to move and a partial function from squares to pieces. For convenience and readability, a few basic functions are defined for examining positions:

$$\begin{aligned}
\mathsf{opponent}\ s &\equiv \mathsf{case}\ s\ \mathsf{of}\ \mathsf{White} \rightarrow \mathsf{Black} \mid \mathsf{Black} \rightarrow \mathsf{White}\ ; \\
\mathsf{to\_move}\ (s, \_) &\equiv s\ ; \\
\mathsf{on\_square}\ (\_, f)\ sq &\equiv f\ sq\ ; \\
\mathsf{empty}\ p\ sq &\equiv (\mathsf{on\_square}\ p\ sq = \mathsf{NONE})\ ; \\
\mathsf{occupies}\ p\ s\ sq &\equiv \exists v.\ \mathsf{on\_square}\ p\ sq = \mathsf{SOME}\ (s, v)\ .
\end{aligned}$$

Once the type representing the game state is fixed, what remains to apply the general theory of two player games is a higher order logic encoding of the legal positions, move relations and winning positions of chess. Such an encoding is a routine formalization, and the remainder of this section demonstrates how naturally the laws of chess can be represented in higher order logic.

Article 2 of the laws of chess in the FIDE handbook describes the geometry of the chessboard:

> **Article 2.1.** The chessboard is composed of an 8 × 8 grid of 64 equal squares alternately light (the 'white' squares) and dark (the 'black' squares). The chessboard is placed between the players in such a way that the near corner square to the right of the player is white.
> **Article 2.4.** The eight vertical columns of squares are called 'files'. The eight horizontal rows of squares are called 'ranks'. A straight line of squares of the same colour, touching corner to corner, is called a 'diagonal'.

This is encoded into higher order logic with the following definitions:

$$\begin{aligned}
\mathsf{files} &\equiv 8\ ; \\
\mathsf{ranks} &\equiv 8\ ; \\
\mathsf{file}\ (f, r) &\equiv f\ ; \\
\mathsf{rank}\ (f, r) &\equiv r\ ; \\
\mathsf{board} &\equiv \{sq \mid \mathsf{file}\ sq < \mathsf{files} \wedge \mathsf{rank}\ sq < \mathsf{ranks}\}\ ; \\
\mathsf{same\_file}\ sq\ sq' &\equiv (\mathsf{file}\ sq = \mathsf{file}\ sq')\ ; \\
\mathsf{same\_rank}\ sq\ sq' &\equiv (\mathsf{rank}\ sq = \mathsf{rank}\ sq')\ ; \\
\mathsf{same\_diag1}\ sq\ sq' &\equiv (\mathsf{file}\ sq + \mathsf{rank}\ sq = \mathsf{file}\ sq' + \mathsf{rank}\ sq')\ ; \\
\mathsf{same\_diag2}\ sq\ sq' &\equiv (\mathsf{file}\ sq + \mathsf{rank}\ sq' = \mathsf{file}\ sq' + \mathsf{rank}\ sq)\ ; \\
\mathsf{diff}\ m\ n &\equiv \mathsf{if}\ m \leq n\ \mathsf{then}\ n - m\ \mathsf{else}\ m - n\ ; \\
\mathsf{file\_diff}\ sq\ sq' &\equiv \mathsf{diff}\ (\mathsf{file}\ sq)\ (\mathsf{file}\ sq')\ ; \\
\mathsf{rank\_diff}\ sq\ sq' &\equiv \mathsf{diff}\ (\mathsf{rank}\ sq)\ (\mathsf{rank}\ sq')\ .
\end{aligned}$$

Notice that the presentational aspect of white and black squares is not included in the higher order logic encoding, only the logically important aspect of the board being an 8 × 8 grid of squares.

Article 3 is entitled *The moves of the pieces*:

> **Article 3.2.** The bishop may move to any square along a diagonal on which it stands.
> **Article 3.3.** The rook may move to any square along the file or the rank on which it stands.

> **Article 3.4.** The queen may move to any square along the file, the rank or a diagonal on which it stands.
> **Article 3.5.** When making these moves the bishop, rook or queen may not move over any intervening pieces.
> **Article 3.6.** The knight may move to one of the squares nearest to that on which it stands but not on the same rank, file or diagonal.
> **Article 3.8.** There are two different ways of moving the king, by:
> 1. moving to any adjoining square not attacked by one or more of the opponent's pieces. The opponent‘s pieces are considered to attack a square, even if such pieces cannot themselves move.
> 2. or 'castling'. [...]

The moves are encoded into higher order logic in three steps. In the first step the basic moves of the pieces are defined:

$$\begin{aligned}
\mathsf{bishop\_attacks}\ sq_1\ sq_2 &\equiv (\mathsf{same\_diag1}\ sq_1\ sq_2 \lor \mathsf{same\_diag2}\ sq_1\ sq_2) \land sq_1 \neq sq_2\ ; \\
\mathsf{rook\_attacks}\ sq_1\ sq_2 &\equiv (\mathsf{same\_file}\ sq_1\ sq_2 \lor \mathsf{same\_rank}\ sq_1\ sq_2) \land sq_1 \neq sq_2\ ; \\
\mathsf{queen\_attacks}\ sq_1\ sq_2 &\equiv \mathsf{rook\_attacks}\ sq_1\ sq_2 \lor \mathsf{bishop\_attacks}\ sq_1\ sq_2\ ; \\
\mathsf{knight\_attacks}\ sq_1\ sq_2 &\equiv ((\mathsf{file\_diff}\ sq_1\ sq_2 = 1) \land (\mathsf{rank\_diff}\ sq_1\ sq_2 = 2)) \lor \\
&\ ((\mathsf{file\_diff}\ sq_1\ sq_2 = 2) \land (\mathsf{rank\_diff}\ sq_1\ sq_2 = 1))\ ; \\
\mathsf{king\_attacks}\ sq_1\ sq_2 &\equiv \mathsf{file\_diff}\ sq_1\ sq_2 \leq 1 \land \mathsf{rank\_diff}\ sq_1\ sq_2 \leq 1 \land sq_1 \neq sq_2\ .
\end{aligned}$$

To improve clarity, the definition of the basic moves is closer to an explanation typically found in a beginner's chess book rather than the letter of the articles. For example, the queen is explicitly defined to move like a rook or a bishop, and the definition of the knight move follows the traditional L-shape explanation rather than the article's more geometric explanation of "[nearest square] not on the same rank, file or diaganal".[2]

The second step formalizes the no-jumping requirement of Article 3.5 by defining the concept of a clear line from a square: all the squares that can be reached horizontally, vertically or diagonally without jumping over any intervening pieces:

$$\begin{aligned}
&\mathsf{between}\ n_1\ n\ n_2\ \equiv\ (n_1 < n \land n < n_2) \lor (n_2 < n \land n < n_1)\ ; \\
&\mathsf{square\_between}\ sq_1\ sq\ sq_2\ \equiv \\
&\ \ \mathsf{if\ same\_file}\ sq_1\ sq_2\ \mathsf{then\ same\_file}\ sq\ sq_1 \land \mathsf{between}\ (\mathsf{rank}\ sq_1)\ (\mathsf{rank}\ sq)\ (\mathsf{rank}\ sq_2) \\
&\ \ \mathsf{else\ if\ same\_rank}\ sq_1\ sq_2\ \mathsf{then\ same\_rank}\ sq\ sq_1 \land \mathsf{between}\ (\mathsf{file}\ sq_1)\ (\mathsf{file}\ sq)\ (\mathsf{file}\ sq_2) \\
&\ \ \mathsf{else\ if\ same\_diag1}\ sq_1\ sq_2\ \mathsf{then\ same\_diag1}\ sq\ sq_1 \land \mathsf{between}\ (\mathsf{file}\ sq_1)\ (\mathsf{file}\ sq)\ (\mathsf{file}\ sq_2) \\
&\ \ \mathsf{else\ if\ same\_diag2}\ sq_1\ sq_2\ \mathsf{then\ same\_diag2}\ sq\ sq_1 \land \mathsf{between}\ (\mathsf{file}\ sq_1)\ (\mathsf{file}\ sq)\ (\mathsf{file}\ sq_2) \\
&\ \ \mathsf{else}\ \bot\ ; \\
&\mathsf{clear\_line}\ p\ sq_1 \equiv \{sq2 \mid \forall sq.\ \mathsf{square\_between}\ sq_1\ sq\ sq_2 \Rightarrow \mathsf{empty}\ p\ sq\}
\end{aligned}$$

The definition of square_between formalizes the notion of a square lying strictly between two others in a straight line: the verbosity is a normal consequence of using algebraic formulas to capture an essentially geometric concept.

[2] A more succinct definition that illustrates the L-shape even better is

$$\mathsf{knight\_attacks}\ sq_1\ sq_2 \equiv (\{\mathsf{file\_diff}\ sq_1\ sq_2, \mathsf{rank\_diff}\ sq_1\ sq_2\} = \{1, 2\})\ ,$$

but this has the drawback of requiring a moment's thought to see that it is correct.

In the third and final step, the basic moves of the pieces and clear lines are brought together to define the set of squares attacked from a square.

$$\begin{array}{l}
\mathsf{attacks}\ p\ sq \equiv \\
\quad \mathsf{board} \cap \mathsf{clear\_line}\ p\ sq \cap \\
\quad (\mathsf{case\ on\_square}\ p\ sq\ \mathsf{of} \\
\qquad \mathsf{NONE} \to \emptyset \\
\qquad \mid \mathsf{SOME}\ (\_, \mathsf{King}) \to \{sq' \mid \mathsf{king\_attacks}\ sq\ sq'\} \\
\qquad \mid \mathsf{SOME}\ (\_, \mathsf{Queen}) \to \{sq' \mid \mathsf{queen\_attacks}\ sq\ sq'\} \\
\qquad \mid \mathsf{SOME}\ (\_, \mathsf{Rook}) \to \{sq' \mid \mathsf{rook\_attacks}\ sq\ sq'\} \\
\qquad \mid \mathsf{SOME}\ (\_, \mathsf{Bishop}) \to \{sq' \mid \mathsf{bishop\_attacks}\ sq\ sq'\} \\
\qquad \mid \mathsf{SOME}\ (\_, \mathsf{Knight}) \to \{sq' \mid \mathsf{knight\_attacks}\ sq\ sq'\})\ .
\end{array}$$

Having defined the moves of the pieces, it is straightforward to formalize the set of legal positions. According to the laws of chess, a position is legal if the side that has just moved is not in check:

> **Article 3.9.** The king is said to be 'in check' if it is attacked by one or more of the opponent's pieces, even if such pieces are constrained from moving to that square because they would then leave or place their own king in check. No piece can be moved that will expose its own king to check or leave its own king in check.

In addition to this, the type of chess positions makes it necessary to require that all of the pieces are on the board. Without this extra requirement, the formalization would capture the game of chess being played on an infinite board!

$$\begin{array}{l}
\mathsf{in\_check}\ s\ p \equiv \\
\quad \exists sq_1, sq_2. \\
\qquad (\mathsf{on\_square}\ p\ sq_1 = \mathsf{SOME}\ (s, \mathsf{King})) \wedge \\
\qquad \mathsf{occupies}\ p\ (\mathsf{opponent}\ s)\ sq_2 \wedge sq_1 \in \mathsf{attacks}\ p\ sq_2\ ; \\
\mathsf{all\_on\_board}\ p \equiv \forall sq.\ \neg\mathsf{empty}\ p\ sq \Rightarrow sq \in \mathsf{board}\ ; \\
\mathsf{chess\_legal}\ p \equiv \mathsf{all\_on\_board}\ p \wedge \neg\mathsf{in\_check}\ (\mathsf{opponent}\ (\mathsf{to\_move}\ p))\ p\ .
\end{array}$$

Using everything that has been defined so far, it is easy to formalize the move relations $\mathsf{chess\_move1}$ (for the White pieces) and $\mathsf{chess\_move2}$ (for the Black pieces). In a nutshell, a move is either a simple move of a piece to an empty square, or a capturing move of a piece to a square occupied by an opponent's piece. For the full details of how this is formalized, please refer to Appendix A.

Finally, all that remains is to define the set of positions that are winning for the player of the White pieces. This is covered back in Article 1, *The nature and objectives of the game of chess*:

> **Article 1.2.** The objective of each player is to place the opponent's king 'under attack' in such a way that the opponent has no legal move. The player who achieves this goal is said to have 'checkmated' the opponent's king and to have won the game. [. . .]

This wordy article can be concisely formalized in higher order logic:

$$\begin{array}{rcl}
\mathsf{game\_over}\ p & \equiv & \mathsf{chess\_legal}\ p \wedge \forall p'.\ \neg\mathsf{chess\_move}\ p\ p'\ ; \\
\mathsf{checkmated}\ p & \equiv & \mathsf{game\_over}\ p \wedge \mathsf{in\_check}\ (\mathsf{to\_move}\ p)\ p\ ; \\
\mathsf{chess\_win}\ p & \equiv & (\mathsf{to\_move}\ p = \mathsf{Black}) \wedge \mathsf{checkmated}\ p\ .
\end{array}$$

Finally, the legal positions, move relations and winning positions are put together to define the two player game of chess:

$$\mathsf{chess} \equiv (\mathsf{chess\_legal}, \mathsf{chess\_move1}, \mathsf{chess\_move2}, \mathsf{chess\_win}) \ .$$

The remainder of this paper presents a method for automatically constructing endgame databases that are formally verified with respect to this theory of the laws of chess. However, it is also possible to prove theorems interactively in the theorem prover, such as the result that a player with only a King can never win. Given a ternary relation $\mathsf{has\_pieces}\ s\ l\ p$ (defined in Appendix A) that holds whenever the side s has precisely the list of pieces l on the board in the position p, it is straightforward to prove the desired theorem

$$\vdash_{\mathsf{HOL}} \forall p.\ \mathsf{all\_on\_board}\ p \wedge \mathsf{has\_pieces\ White\ [King]}\ p \Rightarrow \neg \mathsf{chess\_win}\ p$$

by manually directing the theorem prover to apply standard proof tactics.

3 Constructing Formally Verified Endgame Databases

Recall from Section 2.1 that $\mathsf{win2\_by\ chess}\ n$ is a set of legal chess positions with Black (i.e., *Player II*) to move. The set contains all positions such that however Black moves White can force a checkmate within n moves. By convention the set $\mathsf{win2\_by\ chess}\ 0$ contains all positions where White has already won (i.e., Black is checkmated). Similarly, $\mathsf{win1\_by\ chess}\ n$ is a set of legal positions with White to move. This set contains all positions where there is a White move after which the resulting position lies in the $\mathsf{win2\_by\ chess}\ n$ set: in the chess jargon a position in the $\mathsf{win1\_by\ chess}\ n$ set is called a *mate in* $n + 1$.

Constructing a formally verified endgame database consists of evaluating the $\mathsf{win1\_by\ chess}\ n$ and $\mathsf{win2\_by\ chess}\ n$ sets in the theorem prover. The first problem that occurs is that these sets are extremely large: even with just four pieces on the board, the total number of winning positions can be ten of millions. Thus it is not feasible to aim to prove a theorem of the form

$$\vdash_{\mathsf{HOL}} \mathsf{win1\_by\ chess}\ n = \{p_1, \ldots, p_N\} \ ,$$

where the p_i are an explicit enumeration of the positions in the winning set. Instead, the winning sets are represented symbolically using Binary Decision Diagrams [2], which provide a compact way to represent sets of boolean vectors. A theorem of the form

$$\vdash_{\mathsf{HOL+BDD}} \phi[B_1, \ldots, B_k] \in \mathsf{win1\_by\ chess}\ n \mapsto \mathit{\Delta} \tag{1}$$

is proved, where $[B_1, \ldots, B_k]$ is a vector of boolean variables that encode a position, ϕ is a decoding function from an encoding to a position, and $\mathit{\Delta}$ is a BDD representing a set of boolean vectors. The theorem asserts that for any assignment of booleans b_i to the variables B_i, the position $\phi[b_1, \ldots, b_k]$ is a forced win for White within n moves if and only if the vector $[b_1, \ldots, b_k]$ is in the set represented by the BDD $\mathit{\Delta}$.

The following two sections will discuss the encoding of positions as boolean variables, and the proof tools required to construct theorems of the above form in the theorem prover.

3.1 Encoding Positions as Boolean Variables

The formalization of the laws of chess presented in Section 2.2 is designed to be as natural as possible, so that a human reader (familiar with higher order logic) can be easily convinced that it is a faithful translation of the laws of chess. However, it fails to satisfy two basic requirements for encoding positions as boolean vectors:

1. The position type should be easy to encode as a vector of booleans. Although there are tools in the theorem prover to support boolean encoding of (bounded) numbers and lists, the function from squares to pieces in the position type would require a custom encoder to be written and proved correct.
2. Given a list of White and Black pieces, it should be straightforward to define the set of all positions that have precisely these pieces on the board, since that is how endgame databases are structured. Unfortunately, the square based nature of the position type makes it inconvenient to reason about the pieces on the board.

For both these reasons, the boolean encoding of positions makes use of an intermediate ‘posn’ type defined as follows:

$$\begin{aligned} \text{placement} &\equiv (\text{side} \times \text{piece}) \times \text{square} \;; \\ \text{posn} &\equiv \text{side} \times \text{placement list} \;. \end{aligned}$$

Versions of the legal position predicates, move relations and winning position predicate are defined on type posn, and their definitions are designed for ease of boolean encoding. In addition, a function

$$\mathsf{abstract} : \text{posn} \to \text{position}$$

is defined that lifts elements of type posn to chess positions. With respect to the $\mathsf{abstract}$ function, the two versions of the legal position predicates, move relations and winning position predicates are identical: a useful check for both versions.

The new posn type also satisfies the requirement that positions should be easily categorized according to the pieces on the board. Define a category to be a side to move and a list of pieces on the board:

$$\text{category} \equiv \text{side} \times (\text{side} \times \text{piece}) \text{ list} \;.$$

For example

$$(\mathsf{Black},\ [(\mathsf{White}, \mathsf{King}),\ (\mathsf{White}, \mathsf{Rook}),\ (\mathsf{Black}, \mathsf{King})])$$

is the category of positions where it is Black to move, White has a King and Rook on the board, and Black has only a King. The set of all elements of the posn type in a category (s, l) can be defined as

$$\mathsf{category}\ (s, l) \ \equiv\ \{(s', l') \mid s' = s \wedge \mathsf{map\ fst}\ l' = l\}\ ,$$

where map is the standard list map function and fst is the function that picks the first component from a product.

For each category c, all the positions p in $\mathsf{category}\ c$ are encoded to booleans in the same way. The side to move and pieces in p are fixed, so the only 'state' left to encode as booleans are the squares that the pieces are on, which is a fixed length list of pairs of bounded natural numbers. Encoding this type is a relatively straightforward matter of plumbing together the standard boolean encoders for fixed length lists, products and bounded natural numbers that are already defined in the theorem prover [11]. Given a category c, this process yields a function $\mathsf{encode\_posn}\ c$ for encoding posns in category c as a vector of booleans, and an inverse function $\mathsf{decode\_posn}\ c$ for decoding a vector of booleans as a posn in category c.

For positions in a category c, the decoder function ϕ in Equation (1) can now be expanded to

$$\mathsf{abstract} \circ \mathsf{decode\_posn}\ c\ .$$

3.2 Proving Endgame Database Theorems

The verified endgame database is constructed category by category by symbolically evaluating the winning sets (i.e., calculating the BDDs Δ in Equation (1) for increasing values of n). When a fixed point is found, a stability theorem is proved which is lifted to the position type using $\mathsf{to\_move}$ and $\mathsf{has\_pieces}$ predicates. For example, the lifted stability theorem

$$\begin{array}{l}
\vdash_{\mathsf{HOL+BDD}} \\
\quad \forall p. \\
\qquad \mathsf{all\_on\_board}\ p \wedge (\mathsf{to\_move}\ p = \mathsf{Black}) \wedge \\
\qquad \mathsf{has\_pieces}\ p\ \mathsf{White}\ [\mathsf{King}, \mathsf{Rook}] \wedge \mathsf{has\_pieces}\ p\ \mathsf{Black}\ [\mathsf{King}] \Rightarrow \\
\qquad (p \in \mathsf{win2\ chess} \iff p \in \mathsf{win2\_by\ chess}\ 16)
\end{array}$$

states that for positions with Black to move, White having a King and Rook and Black having only a King, if a position is won at all for White then checkmate can be forces within 16 moves. In addition, a concrete position is lifted from the final BDDs to show that this bound is the best possible:

$$\begin{array}{l}
\vdash_{\mathsf{HOL+BDD}} \\
\quad (\mathsf{Black}, \\
\quad\ \lambda sq. \\
\qquad \mathsf{if}\ sq = (0,0)\ \mathsf{then\ SOME\ (White, King)\ else\ if}\ sq = (5,6)\ \mathsf{then\ SOME\ (White, Rook)} \\
\qquad \mathsf{else\ if}\ sq = (3,6)\ \mathsf{then\ SOME\ (Black, King)\ else\ NONE)}\ \in\ \mathsf{win2\_by\ chess}\ 16\ \wedge \\
\quad (\mathsf{Black}, \\
\quad\ \lambda sq. \\
\qquad \mathsf{if}\ sq = (0,0)\ \mathsf{then\ SOME\ (White, King)\ else\ if}\ sq = (5,6)\ \mathsf{then\ SOME\ (White, Rook)} \\
\qquad \mathsf{else\ if}\ sq = (3,6)\ \mathsf{then\ SOME\ (Black, King)\ else\ NONE)}\ \notin\ \mathsf{win2\_by\ chess}\ 15\ .
\end{array}$$

Calculating the sequence of BDDs representing winning sets for a category is implemented using the category-specific boolean encoding of the move relations and winning position predicate. The winning position predicate is converted to a BDD, and this becomes the first BDD in the sequence. The move relation is also converted to a BDD, and applied to the current winning set to find the set of positions that for which the current winning set is reachable in one White move (this new winning set consists of all the *mate in one* positions). The BDD resulting from this calculation is added to the sequence of BDDs, and becomes the current winning set. The BDD for the move relation is now applied again, but with a universal instead of an existential quantifier, to calculate the set of positions such that *all* Black moves result in a position in the current winning set. The BDD representing this winning set is added to the sequence of BDDs, and becomes the current winning set. This sequence of BDDs representing winning sets is continued until it converges to a fixed point (i.e., the winning set with Black to move is the same as the previous winning set with Black to move).

Since pieces may get captured during play, and this changes the category of the position, it is important to construct the endgame databases for the small categories first, so that captures always reduce to a previously solved position. The base case is two bare Kings on the board, and then different pieces are added to first solve all the three piece endgames, and then the four piece endgames.

There are potential pitfalls to symbolically calculating the winning sets that do not appear in the usual method of explicitly enumerating all positions, but the theorem prover ensures that the reasoning is sound and that no positions are left out. For example, consider the category

$$(\mathsf{White},\ [(\mathsf{White},\mathsf{King}),\ (\mathsf{White},\mathsf{Queen}),\ (\mathsf{White},\mathsf{Rook}),\ (\mathsf{Black},\mathsf{King})])$$

where from any starting position White needs at most six moves to force checkmate. Indeed, during construction of the sequence of BDDs they are seen to converge after six moves. However, because this category of endgame can reduce by a capture to the category where White has a King and Rook against Black's bare King, and because in this smaller category White sometimes needs 16 moves to force checkmate, it is logically necessary to extend the sequence of BDDs to 16 moves in the original category. At that point all the side conditions are satisfied and the stability theorem can be proved:

$$\begin{array}{l}\vdash_{\mathsf{HOL+BDD}}\\ \quad [\cdots] \Rightarrow \\ \quad p \in \mathsf{win1\ chess} \iff p \in \mathsf{win1\_by\ chess}\ 16\ .\end{array}$$

The final step is to prove that the official set of winning positions found after 16 moves is equal to the set of winning positions found after six moves, and thus conclude that the same stability theorem must also hold for six moves:

$$\begin{array}{l}\vdash_{\mathsf{HOL+BDD}}\\ \quad [\cdots] \Rightarrow \\ \quad p \in \mathsf{win1\ chess} \iff p \in \mathsf{win1\_by\ chess}\ 6\ .\end{array}$$

4 Results

The construction of verified endgame databases described in the previous section is implemented in the HOL4 theorem prover,[3] using the `HolBddLib` [5] interface to the BuDDy BDD engine.[4]

One thing that can make a big difference to the performance of a BDD calculation is the ordering of the boolean variables. Recall from Section 3.1 that the 'state' to be encoded as boolean variables is a list of squares on the board. This is exactly how the state breaks down into boolean variables B:

$$\begin{array}{rcl} \text{State} & \longleftarrow & \text{Square} \cdots \text{Square} \\ \text{Square} & \longleftarrow & \text{File Rank} \\ \text{File} & \longleftarrow & B\ B\ B \\ \text{Rank} & \longleftarrow & B\ B\ B \end{array}$$

To test the effect of variable ordering on performance the construction of the King and Rook versus King and Rook endgame database is used as a benchmark.[5] If the variables are ordered exactly as above then the endgame database takes 1,514 seconds to construct, and the BDD engine creates 165,847,971 nodes. If instead the variables for the state are formed by interleaving the variables for each square, then the endgame database takes 543 seconds to construct, and the BDD engine produces 16,413,512 nodes. Finally, if additionally the variables for each square are formed by interleaving the file and rank variables, the endgame database takes 835 seconds to construct and the BDD engine produces 84,019,830 nodes. Clearly the middle option is best, and this has since been confirmed on other benchmark tests.

Another BDD optimization that proved effective was to combine the quantification and logical connective that occurs when the move relation is applied to the current winning set. On a benchmark test of constructing all four piece endgames containing only Kings, Rooks and Knights, the time required fell 19% from 3,251 seconds and 222,122,342 nodes produced to 2,640 seconds and 144,441,858 nodes produced.

The final results for all four piece pawnless endgames are shown in Table 1. The first column shows the pieces on the board: first the White pieces using the standard abbreviations of K for King, Q for Queen, R for Rook, B for Bishop and N for Knight; next an underscore; and finally the Black pieces. The other columns are separated into positions with White to move and positions with Black to move. Within each, the columns are as follows: **max** column shows the maximum number of moves required for White to force checkmate from a winning position, or a dash if there are no positions winning for White; the **%win** column shows the percentage of legal positions that are winning for White, a dash if there are none, or 'ALL' if every legal position is winning for White;

[3] HOL4 is available for download at `http://hol.sf.net/`.

[4] BuDDy is available for download at `http://sourceforge.net/projects/buddy`.

[5] All the results were collected on a Pentium 4 3.2GHz processor with 1Gb of main memory and running the HOL4 theorem prover using Moscow ML 2.01.

the **#win** column shows the total number of positions winning for White; the **bdd** column shows the compression ratio of the number of BDD nodes required to store the winning sets divided by the total number of winning positions; the **#legal** shows the the total number of legal positions; and the final **bdd** column shows the BDD compression ratio for the legal positions.

Pieces	**White to move**						**Black to move**					
	max	**%win**	**#win**	**bdd**	**#legal**	**bdd**	**max**	**%win**	**#win**	**bdd**	**#legal**	**bdd**
K_K	—	—	0	0%	3612	1%	—	—	0	0%	3612	1%
K_KB	—	—	0	0%	223944	0%	—	—	0	0%	193284	0%
K_KBB	—	—	0	0%	13660584	0%	—	—	0	0%	10164056	0%
K_KBN	—	—	0	0%	13660584	0%	—	—	0	0%	10875504	0%
K_KN	—	—	0	0%	223944	0%	—	—	0	0%	205496	0%
K_KNN	—	—	0	0%	13660584	0%	—	—	0	0%	11499304	0%
K_KQ	—	—	0	0%	223944	0%	—	—	0	0%	144508	1%
K_KQB	—	—	0	0%	13660584	0%	—	—	0	0%	7698432	0%
K_KQN	—	—	0	0%	13660584	0%	—	—	0	0%	8245296	0%
K_KQQ	—	—	0	0%	13660584	0%	—	—	0	0%	5657120	0%
K_KQR	—	—	0	0%	13660584	0%	—	—	0	0%	6911296	0%
K_KR	—	—	0	0%	223944	0%	—	—	0	0%	175168	0%
K_KRB	—	—	0	0%	13660584	0%	—	—	0	0%	9366840	0%
K_KRN	—	—	0	0%	13660584	0%	—	—	0	0%	9905048	0%
K_KRR	—	—	0	0%	13660584	0%	—	—	0	0%	8325184	0%
KB_K	—	—	0	0%	193284	0%	—	—	0	0%	223944	0%
KB_KB	1	0%	416	0%	11832464	0%	0	0%	112	0%	11832464	0%
KB_KN	1	0%	16	0%	11832464	0%	0	0%	8	0%	12535256	0%
KB_KQ	—	—	0	0%	11832464	0%	—	—	0	0%	8952608	0%
KB_KR	—	—	0	0%	11832464	0%	—	—	0	0%	10780728	0%
KBB_K	19	49%	5007216	12%	10164056	0%	19	41%	5628080	8%	13660584	0%
KBN_K	33	100%	10822184	30%	10875504	0%	33	82%	11188168	19%	13660584	0%
KN_K	—	—	0	0%	205496	0%	—	—	0	0%	223944	0%
KN_KB	1	0%	40	0%	12535256	0%	0	0%	8	0%	11832464	0%
KN_KN	1	0%	40	0%	12535256	0%	0	0%	8	0%	12535256	0%
KN_KQ	—	—	0	0%	12535256	0%	—	—	0	0%	8952608	0%
KN_KR	1	0%	32	0%	12535256	0%	0	0%	8	0%	10780728	0%
KNN_K	1	0%	1232	0%	11499304	0%	0	0%	240	0%	13660584	0%
KQ_K	10	ALL	144508	19%	144508	1%	10	90%	200896	12%	223944	0%
KQ_KB	17	100%	8925252	19%	8952608	0%	17	77%	9097332	18%	11832464	0%
KQ_KN	21	99%	8894128	21%	8952608	0%	21	80%	10088688	21%	12535256	0%
KQ_KQ	13	42%	3737092	11%	8952608	0%	12	0%	40628	1%	8952608	0%
KQ_KR	35	99%	8863768	52%	8952608	0%	35	66%	7062680	35%	10780728	0%
KQB_K	8	ALL	7698432	11%	7698432	0%	10	91%	12379568	8%	13660584	0%
KQN_K	9	ALL	8245296	9%	8245296	0%	10	90%	12343856	7%	13660584	0%
KQQ_K	4	ALL	5657120	4%	5657120	0%	10	98%	13378232	6%	13660584	0%
KQR_K	6	ALL	6911296	4%	6911296	0%	16	99%	13519192	6%	13660584	0%
KR_K	16	ALL	175168	20%	175168	0%	16	90%	201700	18%	223944	0%
KR_KB	29	35%	3787160	11%	10780728	0%	29	3%	381888	5%	11832464	0%
KR_KN	40	48%	5210920	34%	10780728	0%	40	11%	1364800	23%	12535256	0%
KR_KQ	19	29%	3090088	5%	10780728	0%	18	0%	17136	0%	8952608	0%
KR_KR	19	29%	3139232	5%	10780728	0%	19	1%	72464	1%	10780728	0%
KRB_K	16	ALL	9366840	12%	9366840	0%	16	91%	12458920	10%	13660584	0%
KRN_K	16	ALL	9905048	11%	9905048	0%	16	91%	12406892	10%	13660584	0%
KRR_K	7	ALL	8325184	3%	8325184	0%	16	100%	13621424	6%	13660584	0%
	40	**29%**	**1.179E8**	**6%**	**4.033E8**	**0%**	**40**	**34%**	**1.355E8**	**6%**	**4.033E8**	**0%**

Table 1. Results for all four piece pawnless endgames.

Constructing the whole endgame database took 18,540 seconds (including 418 seconds spent on garbage collection), during which the HOL4 theorem prover executed 82,713,188 primitive inference steps in its logical kernel and the BDD engine produced 882,827,905 nodes.

5 Conclusions

This paper has shown how a theorem prover equipped with a BDD engine can be used to construct an endgame database that is formally verified to logically follow from the laws of chess.

The method has been implemented for all four piece pawnless positions, and the resulting endgame database can be used as a 'golden reference' for other implementors of endgame databases to check against. In addition, the verified endgame database has been used to produce a set of educational web pages showing the best line of defence in each position category.[6]

The approach used to augment standard theorem proving techniques with a tailor made BDD algorithm was found to be convenient for this application, combining the expressive power and high assurance of theorem provers with the compact representation and fast calculation of BDD engines. As seen in Section 3.2, the use of a theorem prover avoided some potential pitfalls that appear when symbolically processing sets of positions.

6 Related Work

The earliest example of applying BDDs to analyze a two player game is the attempt of Baldumus et. al. [1] to solve American Checkers by means of symbolic model checking.

Edelkamp [3] put forward the idea that BDDs are generally suitable for classifying positions in a wide range of two player games, including chess endgames. Edelkamp's encoding of chess positions also includes a bit for the side to move, but otherwise it is identical to the encoding in this paper. This paper can be seen as a continuation of Edelkamp's work, with the addition of a theorem prover to ensure the accuracy of the move encodings and winning sets.

Kristensen [8] investigated the use of BDDs to compress endgame databases, showing BDDs to be comparable to the state of the art in explicit enumeration for 3–4 man endgames, and better for some simple 5 man endgames.

Acknowledgements

This work was initiated by a conversation between the author and Tim Leonard, and the correct method of boolean encoding was hammered out during many discussions with Konrad Slind.

[6] Available at `http://www.anadune.com/chess/endgames.html`.

References

1. Michael Baldamus, Klaus Schneider, Michael Wenz, and Roberto Ziller. Can American Checkers be solved by means of symbolic model checking? In Howard Bowman, editor, *Formal Methods Elsewhere (a Satellite Workshop of FORTE-PSTV-2000 devoted to applications of formal methods to areas other than communication protocols and software engineering)*, volume 43 of *Electronic Notes in Theoretical Computer Science*. Elsevier, May 2001.
2. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
3. Stefan Edelkamp. Symbolic exploration in two-player games: Preliminary results. In *The International Conference on AI Planning & Scheduling (AIPS), Workshop on Model Checking*, pages 40–48, Toulouse, France, 2002.
4. FIDE. *The FIDE Handbook*, chapter E.I. The Laws of Chess. FIDE, 2004. Available for download from the FIDE website.
5. Michael J. C. Gordon. Reachability programming in HOL98 using BDDs. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference: TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 179–196, Portland, OR, USA, August 2000. Springer.
6. E. A. Heinz. Endgame databases and efficient index schemes. *ICCA Journal*, 22(1):22–32, March 1999.
7. David Hooper and Kenneth Whyld. *The Oxford Companion to Chess*. Oxford University Press, 2nd edition, September 1992.
8. Jesper Torp Kristensen. Generation and compression of endgame tables in chess with fast random access using OBDDs. Master's thesis, University of Aarhus, Department of Computer Science, February 2005.
9. E. V. Nalimov, G. McC. Haworth, and E. A. Heinz. Space-efficient indexing of chess endgame tables. *ICGA Journal*, 23(3):148–162, September 2000.
10. John Nunn. *Secrets of Rook Endings*. Gambit Publications, December 1999.
11. Konrad Slind and Joe Hurd. Applications of polytypism in theorem proving. In David Basin and Burkhart Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 103–119, Rome, Italy, September 2003. Springer.

A Formalized Chess

```
pushes $p$ $sq$ $\equiv$
  board $\cap$ clear_line $p$ $sq$ $\cap$
  (case on_square $p$ $sq$ of
    NONE $\to \emptyset$
    | SOME (_, King) $\to \{sq' \mid$ king_attacks $sq$ $sq'\}$
    | SOME (_, Queen) $\to \{sq' \mid$ queen_attacks $sq$ $sq'\}$
    | SOME (_, Rook) $\to \{sq' \mid$ rook_attacks $sq$ $sq'\}$
    | SOME (_, Bishop) $\to \{sq' \mid$ bishop_attacks $sq$ $sq'\}$
    | SOME (_, Knight) $\to \{sq' \mid$ knight_attacks $sq$ $sq'\}$) ;

sorties $p$ $sq$ $\equiv \{sq' \mid sq' \in$ pushes $p$ $sq$ $\land$ empty $p$ $sq'\}$ ;

captures $p$ $sq$ $\equiv \{sq' \mid sq' \in$ attacks $p$ $sq$ $\land$ occupies $p$ (opponent (to_move $p$)) $sq'\}$ ;

simple_move $p$ $p'$ $\equiv$
  $\exists sq_1, sq_2.$
    occupies $p$ (to_move $p$) $sq_1 \land sq_2 \in$ sorties $p$ $sq_1 \land$
    $\forall sq.$
      on_square $p'$ $sq$ $=$
      if $sq = sq_1$ then NONE
      else if $sq = sq_2$ then on_square $p$ $sq_1$
      else on_square $p$ $sq$ ;

capture_move $p$ $p'$ $\equiv$
  $\exists sq_1, sq_2.$
    occupies $p$ (to_move $p$) $sq_1 \land sq_2 \in$ captures $p$ $sq_1 \land$
    $\forall sq.$
      on_square $p'$ $sq$ $=$
      if $sq = sq_1$ then NONE
      else if $sq = sq_2$ then on_square $p$ $sq_1$
      else on_square $p$ $sq$ ;

chess_move $p$ $p'$ $\equiv$
  chess_legal $p$ $\land$ chess_legal $p'$ $\land$
  (to_move $p'$ $=$ opponent (to_move $p$)) $\land$
  simple_move $p$ $p'$ $\lor$ capture_move $p$ $p'$ ;

chess_move1 $p$ $p'$ $=$ chess_move $p$ $p'$ $\land$ (to_move $p$ $=$ White) ;

chess_move2 $p$ $p'$ $=$ chess_move $p$ $p'$ $\land$ (to_move $p$ $=$ Black) ;

has_pieces $p$ $s$ $l$ $\equiv$
$\exists f \in$ Bijection $\{n \mid n <$ length $l\}$ $\{sq \mid$ occupies $p$ $s$ $sq\}$.
  $\forall n.$ $n <$ length $l \Rightarrow$ (on_square $p$ $(f\ n) =$ SOME $(s,$ nth $n$ $l)$) .
```

Exploring New OO-Paradigms with HOL: Aspects and Collaborations

Florian Kammüller

Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik

Abstract. In this paper we report on previous, current and ongoing work on the analysis of collaboration-based and aspect-oriented programming languages with mechanizations in Isabelle/HOL and Coq.

1 Introduction

Triggered by the ever increasing ubiquity of software the demand for more flexibility in the assembly of programming components is being felt more and more. Programs shall be run on small devices like handhelds, on dedicated operating systems, for example JavaCard for SmartCards. Moreover, application must be loadable on demand as resources in small devices are restricted. The concept of object is too fine grained to encapsulate entire applications — the obvious concept therefore is a module, or component as it was initially called. However, it turns out that the assembly of components is still somewhat too restricted. Components have no state, need the additional concept of deployment in different contexts and hence differ in instantiations which blurs the initially clean concept by creating multiple identities. Hence, a migration of already deployed components over the limits of execution environments is difficult.

Collaboration-based programming languages introduce a concept for modules for classes on top of the usual object-oriented language features. The basic idea is similar to packages, or components, but goes beyond that in that these modules can be instantiated. Thereby, we can define whole groups of interacting objects on the abstract, the class level. The defined groups can then be instantiated as a whole to build groups of interacting objects.

Consequently, there was a need to devise more flexible notions one structural level above object and class that would enhance the aforementioned mechanisms of dynamic loading and grouping. The new paradigms for object orientation that have been designed for this purpose are called mixins, traits or object teams. We call them unifyingly collaborations.

Besides this concept, that seems to be ideal for the development of large systems, there is the concept of aspect-orientation that introduces new concepts into object-orientation. Aspect-orientation enables the definition of small pieces of code, so-called aspects, that can then be distributed at certain points throughout an existing application. For example, a login could for security reasons be extended by a password check. Assuming the login is a feature that is used in various parts of the code, one could now weave this aspect of password check into that code before each code segment that contains the login. Aspect-oriented languages comprise features for defining the aspects themselves and features for the definition of the points at which the aspects have to be woven into the code, so called join-points. An example for an aspect-oriented language is AspectJ [1], an extension of Java by aspect-oriented features.

A collaboration may as well be used as a wrapper, and a wrapper realizes an adaptation of behaviour, which is in turn nothing else than applying an aspect. The borderline between the concepts aspects and collaboration seems to be a bit vague.

Therefore, we think that a combination of collaborations and aspect-orientation is a promising direction for programming languages. In order to guarantee that such combined languages catch on, it is necessary to support them with the necessary theoretical foundation. Similar to the efforts to give formal models for Java that can be mechanically verified, we are working on the development of a formal, mechanically supported framework for collaboration-based, aspect-oriented languages in Higher Order Logic.

A formalization of the collaboration-based language Object-Teams in Isabelle/HOL, is a first example of exploring collaborations in HOL. It aimed at proving type-safety, and succeeded in proving confinement. It followed quite strongly the outline of the formalization of Java [11]. It turned out to be very complex, obscuring the most essential language features by being so close to an actual instance of a language with a rich set of features.

Therefore, we continued our work on a more general level. When considering a formal analysis of aspect-orientation we decided to work independently of a specific language, being just inspired by features as they are common in aspect-oriented languages, most prominently AspectJ – but not being restricted in our progress by specific language design decisions. We mechanized this model in Coq [3]. The reasons for this switch are that, first we could base the formalization of aspects on a fairly well-established model of bytecode [2]; second we are planning to do the type safety analysis by translating into League's [10] intermediate language that is also mechanized in Coq.

We give an outline of this ongoing research in the paper: After a brief introduction into the ideas of collaborations and aspects in the remainder of this section, we summarize in Section 2 the formalization that has been done in Isabelle/HOL. We then present the work on aspects in Section 3. Finally in Section 4 we give an account of related work and our future plans.

1.1 Aspects and Collaborations

In this section we give a brief account on the principle concepts of aspect-oriented and collaboration-based programming languages.

Aspect-Orientation

The main idea of aspect-orientation is to adjust programs after their creation by *weaving* in at certain points (*point cuts*) some more program code (*advice*). The procedure of weaving is illustrated in Figure 1. The main concepts of aspect-orientation, i.e. advices, weaving and join-points and the related point-cuts, are summarized as follows.

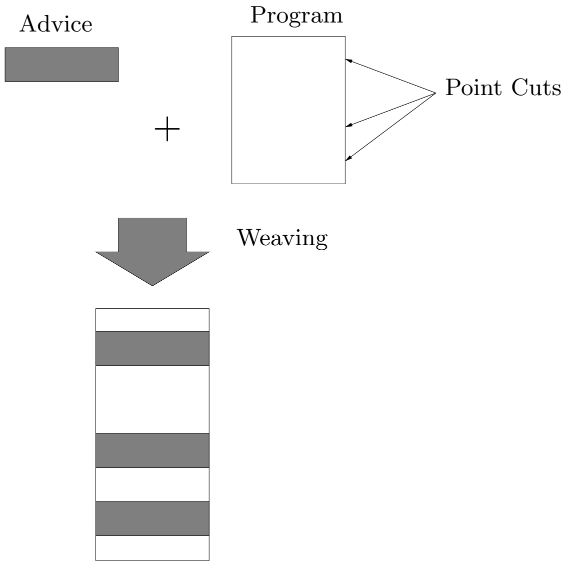


Fig. 1. Weaving advice at point-cuts

- advice is the code that has to be woven into the original program
- point cuts are the points in the program where aspects are woven in
- join-points are sets of point cuts usually described by some predicates using special keys like `call` to refer to all points of method invocation, and usual logical connectives.
- aspect-oriented language: the basic language is usually an object-oriented language, to express programs and advices
- join-point definition language: the language to express logical operators, quantification over program points.

Aspect-oriented program development usually starts by identifying so-called cross-cutting concerns. Starting from a usual object-oriented implementation in a standard way, the cross cutting concerns are then woven into it as advices.

Aspects: Observations When considering the process of weaving we have two ways to proceed:

- compile-time weaving: weave the advice in at the source code level, then compile,
- or run-time-weaving: translate program and advice (and join-points) separately and weave in at the execution level.

We see immediately that run-time weaving is more desirable but clearly also trickier. Compile-time weaving is fairly simple. For example, declarative programming languages, for example Prolog, may be considered as aspect-oriented language. Although there the distinction of compile-time does not apply, we can easily consider Prolog as a join point definition language for any imperative oo-language. To produce a compile-time weaver, just preprocess the source code using the Prolog predicates, then use the old compiler.

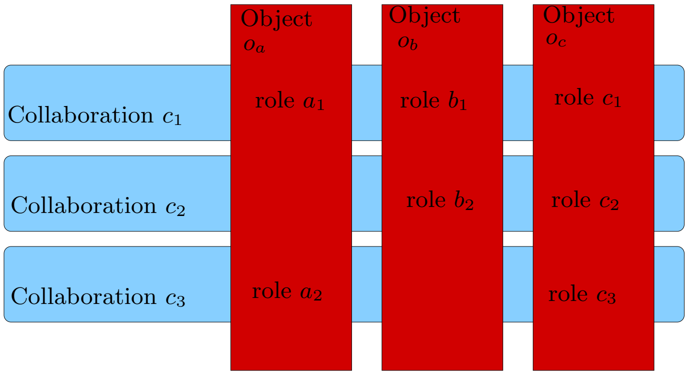
However, at times where global computing is on demand, run-time weaving is needed because we want to be able to adjust executable, deployed, program components by weaving in advice.

Collaboration-Based Programming

The main idea of collaborations is to consider collaborations of objects on the class level. Thereby, different from components, we can consider instantiation — like for usual classes — also for classes of classes, or collaborations. Conceptually, this feature enables the use of the collaboration idea at the source-code level, because classes correspond roughly to types. Therefore, using the collaboration concept we can statically check relationships between objects and can thereby express that

- Objects can be members of different collaborations
- Objects play *roles* in collaborations

If we furthermore adopt the ideas of inheritance for collaborations as well, we can even adapt existing object collaborations statically. To give a simpler intuition that is independent of the concepts of object-oriented languages. The



typical introductory collaboration example is that of hotel, husband, wife, and daughter. If objects of these classes would want to collect at reception the keys for their family rooms in the hotel, a simple identification of the kind "I'm a daughter" does not suffice. Even being member of a super-class family-member does not suffice. Generally, a member object of a *group* needs to know to which instance of the group it belongs. As collaborations are considered on the class-level, i.e. the type-level, this membership creates a type dependency, i.e. the type of the group depends on instances of its constituting member classes.

2 ObjectTeams in Isabelle/HOL

The programming model of Object Teams [7] is an example of a collaboration-based language. The developers of Object Teams are currently working on extending the language to integrate aspect-orientation. For a theoretical idea on how this may be achieved see Section 4. The current version of Object Teams is called ObjectTeams/Java, because it uses Java as a host language, i.e. is tranlated into Java. But, in principle, any other class based-language could be used as host language. This language is well supported and is already used in practical applications in projects with industrial collaboration [6]. The module concept is called team, which is a container for so-called roles. The containment is on the level of classes and instances. Besides the classical inheritance that exists between classes the introduction of the concept of teams enforces a second implicit inheritance, as roles contained in teams inherit from each other, when a team inherits from another team. Teams thereby realize the concept of virtual classes with overriding. Virtual classes enable the refinement of sets of mutually-recursive types. Resulting issues of static type checking are handled by instance based types, a special kind of dependent types. By the combination of implicit inheritance and instance based types, teams realize the concept of family polymorphism [4]. On top of the aforementioned concepts, ObjectTeams/Java allows for different levels of role confinement providing for an ownership-like alias control [16]. Except for some flavors of role confinement these concepts have been formalized and the results of their analysis will be presented next.

2.1 Isabelle/HOL model of ObjectTeams/Java

The master's thesis [14] provides a formalization of the main concepts of the language ObjectTeams/Java. This extensive experiment is strongly based on the works of Nipkow, von Oheimb et al [13, 11]. However, as we introduce completely new concepts, we had to redefine and reprove basically everything. We define abstract syntax and type system. The inheritance relation with its particular extension of implicit inheritance is defined. Well-structuredness of classes and the inheritance relation is proved. The type model is divided into static and dynamic instance based types. As role classes are contained in team classes, their instances have the same containment. Hence a roles type does depend on its instance context. That is what we call instance-based typing. In our model we do not use dependent types explicitly, rather model them by extending the types in the static part by a fixed constant `TTHis` that is then replaced in the dynamic typing by the actual instance context. Thereby we can separate the analysis into static and dynamic part. In order to define well-formedness predicates for the type-safety we define typing rules using inductive definitions. An operational semantics is modelled using inductive definitions as well. Finally, we have partly proved the type soundness theorem. However, we did not succeed as the formal analysis already produced a fundamental problem of the language: as ObjectTeams allow the instantiation of abstract role classes, type checking a sub team class requires more information than type checking a conventional subclass. Besides a super team classs signature, a list of so-called relevant role classes is necessary for the judgment of well-formedness of the sub team class. Relevant role classes are those role classes that a sub team must implement in order to be concrete. This feature has been discovered during the formalization work and is now integrated into the language. However, it prevented us from completing the mechanical proof of type soundness. Assuming for the time being the type soundness as axiom we could, however, prove the following confinement statement.

```
[| G |- (x,(h,l)) - t >-> (v,x,(h,l));
  wf_prog G; conf (h,l) (G,L); (G,L) |= t :: T |]
  ==> (x’= None ⟶ role_referenced_in_context G (h,l) v)
```

The theorem expresses that role objects are only referenced within the context of their enclosing team instance. The proviso has to be read as: a term t evaluates with respect to program `G`, producing a value `v`, transforming the state of heap and local environment `(h,l)` into `(h’,l’)`. Furthermore, it is assumed that the program is wellformed, that the before-state `(h,l)` conforms to the static environment `(G,L)`, that the term `t` is well-typed, and that no exception occurred `x’=None`.

3 Coq Aspects

In this section we present the mechanization of aspects in Coq. The main goal of this formalization is to supply an axiomatic framework that helps to identify conditions for aspects oriented languages to work. The major interest is to realize run-time weaving. That is, it should be able to weave in advice at run time.

3.1 Basis: OO-language

We use a formalization of the main concepts of object-orientation given in [2] – an axiomatic framework for non-interference. Although originally designed for proving and deriving a bytecode-verifier it does also provide a mechanization of the operational semantics of the Java bytecode language. Although we want to treat aspects and collaborations on an abstract level, we think that using the Java bytecode as the execution layer is not such a strong restriction of generality because (a) our model treats basic features that are quite common for most object-oriented languages (b) most object-oriented languages can be translated (and are in fact translated) to the Java bytecode in order to gain wider applicability. For the latter point even C++ offers a translation to Java bytecode.

We give a very short outline of the relevant properties of this basic Coq model leaving out the details that refer to security problems. The bytecode instruction set we consider is formalized using the following inductive definition.

```
Inductive instr : Set :=
  nop  : instr
| push : value -> instr
| iadd : instrin
| load : locs -> instr
| store: locs -> instr
| goto : ppt -> instr
| ifthenelse   : ppt -> instr
| new: Class -> instr
| getfield: Field -> instr
| putfield: Field -> instr
| invoke: method -> instr
| retrn: instr.
```

The set `ppt` is a decidable set of program points and `locs` are the register locations. We assume a program `p` to be a map from program points and method names to instructions. Based on this instruction set the operational semantics is defined introducing operand stacks, object heaps, values as structured types containing pointers and simple values, and register valuations. A state in the semantics is constituted by a stack of method frames. Each of those frames comprises the current program counter, a current variable binding, an operand stack, and a security environment. Both the variable binding and the security environment are treated abstractly with lookup and update functions.

```
Record frame: Set:= {pc: pcs; rm: env ; os: stack}.
```

States are composed as stacks of frames and a heap.

```
Record state: Set:= {fs: stack(method × frame); hp: heap }.
```

In the operational semantics we define the execution of a program step-by-step over program states by a simple case analysis over the type of instructions and assigning the corresponding effect on the state. A general n-step execution `exec` is then defined inductively over this one step execution function.

Now in order to be able to reason about source code we have to build a second instruction layer of source code instructions.

```
Inductive sc_instr: Set :=
  assign : var -> value -> sc_instr
|  add : var -> var -> sc_instr
| ifte   : (var -> bool) -> ppt -> ppt -> sc_instr
| New: Class -> sc_instr
| Getfield: Field -> sc_instr
| Putfield: Field -> sc_instr
| Invoke: method -> sc_instr
| Retrn: sc_instr.
```

and correspondingly develop the notion of state and execution similar to the bytecode level. Compilation between the two levels will be considered next, but first we model join-points.

3.2 Call Join-Points and Weaving

The most frequently used join-point constructor used in aspect-orientation is the `call` construct. With the `call` operator we can construct a predicate that selects program points that contain a method invocation. As arguments to the `call` operator we can, for example in AspectJ, use so-called *wildcards*, annotated by `*`, to create some kind of pattern matching over method names. In the formalization we can easily be more general than this by just allowing any predicate over method names, type `method`, as admissible input to the `call_sc` constructor for call join points at the source level.

```
Definition call_sc : (method -> Prop) -> join_point :=
   (fun mp: (method -> Prop) => fun pc: ppt =>
          match p_sc pc with Invoke m => True
              |  _ => False
          end).
```

The type `join_point` is just an abbreviation for `ppt -> Prop`, i.e predicates over program points.

Compilation of a source program replaces instructions by creating a new binding of program points to bytecode-level instruction. The call join-point constructor is translated as follows.

```
Definition call_bc : (method -> Prop) -> join_point :=
   (fun mp: (method -> Prop) => fun pc: ppt =>
          match p_bc pc with invoke m => True
              |  _ => False
          end).
```

As weaving can be performed as an additional step of compilation, it can as well be seen just on the syntactic representation of programs. In order to analyze the requirements of a run-time weaving we start from assumptions about the compilation process from source to bytecode level.

Assuming a compilation function mapping source code programs `program_sc` to bytecode-programs `program_bc`, we can reason about the weaving process in order to gain precise information about the inherent requirements of aspects.

```
Parameter comp: program_sc -> program_bc.
```

We assume in a first step only non-optimizing compilers, i.e. we assume that each source code command is one to one translated into a sequence of bytecode-level instructions. That is, we assume that the compilation is injective. Hence, we can assume an inverse function.

```
Parameter compinv: program_bc -> program_sc.
Axiom compinj: ∀ psc: program_sc, compinv(comp psc) = psc.
```

Weaving simulation Based on the model of a compiler function, we can now consider the property that represents the question set out in the beginning (cf. 1) whether run-time weaving is possible. More precisely, we try to identify the conditions such that the diagram in Figure 2 commutes. The two weaving functions are represented by the types

$$\begin{tikzcd}
(p_{sc}, jp_{sc}, adv_{sc}) \arrow[r, "\texttt{weave\_sc}"] \arrow[d, "\langle \texttt{comp}{,} jp_{comp}{,} \texttt{comp} \rangle"] & p'_{sc} \arrow[d, "\texttt{comp}"] \\
(p_{bc}, jp_{bc}, adv_{bc}) \arrow[r, "\texttt{texec}"] & p'_{bc}
\end{tikzcd}$$

Fig. 2. Do compile-time and run-time weaving commute?

```
Parameter wv_sc: program_sc -> join_point -> advice_sc -> program_sc.
Parameter wv_bc: program_bc -> join_point -> advice_bc -> program_bc.
```

In order to narrow down the possible specification of these functions we identified the following assumption.

```
Axiom step1: ∀ (pbc: program_bc)(jp: method -> Prop)(adbc: advice_bc),
∀ wpbc: program_bc,
  wpbc = (wv_bc pbc (call_bc jp) adbc) →
  (exists psc: program_sc,
    comp psc  = wpbc ∧ psc = wv_sc (compinv pbc)(call_sc jp)(compinv adbc)).
```

This property states that a bytecode-weave does only produce programs that can also be created as source-code. With this property we can prove the commutation property.

```
Lemma run_time_weave: ∀ mp: (method -> Prop),
  ∀ a_sc: advice_sc, ∀ a_bc: advice_bc,
  comp a_sc = a_bc →
  comp (wv_sc p_sc (call_sc mp) a_sc) =
  wv_bc (comp p_sc)(call_bc mp)(a_bc).
```

Although we can under reasonable assumptions prove the correctness of run-time weaving here, this is only restricted to the standard `call` join-points.

3.3 Logical connectives and Control Flow

The definition of join points offers in general further possibilities beyond the call construct. Mixed join-point expressions can be introduced using logical connectives. The following rule is an instance of the major property shown in the previous section but for the disjunction of join-point expressions.

```
Axiom rt_disj: ∀ jp1 jp2: join_point,
  ∀ a_sc: advice_sc, forall a_bc: advice_bc,
  comp a_sc = a_bc ->
   comp (wv_sc p_sc (jp_disj jp1  jp2) a_sc) =
   wv_bc (comp p_sc)(jp_disj jp1  jp2) a_bc.
```

In order to deal with such combined weaving processes a decomposition may be achieved using the following lemma to iterate the construction.

```
∀ jp1 jp2: join_point,
  ∀ a_sc: advice_sc, forall a_bc: advice_bc,
wv_sc p_sc (jp_disj jp1 jp2) a_sc = wv_sc (wv_sc p_sc jp1 a_sc) jp2 a_sc.
```

In the example shown we illustrate disjunction. The decomposition does not work like this for conjunction of join point expressions. Here, further thought is needed.

Another possibility for constructing join-point expressions is the selection of join-points by selecting a control flow. Here the cflow(cl, mn) enables to select program points from the beginning to the end of method execution of method mn of class cl. For the analysis of weaving advice according to join-points selected in this manner the contemplation at the execution level, i.e. according to program behaviour, seems more appropriate.

3.4 Weaving equivalence based on Behaviour

The properties related so far are all based on the idea to show the correctness of run-time weaving at the syntactic level. For example, using the execution function, provided in the operational semantics [2] we can prove,

```
Axiom rt_conform: ∀ jp1 jp2: join_point,
  ∀ a_sc: advice_sc, forall ,
   exec (comp (wv_sc p_sc (call_sc mp) a_sc) =
   exec (wv_bc (comp p_sc)(call_bc mp)(comp a_sc)).
```

by just substituting with lemma run_time_weave. Hence, the condition we have derived is clearly sufficient, for call, but probably a bit strong in general. Possibly, when translating onto different versions of bytecode, it might be that we cannot rely on the conformance relations between compilers and program points. Optimizing compilers could also destroy the injectivity assumptions.

Here, we have to show property rt_conform directly over the operational semantics not syntactical equivalences.

4 Related Work and Outlook

4.1 Intermediate Language LITL

As mentioned before League and Monnier devise an intermediate language called LITL that aims at type preserving compilation of class-based languages. LITL is based on the intermediate language Links [5]. Similar to this predecessor the intermediate language LITL enables to compile various flavours of class concepts, including collaborations (there called traits or mixins). Moreover, in contrast to Links, LITL is typed, as it is embedded into Coq. The typing of LITL is the major contribution of this work. Although the authors have already achieved a typing of LITL the second point – the translation of collaborations onto LITL, and thus the typing of collaborations — is not yet finished.

For our project LITL is a well-suited target language, as we can experiment with language features and — given the translation works — can see whether typability by typed compilation onto LITL is preserved. However, as LITL does not support aspects directly, we have to find a way to either integrate aspects into collaborations or find out what are the fundamental differences that could clarify the differences between the concepts. For the former possibility we consider a way to integrate aspects and collaborations.

4.2 Aspects as Collaborations

The question that we address in this subsection is what have aspects to do with collaborations. We present the conceptual idea how to use collaborations to represent aspects. This method can in principle be used in our mechanization to create a unified framework for aspects and collaborations. As we see in Figure 3 collaborations act as wrappers. Using the possibility of method redefinition in

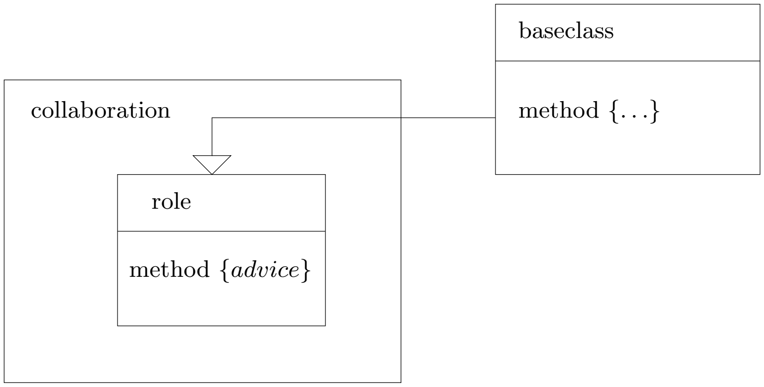


Fig. 3. Aspects represented by collaborations

a collaboration the role class adjusts method `method`. That is, we realize aspects by a navigation between role classes and their base classes. Instead of using method over-riding as in the example, we could use so-called callin and callout constructors, as they exist for example in Object-Teams, to adjust the base class method.

Although in principle, this mimics the behaviour of advice weaving, it leaves open how to define such adjustments of existing behaviour in a controlled way similar to the concepts of join-points and weaving. The development of join-point languages for collaborations, in order to specify and quantify the locations for the advice application is currently under research with the language developers. Nevertheless, we think about offering solutions on the level of mechanized specification.

4.3 Discussion

We have introduced the notions of collaborations and aspects and described in outline a mechanization of the collaboration concept in Isabelle/HOL as well as an axiomatic approach to aspects in Coq.

The mechanization of collaborations at the example of Object Teams showed up difficulties with respect to type safety and enabled the proof of a confinement property, given type safety. The abstract formalization of aspects yielded properties that are a suitable frame for the analysis of concrete weaving functions with respect to facilities for the definition of join points and the crucial question of run-time weaving.

The translation of the formalization of ObjectTeams into Coq will enable us to experiment with the mechanical analysis of the integration of Aspects and Collaborations as sketched in the previous section.

We aim at providing a mechanized logical framework that enables the experimentation with language features, like exception handling, dynamic class loading, method overriding, in order to see if typability and security features are violated. This is precisely the kind of tool that is needed for the support of the language development: a workbench that enables to test the implication of the redefinition of generalization of a language concept. The challenge for this project is whether we will – beyond analyzing the principal language properties of a restricted sublanguage – be able to answer this need.

References

1. ASpectJ – Java with Aspects. www.eclipse.org/aspectj.
2. G. Barthe and F. Kammüller. Certified Bytecode Verifier for Noninterference. Technical Report, INRIA Sophia-Antipolis, 2005. In print.
3. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
4. E. Ernst. Family polymorphism. In *Springer LNCS 2072*, 2001.
5. K. Fisher, J. Reppy, and J.G. Riecke. A calculus for compiling and linking classes. In Programming Language Design and Implementation, 2000.

6. S. Herrmann. TOPPrax - Aspektorientierte Programmierung für die Praxis. http://www.topprax.de.
7. S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Springer LNCS 2591*, 2003.
8. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java — A Minimal Calculus for Java and GJ. In *Proceedings of OOPSLA'99, ACM SIGPLAN*, 1999.
9. F. Kammüller. Modular structures as dependent types in isabelle. In *Springer LNCS 1657*, 1998.
10. C. League and S. Monnier, Typed Compilation Against Non-Manifest Base Classes. In CASSIS '05, March 2005, Nice. To appear in Springer LNCS.
11. T. Nipkow et al.. Java Source and Bytecode Formalizations in Isabelle: Bali. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/past.html, 20.02.2004.
12. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.
13. D. v. Oheimb. p *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.
14. R. Oeters. *Foundation of ObjectTeams/Java in Isabelle/HOL.* Diplomarbeit. Technische Universität Berlin, 2003.
15. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, Springer LNCS, **828**, 1994.
16. J. Vitek and B. Bokowski. Confined types in Java. *Software, Practice and Experience*, 31(6):507-532, 2001.

Formalization of Hensel's Lemma

Hidetune Kobayashi[1], Hideo Suzuki[2], and Yoko Ono[3]

[1] Nihon University hikoba@math.cst.nihon-u.ac.jp
[2] Polytechnic University hsuzuki@tokyo-pc.ac.jp
[3] The University of Shimane y-ono@u-shimane.ac.jp

Abstract. Formalization of Hensel's lemma in Isabelle/HOL is reported. Polynomial rings and valuations are formalized to express Hensel's lemma. Our thy files Algebra and Valuation1-3 are totally 3.2MB with 58,963 lines and 2,842 lemmas.

1 Introduction

We are formalizing abstract algebra in Isabelle/HOL to make a computer supporting system of mathematical study focused on "algebraic geometry". At present, we have formalized abstract rings, modules(Algebra1-9 in AFP " Groups, Rings and Modules " are revised) and valuations.

Valuations are used to describe points of algebraic curves and algebraic curves are relatively simple examples of algebraic geometry, therefore our formalization of valuations will be useful when we treat algebraic geometry in general.

Hensel's lemma is one of the important lemmas, and there are some approaches to formalize. We formalized it as a lemma derived from the topology defined by the valuation. The topology can be discussed without introducing valuation[1], but the power of the maximal ideal of a valuation ring is equivalent to the normal valuation. Therefore Hensel's lemma in this report is not less general than that in [1].

We note that there is sophisticated formalization of polynomial ring by Ballarlin[2], and one of the authors is also formalizing valuations in MIZAR with three Polish experts' help. As far as we checked, there is formalization of elementary concepts of algebra, but we couldn't find formalization covering the rich contents of ours in other formalization languages.

2 Polynomial rings

Hensel's lemma is a lemma concerning factorization of polynomials of one variable. We deal this lemma with abstract ring coefficients. In this section we present formalization of polynomial ring over an abstract commutative ring.

2.1 Fundamental definitions

Polynomial ring R is a ring generated by polynomials with coefficients in a subring S of R having a variable X. To formalize a polynomial ring, we define the following items:

```
pol_coeff::"[('a, 'more) RingType_scheme, nat, nat ⇒ 'a] ⇒ bool"
"pol_coeff S n f == f ∈ Nset n → carrier S"

pol_coeffs::"('a, 'more) RingType_scheme ⇒ (nat ⇒ 'a) set"
"pol_coeffs S == ⋃{X. ∃n. X = Nset n → carrier S}"

coeff_len::"[('a, 'more) RingType_scheme, nat ⇒ 'a] ⇒ nat"
"coeff_len S f == SOME n. f ∈ (Nset n → carrier S)"

coeff_max::"[('a, 'b) RingType_scheme, nat, nat ⇒ 'a] ⇒ nat"
"coeff_max S n f == n_max {j. j ≤ n ∧ f j ≠ 0_S}"

polyn_expr::"[('a, 'more) RingType_scheme, 'a, nat, nat ⇒ 'a]
⇒ 'a"
"polyn_expr R X n f == eSum R (λj. (f j) ·_R (X^Rj)) n"

algfree_cond::"[('a, 'm) RingType_scheme,
('a, 'm1) RingType_scheme, 'a] ⇒ bool"
"algfree_cond R S X == ∀n f. pol_coeff S n f ∧
eSum R (λj. (f j) ·_R (X^Rj)) n = 0_R ⟶ (∀j∈Nset n. f j = 0_R)"

polyn_ring::"[('a, 'm) RingType_scheme,
('a, 'm1) RingType_scheme, 'a] ⇒ bool"
"polyn_ring R S X == algfree_cond R S X ∧ ring R
∧ ¬ zeroring R ∧ Subring R S ∧ X ∈ carrier R
∧ (∀g∈carrier R. ∃f. f ∈ pol_coeffs S
∧ g = eSum R (λj. (f j) ·_R (X^Rj)) (coeff_len S f))"
```

Here, `Nset n → carrier S` is a set of functions from the set {0, 1,. . ., n} to the carrier of a ring S, and `pol_coeff S n f` is a test function whether f is a function from `Nset n` to the `carrier S`, which returns true or false. `eSum R (λj. (f j)` $\cdot_R$ $(X\hat{\ }^{Rj})$`) n` is our formalized expression of a polynomial (f 0) + (f 1)·X + . . . + (f n)·X^n. `algfree_cond R S X` is a condition that there is no algebraic relation between elements of S and monomials generated by X. This condition guarantees that the polynomial expression is unique, that is if (f 0) + (f 1)·X + . . . + (f n)·X^n = (g 0) + (g 1)·X + . . . + (g m)·X^m then (f 0) = (g 0),. . ., (f n) = (g n) and (g (n+1)) = . . . = (g m) = 0 if n < m. A polynomial ring is a ring R, generated by elements having a polynomial expression. Our formalization of a polynomial ring is equivalent to the mathematical expression R = S[X].

2.2 The degree of a polynomial

We can prove two polynomial expressions `polyn_expr R S n f` and `polyn_expr R S m g` satisfying conditions (f n) ≠ 0, (g m) ≠ 0 and polyn_expr R S n f = polyn_expr R S m g then n = m and (f 0) = (g 0), (f 1) = (g 1), . . ., (f n) = (g n). Hence degree is determined to each nonzero element of R. To formalize the degree, at first we define coeff_sol:

```
coeff_sol::"[('a, 'b) RingType_scheme, ('a, 'b1) RingType_scheme,
'a, 'a, nat ⇒ 'a] ⇒ bool"
"coeff_sol R S X g f == f ∈ pol_coeffs S ∧
g = polyn_expr R X (coeff_len S f) f"
```

By definition of the polyn_ring, there is a coefficient function giving a polynomial expression of an element g of R.

```
lemma ex_polyn_expr1:"[|ring R; ring S; polyn_ring R S X;
g ∈ carrier R|] ⟹ ∃f. coeff_sol R S X g f"
by (simp add:coeff_sol_def,
frule ex_polyn_expr[of "R" "S" "X" "g"], assumption+)
```

This implies the following is well defined:

```
deg_n ::"[('a, 'b) RingType_scheme, ('a, 'b1) RingType_scheme,
'a, 'a] ⇒ nat"
"deg_n R S X p == coeff_max S (coeff_len S (SOME f. coeff_sol
R S X p f)) (SOME f. coeff_sol R S X p f)"
```

We introduced a new type ant of augmented integers $\{-\infty\} \cup Z \cup \{\infty\}$, and formalized the degree in general sense as

```
deg ::"[('a, 'b) RingType_scheme, ('a, 'b1) RingType_scheme,
'a, 'a] ⇒ ant"
"deg R S X p == if p = 0_R then -∞ else (an (deg_n R S X p))"
```

Thus we take deg 0 as -∞. We defined -∞ + -∞ = -∞, -∞ + n = -∞.

coeff_sol is, as defined above, a boolean function to test that for a given polynomial g we have a coefficient f satisfying an equation
g = polyn_expr R X (coeff_len S f) f , where coeff_len S f is some natural number such that f ∈ Nset (coeff_len S f) → carrier S.

Of course this is logically trivial, but to use SOME, we need coeff_len S f. coeff_max is defined as n_max {j. j ≤ n ∧ f j ≠ 0_S}, for a coefficient {(f 0), (f 1),. . ., (f n)}.

We have two lemmas concerning the degree of a polynomial.

```
lemma pol_of_deg0:"[|ring R; ring S; polyn_ring R S X;
      p ∈ carrier R; p ≠ 0_R |] ⟹
      (deg_n R S X p = 0) = (p ∈ carrier S)"

lemma pol_of_deg0_1:"[|ring R; ring S; polyn_ring R S X;
      p ∈ carrier R |] ⟹
     (deg R S X p = 0) = (p ∈ carrier S - {0_S})"
```

2.3 A homomorphism of a polynomial ring to a homomorphism ring

A set of homomorphisms from a polynomial ring R = S[X] to a polynomial ring A = B[Y] is defined as

```
polyn_Hom::"[('a, 'm) RingType_scheme, ('a, 'm1) RingType_scheme,
'a, ('b, 'n) RingType_scheme,
('b, 'n1) RingType_scheme, 'b] ⇒ ('a ⇒ 'b) set"
("(pHom _ _ _, _ _ _)" [67,67,67,67,67,68]67)
"pHom R S X, A B Y == {f. f ∈ rHom R A
∧ f`(carrier S) ⊆ carrier B ∧ f X = Y}"
```

and from this definition we can prove a simple lemma:

```
lemma pHom_mem:"[|ring R; ring S; polyn_ring R S X; ring A;
ring B; polyn_ring A B Y; f ∈ pHom R S X, A B Y;
pol_coeff S n c|]
⟹ f (polyn_expr R X n c) = polyn_expr A Y n (cmp f c)"
```

In ordinary mathematical expression, we can write this lemma as

lemma pHom_mem:
Let R *be a polynomial ring* S[X] *and let* A *be a polynomial ring* B[Y].
If f *is a ring homomorphism of* S *to* B, *then* f *is uniquely extended to a homomorphism* F *of* S[X] *to* B[Y] *such that*
F $(\mathrm{a}_0 + a_1X + \ldots + a_n\ \mathrm{X}^n\ ) = (\mathrm{f\ a}_0) + (\mathrm{f\ a}_1)\ \mathrm{Y} + \ldots + (\mathrm{f\ a}_n)\ \mathrm{Y}^n$.

In our formalization we have a complicated expression for the extended homomorphism F above:

```
ext_rH::"[('a, 'm) RingType_scheme, ('a, 'm1) RingType_scheme,
'a, ('b, 'n) RingType_scheme, ('b, 'n1) RingType_scheme,
'b, 'a ⇒ 'b] ⇒ ('a ⇒ 'b)"
"ext_rH R S X A B Y f == λx∈carrier R. (if x = 0_R then 0_A
     else polyn_expr A Y (deg_n R S X x)
    (cmp f (SOME h. d_cf_sol R S X x h)))"
```

`d_cf_sol R S X x h` is a condition

```
d_cf_sol::"[('a, 'b) RingType_scheme, ('a, 'b1) RingType_scheme,
'a, 'a, nat ⇒ 'a] ⇒ bool"
"d_cf_sol R S X p f == pol_coeff S (deg_n R S X p) f ∧
p = polyn_expr R X (deg_n R S X p) f ∧ f (deg_n R S X p) ≠ 0_S"
```

and `ext_rH R S X A B Y f` is the extended homomorphism of the polynomial ring `polyn_ring R S X` to the `polyn_ring A B Y` of a ring homomorphism S to B.

We see the degrees satisfy the inequality:

```
lemma pHom_dec_deg:"[|ring R; ring S; polyn_ring R S X; ring A;
ring B; polyn_ring A B Y; g ∈ pHom R S X, A B Y; p ∈ carrier
R; g p ≠ 0_A|] ⟹ deg_n A B Y (g p) ≤ deg_n R S X p"
```

This lemma means if g is a homomorphism satisfying g (carrier S) $\subseteq$ carrier B and g(X) = Y, then the degrees satisfy above inequality. Here, we note why we use variables X and Y. This is because restriction of types. When we treat residue class ring R/P, the element of R/P has type 'a set if element of R is of type 'a, and in our definition X has type 'a, so we cannot take X as a variable over O/P.

2.4 Relatively prime polynomials

Hensel's lemma is a lemma giving a factorization by using relatively prime polynomials over a residue class field. We formalize the concept of relatively prime polynomials as

```
rel_prime_pols::"[('a, 'm) RingType_scheme,
('b, 'm1) RingType_scheme, 'a, 'a, 'a ] ⇒ bool"
"rel_prime_pols R S X p q == (1_R) ∈ ((Rxa R p) +_R (Rxa R q))"
```

Here p and q are polynomials(i.e. elements of R) in a polynomial ring R = S[X], and `Rxa R p` (later `Rxa R p` is denoted by `R ◇ p`) is an ideal of R generated by p. Therefore, two polynomials p, q are relatively prime if and only if there are two polynomials u, v such that u p + v q = 1_R, where 1_R is the multiplicative unit of R.

The division is formalized as

```
lemma divisionTr4:"[|ring R; field S; polyn_ring R S X;
g ∈ carrier R; g ≠ 0_R; 0 < deg_n R S X g; f ∈ carrier R|] ⟹
∃q∈carrier R. (f = q ·_R g) ∨ (∃r∈carrier R. r ≠ 0_R ∧
(f = (q ·_R g) +_R r) ∧ (deg_n R S X r) < (deg_n R S X g))"
```

We can read this lemma as

Let R be a polynomial ring S[X],and let f, g be polynomials in R such that g is a nonzero polynomial with $0 < \deg g$*.Then, there is a polynomial* $q \in R$ *such that* $f = qg$ *or there is a nonzero polynomial r such that* $f = qg + r$ *with* $\deg r < \deg g$.

We have a lemma

```
lemma rel_prime_equation:"[|ring R; field S; polyn_ring R S X;
f ∈ carrier R; g ∈ carrier R; f ≠ 0_R;
0 < deg_n R S X f; g ≠ 0_R; 0 < deg_n R S X g;
h ∈ carrier R; h ≠ 0_R; rel_prime_pols R S X f g|]
⟹ ∃u ∈ carrier R. ∃v ∈ carrier R. (u = 0_R
∨ (u ≠ 0_R ∧ deg_n R S X u ≤ max ((deg_n R S X h)
- (deg_n R S X f)) (deg_n R S X g))) ∧ (v = 0_R
∨ (v ≠ 0_R ∧ deg_n R S X v ≤ (deg_n R S X f)))
∧ (u ·_R f +_R (v ·_R g) = h)"
```

In terms of the generalized deg, the above lemma is expressed simply as

```
lemma rel_prime_equation:"[| ring R; field S; polyn_ring R S X;
f ∈ carrier R; g ∈ carrier R; 0 < deg R S X f;
0 < deg R S X g; rel_prime_pols R S X f g;
h ∈ carrier R |]
⟹ ∃ u ∈ carrier R. ∃ v ∈ carrier R.
(deg R S X u ≤ amax ((deg R S X h) - (deg R S X f))
(deg R S X g)) ∧ (deg R S X v ≤ (deg R S X f))
∧ (u ·_R f +_R (v ·_R g) = h)"
```

This means:
Let R = S[X] be a polynomial ring with field coefficients, and let f, g be relatively prime polynomials in R with nonzero degree, and let h be a polynomial. Then there are polynomials u, v such that u f + v g = h, satisfying conditions $\deg u \leq \max((\deg h - \deg f), \deg g)x$ *and* $\deg v \leq \deg f$.

Now, we explain Hensel's lemma in short.
Let S be a valuation ring with the maximal ideal (t), t ∈ S. Let R be the polynomial ring S[X] and let f be an element of R. If the natural image $\bar{f}$ *of f in the ring R' = (S/(t))[Y] is factorized as* $\bar{f} = g'h'$ *with relatively prime polynomials g', h' ∈ R'. Then we have polynomials g, h such that* $\bar{g} = g'$, $\bar{h} = h'$ *and* $f = gh$. *Moreover we have* $\deg g \leq \deg g'$.

A proof of this lemma is to make approximations of g and h recursively and show that the limits of those approximations gives factors. A recursive construction requires the following existence lemma:

```
lemma P_mod_approximation:"[|(1) ring R; integral_domain S;
polyn_ring R S X;
(2) t ∈ carrier S; t ≠ 0_S; maximal_ideal S (S ♢ t);
(3) ring R'; polyn_ring R' (ringF (S /_r (S ♢ t))) Y;
(4) f ∈ carrier R;
(5) ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t)) f
≠ 0_R';
(6) g ∈ carrier R; h ∈ carrier R;
(7) deg_n R S X g + deg_n R S X h ≤ deg_n R S X f;
(8) ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t)) g
≠ 0_R';
(9) 0 < deg_n R' (ringF (S /_r (S ♢ t))) Y
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y
(pj S (S ♢ t)) g);
(10) ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t))
h ≠ 0_R';
(11) 0 < deg_n R' (ringF (S /_r (S ♢ t))) Y
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y
(pj S (S ♢ t)) h);
(12) rel_prime_pols R' (ringF (S /_r (S ♢ t))) Y
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y
(pj S (S ♢ t)) g)
(ext_rH R S X R' (ringF (S /r (S ♢ t))) Y
(pj S (S ♢ t)) h);
(13) P_mod R S X (S ♢ (t^Sm)) (f +R
-R (g ·R h)); 0 < m |] ⟹
(14) ∃g1 h1. g1 ∈carrier R ∧ h1 ∈ carrier R ∧
(15) (deg_n R S X g1 ≤ deg_n R S X g) ∧
(16) P_mod R S X (S ♢ t) (g +R -R g1) ∧
(17) P_mod R S X (S ♢ t) (h +R -R h1) ∧
(18) P_mod R S X (S ♢ (t^S(Suc m))) (f +R (-R (g1 ·R h1)))"
```

We describe the meaning of the above lemma:

(1) Let R be a polynomial ring $S[X]$, with an integral domain S.
(2) Let (t) with $t \neq 0$ be a maximal ideal of S.
(3) Let R' be a polynomial ring $S/(t)[Y]$. Since $S/(t)$ is a field, we write ring$F(S/(t))$ to denote that there is an inverse operator for multiplication.
(4) Let f be a polynomial in $S[X]$.
(5) the natural image $\bar{f}$ in R' of f is nonzero.
(6) g, h be polynomials in $S[X]$.
(7) $\deg g + \deg h \leq f$
(8) the natural image $\bar{g}$ in R' of g is nonzero.
(9) $0 < \deg g'$ in R'.
(10) the natural image $\bar{g}$ in R' of g is nonzero.
(11) $0 < \deg h'$ in R'.
(12) $1 \in (g') + (h')$.
(13) $f \cong gh(\mathrm{mod}(t^m)), 0 < m$.
$\Longrightarrow$
(14) exists g_1, h_1 in R such that,
(15) $\deg g_1 \leq \deg g$,
(16) $g \cong g_1 \mathrm{mod}(t)$ in R,
(17) $h \cong h_1 \mathrm{mod}(t)$ in R,
(18) $f \cong g_1 h_1 \mathrm{mod}(t^{m+1})$

To use this lemma recursively, we define a pair of polynomials:

```
Hensel_next::"[('a, 'b) RingType_scheme, ('a, 'c) RingType_scheme,
'a, 'a, ('a set, 'm) RingType_scheme, 'a set,'a, nat] ⇒
('a × 'a) ⇒ ('a × 'a)"
```

```
("(9Hen_ _ _ _ _ _ _ _ _ _)" [67,67,67,67,67,67,67,68]67)

"Hen_{RSXtR'Yf} m gh == SOME gh1. gh1 ∈ carrier R <*> carrier R ∧
(deg R S X (fst gh1) ≤ deg R' (ringF (S /_r (S ♢ t))) Y
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t))
(fst gh1))) ∧ P_mod R S X (S ♢ (t^{S m})) ((fst gh) +_R -_R
(fst gh1)) ∧ (deg R S X (snd gh1) + deg R' (ringF
(S /_r (S ♢ t))) Y (ext_rH R S X R' (ringF (S /_r (S ♢ t)))
Y (pj S (S ♢ t)) (fst gh1)) ≤ deg R S X f) ∧
P_mod R S X (S ♢ (t^{S m})) ((snd gh) +_R -_R (snd gh1)) ∧
P_mod R S X (S ♢ (t^{S (Suc m)}))
(f +_R(-_R ((fst gh1) ·_R (snd gh1))))"
```

The following Hensel_pair is a pair of polynomials and the limits of each component is a factor of the given polynomial f which is to be factorized.

```
Hensel_pair::"[('a, 'b) RingType_scheme, ('a, 'c) RingType_scheme,
'a, 'a, ('a set, 'm) RingType_scheme, 'a set, 'a, 'a, 'a,
nat] ⇒ ('a × 'a)"
("(10Hpr_________ _)" [67,67,67,67,67,67,67,67,67,68]67)

primrec
Hpr_0: "Hpr_{RSXtR'Yfgh} 0 = (g, h)"
Hpr_Suc: "Hpr_{RSXtR'Yfgh} (Suc m) =
Hen_{RSXtR'Yf} (Suc m) (Hpr_{RSXtR'Yfgh} m)"
```

We have a lemma which implies the Hensel_pair is a Cauchy sequence of polynomials.

```
lemma P_mod_diffxxx5_1:"[| ring R; integral_domain S;
polyn_ring R S X; t ∈ carrier S; maximal_ideal S (S ♢ t);
ring R'; polyn_ring R' (ringF (S /_r (S ♢ t))) Y;
f ∈ carrier R; g ∈ carrier R; h ∈ carrier R;
deg R S X g ≤ deg R' (ringF (S /_r (S ♢ t))) Y
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t)) g);
deg R S X h + deg R' (ringF (S /_r (S ♢ t))) Y
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t)) g)
≤ deg R S X f;
0 < deg R' (ringF (S /_r (S ♢ t))) Y
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t)) g);
0 < deg R' (ringF (S /_r (S ♢ t))) Y
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t)) h);
rel_prime_pols R' (ringF (S /_r (S ♢ t))) Y
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t)) g)
(ext_rH R S X R' (ringF (S /_r (S ♢ t))) Y (pj S (S ♢ t)) h);
P_mod R S X (S ♢ t) (f +_R -_R (g ·_R h))|] ⟹
(Hpr_{RSXtR'Yfgh} (Suc m)) ∈ carrier R × carrier R ∧
```

```
ext_rH R S X R' (ringF (S /_r (S ◇ t))) Y (pj S (S ◇ t))
              (fst (Hpr_RSXtR'Yfgh (Suc m))) =
ext_rH R S X R' (ringF (S /_r (S ◇ t))) Y (pj S (S ◇ t))
               (fst (g, h)) ∧
ext_rH R S X R' (ringF (S /_r (S ◇ t))) Y (pj S (S ◇ t))
              (snd (Hpr_RSXtR'Yfgh (Suc m))) =
ext_rH R S X R' (ringF (S /_r (S ◇ t))) Y (pj S (S ◇ t))
              (snd (g, h)) ∧
(deg R S X (fst (Hpr_RSXtR'Yfgh (Suc m))) ≤
deg R' (ringF (S /_r (S ◇ t))) Y (ext_rH R S X R'
  (ringF (S /_r (S ◇ t))) Y (pj S (S ◇ t))
               (fst (Hpr_RSXtR'Yfgh (Suc m))))) ∧
P_mod R S X (S ◇ (t^S^(Suc m)))) ((fst (Hpr_RSXtR'Yfgh m)) +_R
              -_R (fst (Hpr_RSXtR'Yfgh (Suc m)))) ∧
(deg R S X (snd (Hpr_RSXtR'Yfgh (Suc m))) +
deg R' (ringF (S /_r (S ◇ t))) Y (ext_rH R S X R'
(ringF (S /_r (S ◇ t))) Y (pj S (S ◇ t))
(fst (Hpr_RSXtR'Yfgh (Suc m)))) ≤ deg R S X f) ∧
P_mod R S X (S ◇ (t^S^(Suc m))) ((snd (Hpr_RSXtR'Yfgh m)) +_R
             -_R (snd (Hpr_RSXtR'Yfgh (Suc m)))) ∧
P_mod R S X (S ◇ (t^S^(Suc(Suc m)))) (f +_R -_R
((fst (Hpr_RSXtR'Yfgh (Suc m))) ·_R (snd (Hpr_RSXtR'Yfgh (Suc m)))))"
```

3 Valuations

3.1 Definition of a valuation

We formalize Hensel's lemma for a polynomial ring with coefficients in a valuation ring. If the valuation ring is "complete", there is a limit polynomial which is a factor of a given polynomial.

Valuation is a function v from the carrier of a field K to Z ∪ $\{\infty\}$ satisfying the following conditions:

(1) v (0) = ∞

(2) forall x, y in the carrier K, we have an equation
v (x y) = v (x) + v(y)

(3) forall x in the carrier K, if $0 \leq$ v(x) then $0 \leq$ v (1 + x)

(4) there exists an element x of carrier K, such that v (x) $\neq \infty$ and
v (x) $\neq 0$

The set Z ∪ $\{\infty\}$ is the set of augmented integers not equal to $-\infty$, and we denote this set by Z_∞. The ordering of Z_∞ is that induced naturally by integer ordering and for any integer z, z $< \infty$.

```
valuation::"['b FieldType, 'b ⇒ ant] ⇒ bool"
"valuation K v == v ∈ extensional (carrier K)
∧ v ∈ carrier K → Z∞ ∧ v (0_K) = ∞
∧ (∀ x ∈ ((carrier K) - 0_K). v x ≠ ∞)
∧ (∀ x ∈ (carrier K).
∀ y ∈ (carrier K). v (x ·_K y) = (v x) + (v y))
∧ (∀ x ∈ (carrier K). 0 ≤ (v x) ⟶ 0 ≤ (v (1_K +_K x)))
∧ (∃ x. x ∈ carrier K ∧ (v x) ≠ ∞ ∧ (v x) ≠ 0)"
```

We have a simple

```
lemma amin_le_plus:"[| field K; valuation K v; x ∈ carrier K;
y ∈ carrier K |]
⟹ (amin (v x) (v y)) ≤ (v (x +_K y))"
```

Here, `(amin (v x) (v y))` is minimum of augmented integers.

3.2 The valuation ring of a valuation v

The valuation ring of a valuation v is formalized as

```
Vr::"['r FieldType, 'r ⇒ ant] ⇒ 'r RingType"
"Vr K v == Sr K (x. x ∈ carrier K ∧ 0 ≤ (v x))"
vp::"['r FieldType, 'r ⇒ ant] ⇒ 'r set"
"vp K v == x. x ∈ carrier (Vr K v) ∧ 0 < (v x)"
r_apow::"[('r, 'm) RingType_scheme, 'r set, ant] ⇒ 'r set"
"r_apow R I a == if a = ∞ then 0_R
else (if a = 0 then carrier R else I^(♢R(na a)))"
```

`r_apow R I a` is the power of an ideal I with augmented integer coefficient a. Sr K {_} means a subring of K having the carrier {_}, and we can show

```
lemma Valuation_ring:"[| field K; valuation K v|]
⟹ ring (Vr K v)"
```

Hence the valuation ring is a ring and

```
lemma Vring_integral:"[| field K; valuation K v|]
⟹ integral_domain (Vr K v)"
```

The above lemma shows that a valuation ring is an integral domain. As for a subset `vp K v` consisting of elements of the `carrier K` having strictly positive value:

```
vp::"['r FieldType, 'r ⇒ ant] ⇒ 'r set"
"vp K v == {x. x ∈ carrier (Vr K v) ∧ 0 < (v x)}"
```

We have

```
lemma vp_ideal:"[| field K; valuation K v|]
⟹ ideal (Vr K v) (vp K v)"
```

Hence vp K v is an ideal of the valuation ring.

```
lemma vp_maximal:"[| field K; valuation K v|]
⟹ maximal_ideal (Vr K v) (vp K v)"
```

Hence we see that vp is a maximal ideal of the valuation ring.

```
lemma Vring_local:"[| field K; valuation K v;
maximal_ideal (Vr K v) I|] ⟹ (vp K v) = I"
```

This lemma shows the valuation ring Vr K v is a local ring (i.e. has only one maximal ideal). We have the following lemma, which shows that the valuation ring is a principal ideal:

```
lemma Vring_principal:"[|field K; valuation K v;
ideal (Vr K v) I|] ⟹ ∃ x ∈ I. I = Rxa (Vr K v) x"
```

Hence the maximal ideal vp K v is generated by one element t (t $\neq$ 0) of Vr K v.

Because the valuation ring is a principal ideal domain, the following lemma is natural, but non-trivial.

```
lemma ideal_apow_vp:"[| field K; valuation K v;
ideal (Vr K v) I |] ⟹ I = (vp K v)^((VrKv)(n_valKv(IgKvI))
```

We have a lemma connecting n_val K x and the principal ideal (x) generated by x, with x in the valuation ring Vr K v.

```
lemma ideal_apow_n_val:"[| field K; valuation K v; x ∈ carrier
(Vr K v) |] ⟹ (Vr K v) ♢ x = (vp K v)^((VrKv)(n_valKvx))"
```

3.3 Limit with respect to a valuation v

Given a valuation v of a field K, we can formalize "limit" as

```
limit ::"['b FieldType, 'b ⇒ ant, nat ⇒ 'b, 'b] ⇒ bool"
("(4lim _ _ _ _)" [90,90,90,91]90)
"lim_(K v) f b == ∀N. ∃M.
(∀n. M < n ⟶ ((f n) +_K (-_K b)) ∈ (vp K v)^(♢(VrK v) N))"
```

Here, K is a field on which the value v is defined and f is a function from the set NSet of all natural numbers to the carrier of K. We can take f as a sequence of elements of the carrier K. vp K v is the maximal ideal of the valuation ring Vr K v and b is an element of the carrier K. For a short explanation of the meaning of the above definition, let us denote the valuation ring as O and let us denote vpr as P, then we see limit...f b means

$$\forall N \text{ there is a natural number } M \text{ such that } \forall n > M, f(n) - b \in P^N$$

The following lemma shows that limit is also described in terms of the valuation.

```
lemma limit_diff_val:"[| field K; b ∈ carrier K; ∀j.
f j ∈ carrier K; valuation K v|]
⟹ (lim_{Kv} f b) = (∀N. ∃M. ∀n. M < n
⟶ (an N) ≤ (v((f n) +_K (-_K b))))"
```

The following lemma guarantees that the limit value is unique.

```
lemma vp_pow_inter_zero:"[| field K; valuation K v|]
⟹ (∩ {I. ∃n. I = (vp K v)^{◇(Vr K v) n}}) = {0_K}"
```

In short, by using above notation we can express this as

$$\cap_{\mathtt{n}}\ \mathtt{P}^{\mathtt{n}}\ \mathtt{=}\ \{\mathtt{0}\}$$

By using O and P above we can express $\mathtt{(vp\ K\ v)}^{\diamond(\mathtt{Vr\ K\ v})\ \mathtt{n}}$ as P^n and this means n-th power of the ideal P.

A complete field K with respect to a valuation v is a field in which any Cauchy sequence converges to some element of K. Cauchy sequence is formalized as

```
Cauchy_seq::"['b FieldType, 'b ⇒ ant, nat ⇒ 'b] ⇒ bool"
("(3Cauchy _ _ _ _ _ _)" [90,90,91]90)
"Cauchy_{K v} f == (∀n. (f n) ∈ carrier K)
∧ ( ∀N. ∃M. (∀n m. M < n ∧ M < m
⟶ ((f n) +_K (-_K (f m))) ∈ (vp K v)^{◇(Vr K v) N}))"
```

A complete field is formalized as

```
v_complete::"['b ⇒ ant, 'b FieldType] ⇒ bool"
("(2Complete_ _)" [90,91]90)
"Complete_v K == ∀f. (Cauchy_{K v} f)
⟶ (∃b. b ∈ (carrier K) ∧ lim_{K v} f b)"
```

that is, any Cauchy sequence converges to some element of K.

Let K be a field having a valuation v and let t be an element of the valuation O of v and let (t) = P the maximal ideal of O. Let S be a complete system of representatives of O/P, i.e. image of a function s : O/P → O such that p∘s = id, where p is the natural projection p:O → O/P.

We have an expansion theorem:

Let K be a field and let v be a valuation. Let t be an element of K such that (t) is the maximal ideal of the valuation ring of v. Then for any $x \in K$,

$$x = t^n(a_0 + a_1t + \ldots + a_nt^n + \ldots)$$

A formalization of this lemma is as follows:

```
consts partial_sum :: "['b FieldType, 'b, 'b ⇒ ant,'b]
⇒ nat ⇒ 'b" ("(5psum _ _ _ _ _)" [96,96,96,96,97]96)
```

```
primrec psum_0 : "psumK x v t 0 = (csrp_fn (Vr K v)
(vp K v) (pj (Vr K v) (vp K G v) (x ·K tK(-(tna (v x))))))
·K (tK( tna (v x)))"

psum_Suc: "psumK x v t (Suc n) = (psumK x v t n) +K ((csrp_fn
(Vr K v) (vp K v) (pj (Vr K v) (vp K v) ((x +K -K
(psumK x v t n)) ·K (tK(- (tna (v x) + int (Suc n)))))))
·K (tK(tna (v x) + int (Suc n))))"
```

Here `csrp_fn (Vr K v) (vp K v)` is the map s above having one more condition that s (P) = 0. `pj (Vr K v) (vp K v)` is the map p above. Let x be an element of K, then the value v x is in Z_∞. We see v (t^m) = an m. If v x = an m then we have v $(x/t^{-m}) = 0$ and $x/t^{-m} \in O$. We put $a_0 = s \circ p\ (x/t^{-m})$, then $\mathrm{psum}_{K\,x\,v\,t}$ $0 = a_0 t^m$. Since $p(a_0) = p(x/t^{-m})$, we have x/t^{-m} - $a_0 \in P$, therefore we have an element a'_1 such that

$$x/t^{-m} - a_0 = a'_1\ t$$

from this we have $(x/t^{-m} - a_0)/t = a'_1$. We have a_1 in S such that $a_1 \cong a'_1$, and we see that there exists an element a'_2 in O satisfying $a'_1 = a_1 + a'_2 t$.

Hence we have an equation

$$(x/t^{-m} - a_0)/t = a_1 + a'_2 t,$$

and this equation is equal to the equation

$$x = t^m(a_0 + a_1 t + a'_2 t^2)$$

Thus we see that

$$\mathrm{psum}_{K\,x\,v\,t}\ n = t^m(a_0 + a_1 t + \ldots + a_n t^n),$$

We have formalization of the expansion theorem:

```
theorem expansion_thm:"[| field K; valuation K v;
t ∈ carrier K; t ≠ 0K; v t = 1; x ∈ carrier K; x ≠ 0K|]
⟹ limK v (partial_sum K x v t) x"
```

The maximal ideal P of a valuation ring O is generated by one element, and we saw $\cap_n P^n = \{0\}$, the limit of a Cauchy sequence is determined uniquely. And if we have a sequence of polynomials $\{f_i(X)\}_{i=1,2,\ldots}$ satisfying a condition

$$\forall N.\ \exists M \text{ such that } \forall\ n\ m. M < m \wedge M < n \Longrightarrow f_n - f_m \in P^N O[x]$$

then we see that there is a unique limit polynomial f of the sequence $\{f_i(X)\}_{i=1,2,\ldots}$.

4 Hensel's lemma

We give a polynomial f in a ring of polynomials O[X] with O a valuation ring. Let M be the maximal ideal of O, then Hensel's lemma states that if $f \in O[X]$ is factorized as $\bar{f} = g'h'$ in (O/M)[X], then there are two polynomials g, $h \in O[X]$ such that f = g h.

A Cauchy sequence of polynomials and limit of a sequence of polynomials are defined as above. One point we have to note is

```
lemma Plimit_deg1:"[| field K; valuation K v; ring R;
polyn_ring R (Vr K v) X; ∀ n. F n ∈ carrier R;
∀ n. deg R (Vr K v) X (F n) ≤ ad; p ∈ carrier R;
Plimit_R X K v F p |] ⟹ deg R (Vr K v) X p ≤ ad"
```

This lemma means a sequence of polynomials has a upper bound, then a limit is also bounded by the bound.

The limit of a Cauchy sequence is determined uniquely. We formalized Hensel's lemma as

```
theorem Hensel:"[| (1) field K; valuation K v;
(2) Complete_v K; (3) ring R; polyn_ring R (Vr K v) X;
(4) ring S; polyn_ring S (ringF ((Vr K v) /_r (vp K v))) Y;
(5) f ∈ carrier R; f ≠ 0_R; g' ∈ carrier S; h' ∈ carrier S;
0 < deg S (ringF ((Vr K v) /_r (vp K v))) Y g';
0 < deg S (ringF ((Vr K v) /_r (vp K v))) Y h';
((ext_rH R (Vr K v) X S (ringF ((Vr K v) /_r (vp K v))) Y
           (pj (Vr K v) (vp K v))) f) = g' ·_S h';
(6)rel_prime_pols S (ringF ((Vr K v) /_r (vp K v))) Y g' h'|] ⟹
(7) ∃ g h. g ∈ carrier R ∧ h ∈ carrier R ∧
deg R (Vr K v) X g ≤ deg S (ringF ((Vr K v) /_r (vp K v))) Y g'
∧ f = g ·_R h"
```

To explain the meaning of this theorem, we put line numbers. We denote the valuation ring (`Vr K v`) as O, the maximal ideal (`vp K v`) as (t) or as P and the natural projection from O to O/P as p. Let g be an element of O, we also denote p (g) as $\bar{g}$.

(1) K is a field with a valuation v

(2) K is a field which is complete with respect to the valuation v

(3) R is a polynomial ring O[X]

(4)S is a polynomial ring (O/P)[Y]. We note that O/P is a field since P is the maximal ideal of O. Note that ringF (O/P) is a FieldType.

(5) f ∈ R, g′ ∈ S and h′ ∈ S and $\bar{f}$ = g′ h′ where g′ h′ are non constant polynomials. That is "a factorization is given in the polynomial ring O/P[Y]".

(6) (g′, h′) = 1 in S, that is, these two are relatively prime nonconstant polynomials.

(7) There are two polynomials g, h in R such that f = g h, with deg g ≤ deg g′

References

[1] Atiyah, M.F. and Macdonald, I.C. (1969) Introduction to Commutative Algebra, Addison-Wesley Publishing Co.

[2] Ballarin, C. (1996) Univariate Polynomials, Isabelle2004/src/HOL/Algebra/pol

[3] Iwasawa, K. (1952) Algebraic Functions(in Japanese). Iwanami Shoten.

[4] Nipkow, T., Paulson, L.C. and Wenzel, M.(2002) Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Springer Verlag.

Tactic-based Optimized Compilation of Functional Programs

Thomas Meyer and Burkhart Wolff

Universität Bremen, Germany
ETH Zürich, Switzerland

Abstract Within a framework of correct code-generation from HOL-specifications, we present a particular instance concerned with the optimized compilation of a lazy language (called MiniHaskell) to a strict language (called MiniML).
Both languages are defined as shallow embeddings into denotational semantics based on Scott's cpo's, leading to a derivation of the corresponding operational semantics in order to cross-check the basic definitions.
On this basis, translation rules from one language to the other were formally derived in Isabelle/HOL. Particular emphasis is put on the optimized compilation of function applications leading to the side-calculi inferring e.g. strictness of functions.
The derived rules were grouped and set-up as an instance of our generic, tactic-based translator for specifications to code.

1 Introduction

The verification of compilers, or at least the verification of compiled code, is known to be notoriously difficult. This problem is still an active research area [3, 4, 12]. In recent tools for formal methods, the problem also re-appears in the form of code-generators for specifications — a subtle error at the very end of a formal development of a software system may be particularly frustrating and damaging for the research field as a whole.

In previous work, we developed a framework for *tactic-based* compilation [5]. The idea is to use a theorem prover itself as a tool to perform source-to-source transformations, controlled by tactic programs, on programming languages embedded into a HOL prover. Since the source-to-source transformations can be derived from the semantics of the program languages embedded into the theorem prover, our approach can guarantee the correctness of the compiled code, provided that the process terminates successfully and yields a representation that consists only of constructs of the target language. Constructed code can be efficient, since our approach can be adopted to optimized compilation techniques, too.

In this paper, we discuss a particular instance of this framework. We present the semantics of two functional languages, a Haskell-like language and an ML-like language for which a *simple* one-to-one translator to SML code is provided. We apply the shallow embedding technique for these languages [1] into standard denotational semantics — this part of our work can be seen as a continuation of the line of "Winskel is almost right"-papers [8], which formalize proofs of a denotational semantics textbook [11, chapter 9].

As a standard translation, a lazy language can be transformed semantically equivalent via continuation passing style [2] into an eager language. While this compilation is known to produce fairly inefficient code, we also use derived rules for special cases requiring strictness- or definedness analysis. While we admit that the basic techniques are fairly standard in functional compilers, we are not aware of any systematic verification of the underlying reasoning in a theorem prover. Thus, we see here our main contribution.

The plan of the paper is as follows: After a brief outline of the general framework for tactic based compilation and a brief introduction into the used theories for denotational semantics, we discuss the embeddings of MiniHaskell and MiniML into them. These definitions lead to derivations of "classical" textbook operational semantics. In the sequel, we derive transformation rules between these two languages along the lines described by our framework. Then we describe the side-calculus to infer strictness required for optimized compilation; an analogous calculus for definedness is ommitted here.

2 Background

2.1 Concepts and Use of Isabelle/HOL

Isabelle [9] is a generic theorem prover of the LCF prover family; as such, we use the possibility to build programs performing symbolic computations over formulae in a logically safe (conservative) way on top of the logical core engine: this is what our code-generator technically is. Throughout this paper, we will use Isabelle/HOL, the instance for Church's higher-order logic. Isabelle/HOL offers support for data types, primitive and well-founded recursion, and powerful generic proof engines based on higher-order rewriting which we predominantly use to implement the translation phases of our code-generator.

Isabelle's type system provides parametric polymorphism enriched by type classes: It is possible to constrain a type variable $\alpha :: \texttt{order}$ to specify that an operator $\_ <= \_$ must be declared on any α; this syntactic concept known from languages such as Haskell is extended in Isabelle by semantic constraints: the operator must additionally fulfill the properties of a partial order.

The proof engine of Isabelle is geared towards rules of the form $A_1 \Rightarrow (\ldots \Rightarrow (A_n \Rightarrow A_{n+1}) \ldots)$ which can be interpreted as "from assumptions A_1 to A_n, infer conclusion A_{n+1}". This corresponds to the textbook notation

$$\frac{A_1 \quad \ldots \quad A_n}{A_{n+1}}$$

which we use throughout this paper.

Inside these rules, the meta-quantifier $\bigwedge$ is used to capture the usual side-constraint "x must not occur free in the assumptions" for quantifier rules; meta-quantified variables can be considered as "fresh" free variables.

2.2 The Framework for Code-Generation

Our generic framework [5] is designed to cope with various executability notions and to provide technical support for them. The following diagram in figure 1 represents the particular instance of the general framework discussed in this paper.

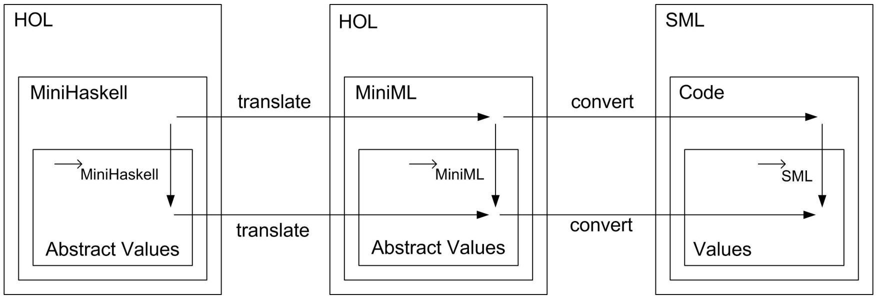

HOL
MiniHaskell
$\longrightarrow_{\text{MiniHaskell}}$
Abstract Values
translate
translate
HOL
MiniML
$\longrightarrow_{\text{MiniML}}$
Abstract Values
convert
convert
SML
Code
$\longrightarrow_{\text{SML}}$
Values

Figure 1. Basic Concepts

Here, the left block represents the language MiniHaskell, the center block the language MiniML, which are both presented as conservative shallow embedding into a theory of Scott Domains described in Section 2.3. A subset of both languages are the set of *abstract values*. The embeddings are mirrored by the corresponding terms of a (concrete) programming language, i.e. SML, and its subset of (concrete) *values* like e.g. the integers 1,2,3,.... The first two worlds are connected by the translate function, that consists of several tactics that control the translation process by source-to-source translation rules. The latter two worlds are connected by the code-generation function convert provided by our framework that is required to be total on the domain of abstract programs.

The three relations $\to_{\mathsf{MiniHaskell}}$, $\to_{\mathsf{MiniML}}$ and $\to_{\mathsf{SML}}$ represent the operational semantics of the three languages. We require that they represent partial functions from programs to values. These operational semantics serve as cross-check of our denotational definitions of the language; in particular, $\to_{\mathrm{SML}}$ can be compared against an (abstracted) version of the *real* SML semantics [6] in order to validate convert. Making these two diagrams commute (while the first commutation is based on formal proofs presented in this paper) constitutes the correctness of our overall translation process.

2.3 Denotational Semantics in HOL

The cornerstone of any denotational semantics is its fixpoint theory that gives semantics to systems of (mutual) recursive equations. The well-known Scott-Stratchey-approach is based on complete partial orders (*cpo*'s); variants thereof have also been used in standard semantics textbooks such as [11] to give semantics to the languages we discuss here (cf. chapter 9).

Several versions of denotational semantics theories are available for Isabelle [7, 10]. In both, the type class mechanism is used in order to model cpo's, which provide a least element $\perp$ and completeness on any type belonging to class `cpo`. This is essentially captured in the theory [10] underlying this work in the axiomatic class definition

```
axclass
  cpo < cpo0
  least        ⊥ ≤ x
  complete     directed X ⇒ (∃ b. X <<| b)
```

i.e. completeness means that for any directed set (any non-empty set where two elements have a supremum) there exists a least upper bound.

Moreover, in this type class a number of key concepts such as definedness and strictness of a function and making a function strict are defined:

```
DEF        :: α::cpo0 ⇒ bool          DEF x ≡ x≠⊥
is_strict  :: (α::cpo0 ⇒ β::cpo0) ⇒ bool
              is_strict f ≡ (f ⊥ = ⊥)
strictify  :: (α ⇒ β::cpo) ⇒ α ⊥ ⇒ β
              strictify f x ≡ if DEF(x) then f(x) else ⊥
```

Further, a type constructor can be defined that assigns to each type τ a *lifted type* $\tau_\perp$ by disjointly adding the $\perp$-element. All types lifted by this type constructor are automatically in the type class `cpo` but not necessarily vice versa. The function $\lfloor \_ \rfloor : \alpha \to \alpha_\perp$ denotes the injection, the function $\lceil \_ \rceil : \alpha_\perp \to \alpha$ its inverse, extended by $\lceil \perp \rceil = \perp$.

On cpo's, the usual fixpoint combinator `fix` is defined that is shown to posses the crucial fixpoint property

$$\frac{\texttt{cont}\ f}{\texttt{fix}\ f = f(\texttt{fix}\ f)}$$

for all functions f that are continuous. Further, there is the usual induction principle for all fixpoints of all types belonging to class `cpo`:

$$\frac{\texttt{cont}\ f \quad \texttt{adm}\ P \quad \bigwedge x.\ \begin{array}{c} P(x) \\ \vdots \\ P(f\ x) \end{array}}{P(\texttt{fix}\ f)}$$

where the second-order predicate `adm` for *admissibility* captures that a predicate P holds for a fixpoint if it holds for any approximation of it. `adm` distributes over universal quantification, conjunction and disjunction, but not necessarily over negation. Being defined is an admissible predicate, being total not. As a consequence of induction, we derived a kind of bi-simulation principle:

$$\frac{\texttt{cont}\ f \quad \texttt{cont}\ f' \quad \bigwedge x.\ \begin{array}{c} [P\ x] \\ \vdots \\ f\ x = f'\ x \end{array} \quad \bigwedge x.\ \begin{array}{c} [P\ x] \\ \vdots \\ P(f\ x) \end{array} \quad \texttt{adm}\ P}{\texttt{fix}\ f = \texttt{fix}\ f'}$$

which is the key for the proof of several crucial inference principles over recursive programs to be described in the subsequent sections. If some property P is invariant through execution of the body f, then P can be assumed for the "inner call" when proving the bodies f and f' equivalent over them.

3 The Semantics of MiniHaskell and MiniML

3.1 The Denotational Semantics of MiniHaskell

Based on the theories of denotational semantics, we define our first contribution — the formal definition of the lazy language MiniHaskell. The types of basic operations like `Bool` were lifted from HOL types

```
types
  Bool = bool⊥   Nat = nat⊥   Unit = unit⊥
```

and basic constants such as `TRUE` or `ONE` are defined accordingly by

```
TRUE :: Bool   TRUE ≡ ⌊True⌋
ONE  :: Nat    ONE  ≡ ⌊1⌋
```

The core of the MiniHaskell semantics consists of the definitions for the abstraction, application, conditional and the `LET`-construct. As well-known in the literature, an important difference between the denotational theory and the object language has to be made: the abstraction in MiniHaskell is a *value* — a so-called *closure* — and not a function space. Thus, a naive identification of the object language `LAM` with the meta language λ results in a completely wrong model of the operational behaviour: the expression `LAM` x. `ONE DIV ZERO` should be a *value*, i.e. different from λ`x. 1 DIV 0`, which is just λ`x.`$\bot$ or just $\bot$ in the function space. Consequently, the *lifted* function space is used, defined by:

```
types (α,β) ⇛ = (α ⇒ β)⊥
```

which results in the following definitions for the abstraction

```
Lam :: (α::cpo ⇒ β::cpo) ⇒ (α ⇛ β)
       Lam F ≡ ⌊F⌋
```

and its inverse, the application

```
▷l :: (α::cpo ⇛ β::cpo) ⇒ α ⇒ β
     F ▷l x ≡ ⌈F⌉ x
```

where we may write `LAM` x. $P\ x$ for `Lam` P. The `LET` construct is just a syntactical shortcut and defined by the application. The remaining definitions of the conditional and the recursor are standard:

```
If :: [Bool, α::cpo, α] ⇒ α
      IF x THEN y ELSE z ≡ case x of
                                 ⌊v⌋ ⇒ if v then y else z
                               | ⊥   ⇒ ⊥
REC :: (α::cpo ⇒ α) ⇒ α
       REC f ≡ fix f
```

The basic operations of MiniHaskell are just strictified versions of the elementary operations of HOL. The paradigmatic example for a 1-ary and a 2-ary function are defined as follows:

```
SUC :: Nat ⇒ Nat
       SUC ≡ strictify(λx. ⌊Suc x⌋)
^<^ :: [Nat, Nat] ⇒ Bool
       (op ^<^) ≡ strictify(λx. strictify(λy. ⌊x<y⌋))
```

An example for a partial function is `DIV`:

```
DIV :: [Nat, Nat] ⇒ Nat
       DIV ≡ strictify(λx.
                 strictify(λy. if y=0 then ⊥
                               else ⌊x div y⌋))
```

As top-level constructs, we introduce the following two program definition constructs:

```
VAL  :: [α,α] ⇒ bool
        VAL f E ≡ (f = E)
FUN  :: [α::cpo, α ⇒ α] ⇒ bool
        FUN f F ≡ (f = REC(F)) ∧ cont F
```

This means that a recursive program is representable by the recursor `REC` of the language MiniHaskell under the condition, that the representing functional `F` is continuous. The Isabelle syntax engine is set up to parse also mutual recursive function definitions as a combination of `fix` and pairing. For example, a mutual recursive program in the object language MiniHaskell looks as follows:

```
fun fac x = IF x^=^ZERO THEN ONE ELSE x*(fac ▷l (x-ONE))
and add_fac x y = x+fac ▷l y
and suc_fac a   = add_fac ▷l ONE ▷l a;
```

Note, that the operators (`op` +), (`op` -) and (`op` *) are the overloaded (strictified) variants from MiniHaskell.

3.2 Lazy Operational Semantics of MiniHaskell

In the following, we derive the operational semantics presented in [11] in order to validate our denotational definitions. The basic concept of this operational semantics is a notion of terms representing values, called *canonical forms*. The judgment $t \in C_\tau$ states that a term t is a canonical form of type τ. It is defined by the following structural induction on the type τ:

Ground type: $n \in C_{\texttt{int}} = \{$`ZERO`, `ONE`, `TWO`, $\ldots\}$ and
$b \in C_{\texttt{bool}} = \{$`TRUE`, `FALSE`$\}$

Function type: Closed abstractions are canonical forms, i.e.
(`LAM` $x.\ t) \in C_{\tau_1 \to \tau_2}$ if t is closed

Note, that we can not give an inductive definition for canonical forms since we use a shallow embedding (the types presented above are represented on the meta-level). Nevertheless, by defining the evaluation relation $\to_l$ as equivalent to the logical equality (i.e. evaluation must be correct), we can now derive the rules for the evaluation relation and check that they have the appropriate form $t \rightarrow_l c$, where t is a typeable closed term and c is a canonical form, meaning t evaluates to c. In the following, c, c_1, c_2 and c_3 range over canonical forms:

$$c \rightarrow_l c \qquad \frac{t_1 \rightarrow_l c_1 \quad t_2 \rightarrow_l c_2}{t_1\ op\ t_2 \rightarrow_l c_1\ op\ c_2}$$

$$\frac{t_1 \rightarrow_l \texttt{TRUE} \quad t_2 \rightarrow_l c_2}{(\texttt{IF}\ t_1\ \texttt{THEN}\ t_2\ \texttt{ELSE}\ t_3) \rightarrow_l c_2} \qquad \frac{t_1 \rightarrow_l \texttt{FALSE} \quad t_3 \rightarrow_l c_3}{(\texttt{IF}\ t_1\ \texttt{THEN}\ t_2\ \texttt{ELSE}\ t_3) \rightarrow_l c_3}$$

$$\frac{t_1 \rightarrow_l \texttt{LAM}\ x.\ t \quad t[x := t_2] \rightarrow_l c}{t_1 \rhd_l t_2 \rightarrow_l c} \qquad \frac{t_2[x := t_1] \rightarrow_l c}{(\texttt{LET}\ x = t_1\ \texttt{IN}\ t_2 \rightarrow_l c)}$$

$$\texttt{REC}\ y.\ (\texttt{LAM}\ x.\ t) \rightarrow_l \texttt{LAM}\ x.\ t\,[y := \texttt{REC}\ y.\ (\texttt{LAM}\ x.\ t)]$$

As can be expected, the rule for canonical forms expresses that canonical forms evaluate to themselves. A key rule is that for the evaluation of applications: the evaluation of an application proceeds by the substitution of the argument into the function body; the treatment of the LET $x = t_1$ IN t_2 is analogously. The rule for recursive definitions unfolds the recursion REC y. (LAM x. t) once, leading immediately to an abstraction LAM x. $t\,[y :=$ REC y. (LAM x. t)], and so a canonical form.

3.3 The Denotational Semantics of MiniML

Our semantic interface to the "real" SML target language, the language MiniML, differs with two regards from MiniHaskell:

1. syntactically, all constant symbols in MiniML are followed by a prime, e.g. ZERO', ONE', in order to distinguish them from their counterparts in MiniHaskell. This is for the sake of presentation only.
2. semantically, the two constructs application and LET differ from their counterparts in MiniML.

In the sequel, we turn to the semantic issues. In most cases, the semantics of the strict and the lazy constructs are the same. This holds for basic operators like NOT' or SUC' as well as the abstraction, the conditional and the REC' construct. This justifies logical equations such as NOT' $\equiv$ NOT etc.

The crucial difference between the two languages is the strict application. As usual, its denotational definition in MiniML is given by:

```
$\rhd_s$ :: ($\alpha$::cpo $\Rrightarrow$ $\beta$::cpo) $\Rightarrow$ $\alpha$ $\Rightarrow$ $\beta$
      F $\rhd_s$ x $\equiv$ if x = $\bot$ then $\bot$
                else if F = $\bot$ then $\bot$ else $\lceil$F$\rceil$ x
```

The LET' construct is defined as usual in terms of abstraction and strict application (enforcing the evaluation of the let-expression prior to the evaluation of its body).

3.4 Eager Operational Semantics of MiniML

The rules for the strict evaluation relation $\rightarrow_s$ is derived analogously to the lazy one $\rightarrow_l$. Therefore, we can focus on the differences to MiniHaskell, which are just the rules for the different constructs for the strict application and `LET'`. In contrast to MiniHaskell, the arguments are first evaluated before performing a substitution:

$$\frac{t_1 \rightarrow_s \texttt{LAM'}\ x.\, t \quad t_2 \rightarrow_s c_2 \quad t[x := c_2] \rightarrow_s c}{t_1 \rhd_s t_2 \rightarrow_s c}$$

$$\frac{t_1 \rightarrow_s c_1 \quad t_2[x := c_1] \rightarrow_s c}{(\texttt{LET'}\ x = t_1\ \texttt{IN'}\ t_2) \rightarrow_s c}$$

This concludes our definition and validation of the two languages MiniHaskell and MiniML in terms of a (pre-existing) theory of denotational semantics. In the following, we turn to the semantic translation between these languages by means of derived rules.

4 The Semantic Translation

Between the considered languages, the translation of most language constructs is a trivial rewriting due to semantic equivalence. The challenge, however, is the translation of the lazy application to the strict one, and, on the larger scale, the translation of lazy *user-defined* definition constructs to one or more strict versions.

The default solution is well-known and simple: each expression is *delayed* i.e. converted into a closure, and all basic operations were enabled to apply its argument first to the unit-element () in order to *force* the argument closure and to produce an elementary value only when finally needed. Thus, any lazy application can be simulated by an strict one, provided that arguments of applications have been sufficiently delayed.

However, the default solution is fairly inefficient since it delays *any* computation. Therefore, optimizations are mandatory. The principle potentials for such optimizations are

1. the strictness of the function to be applied to an argument (i.e. the argument is used under all possible evaluations) or
2. the definedness of the argument (i.e. delaying is inherently unnecessary).

The concepts discussed above were made precise by a number of combinators which serve either as coding primitive (such as the combinator `delay` and `force`) or as combinators such as `forcify` that represents intermediate states of the translation. We will derive rules that allow to "push" `forcify` combinators throughout a program and thus perform the translation.

In the following, we present these concepts formally. First, we introduce the type constructor `del` for representing delayed, i.e. suspended values:

```
types
  α del = Unit ⇛ α
```

The `delay`-constructor and the corresponding suspension destructor `force` can both be defined completely in terms of our target language MiniML:

```
delay :: α::cpo ⇒ α del
          delay f ≡ (LAM' x. f)
force :: (α::cpo)del ⇒ α
          force f ≡ (f ▷ₛ UNIT')
```

Both combinators may remain in final program representations and are treated as primitive by the translation function convert.

It turns out that from these definitions the characteristic theorem

$$\texttt{force (delay } e\texttt{)} = e$$

can be derived as could be expected.

Now we define the `forcify` combinator that converts a function into its counterpart that deals with delayed values:

```
forcify :: (α ⇛ β::cpo) ⇒ (α del ⇛ β)
            forcify f ≡ LAM' x. ⌈f⌉(force x)
```

While the `delay` and `force` combinator can be understood as a primitive that can be coded by the converter, `forcify` is a combinator that is uncodable. It is only used internally in the source-to-source translation and has to disappear at the end.

The overall translation process consists of one language translation calculus and three side-calculi — `forcify`-propagation, strictness-reasoning and definedness reasoning, which consist, as mentioned, of derived rules.

4.1 Language Translation Calculus

As mentioned, all but two language constructs have equal semantics can therefore be converted straight-forward by a trivial rewrite rule such as

$$\texttt{SUC} = \texttt{SUC'}$$

The key translation rule for the lazy application has the following form:

$$(f \rhd_l a) = (\texttt{forcify } f) \rhd_s (\texttt{delay } a)$$

This rule states that a lazy application can always be converted into a strict one; the price is the delay of the argument and the necessary forcification of the function of the application. This rule represents the default translation rule, which is — since resulting in inefficient code — avoided whenever possible. The following two rules represent the optimized alternatives of the default scheme: a lazy application is identical with a strict application if its function is strict or if the argument is known to be defined and the function is not the totally undefined one:

$$\frac{\texttt{is\_strict } f}{(f \rhd_l a) = (f \rhd_s a)} \qquad \frac{\texttt{DEF } a \quad \texttt{DEF } f}{(f \rhd_l a) = (f \rhd_s a)}$$

For the LET'-construct, these three cases are analogously. The Isabelle proofs of these rules are not very hard but reveal a number of technicalities that are easily overlooked in paper-and-pencil proofs.

These optimized translation rules lead to side-calculi that attempt to infer the necessary information. One of them, the strictness calculus, will be discussed in the following subsections.

4.2 Forcification-Propagation Calculus

In the following, we turn to the key of the default translation to MiniML, the forcification-propagation. The base cases treat identities and constant abstractions as well as basic operators. For the latter, we can assume by construction that they are strict since we only used a particular pattern of their definition built upon strictify and HOL-functions.

$$\texttt{forcify}\ (\texttt{LAM}\ x.\ x) = \texttt{LAM}\ x.\ \texttt{force}\ x \qquad \texttt{forcify}\ (\texttt{LAM}\ x.\ c) = \texttt{LAM}\ x.\ c$$

$$\frac{f \equiv \texttt{strictify}\ g}{\texttt{forcify}\ (\texttt{LAM}\ x.\ f\ x) = \texttt{LAM}\ x.\ f\ (\texttt{force}\ x)}$$

$$\frac{\forall f.\ f \equiv \texttt{strictify}\ (\lambda x.\ \texttt{strictify}\,(g\ x))}{\texttt{forcify}\ (\texttt{LAM}\ x.\ f\ c\ x) = \texttt{LAM}\ x.\ f\ c\ (\texttt{force}\ x)}$$

The following rules describe the propagation over the core language constructs for application, abstraction and conditional:

$$\begin{aligned}&\texttt{forcify}\ (\texttt{LAM}\ x.\ ((f\ x) \rhd_l (g\ x))) = \\ &\qquad \texttt{LAM}\ x.\ (\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (f\ x)) \rhd_l x) \rhd_l \\ &\qquad\qquad (\texttt{forcify}\ (\texttt{LAM}\ x.\ (g\ x)) \rhd_l x)\ \ )\end{aligned}$$

$$\begin{aligned}&\texttt{forcify}\ (\texttt{LAM}\ x.\ (\texttt{LAM}\ y.\ (f\ x\ y))) = \\ &\qquad \texttt{LAM}\ x.\ \texttt{LAM}\ y.\ (\texttt{forcify}(\texttt{LAM}\ x.\ (f\ x\ y)) \rhd_l x)\end{aligned}$$

$$\begin{aligned}&\texttt{forcify}\ (\texttt{LAM}\ x.\ (\texttt{IF}\ c\ x\ \texttt{THEN}\ f\ x\ \texttt{ELSE}\ g\ x)) = \\ &\qquad \texttt{LAM}\ x.\ (\ \texttt{IF}\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (c\ x)) \rhd_l x) \\ &\qquad\qquad \texttt{THEN}\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (f\ x)) \rhd_l x) \\ &\qquad\qquad \texttt{ELSE}\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (g\ x)) \rhd_l x)\ \ )\end{aligned}$$

Of particular interest is also the rule for the propagation of forcification over the REC operator, which allows for the generation of recursive program definitions. In particular, applications like $\texttt{forcify}f$ are mapped to the reference f', where we assume that for f there has been the previous statement $\texttt{fun}\ f\ x = E$ which has been converted to the code-variant $\texttt{fun}\ f' = \texttt{forcify}\ (\texttt{LAM}\ x.\ E)$. It is automatically proven that this precompiled variant satisfies the property $(\texttt{forcify}\ f) \rhd_s x = (f' \rhd_s x)$ which justifies the mapping mentioned above. Thus, “forcified” calls to previously defined functions were mapped to calls of “forcified” definitions.

$$\begin{aligned}&\texttt{forcify}\ (\texttt{LAM}\ x.\ (\texttt{REC}\ (f\ x))) = \\ &\qquad \texttt{LAM}\ x.\ \texttt{REC}\ (\texttt{forcify}\ (\texttt{LAM}\ x.\ (f\ x)) \rhd_l x)\end{aligned}$$

For n-ary functions, analogous rules have to be derived. Moreover, since any function may be strict in the first argument, but not in the second, or vice versa, or non-strict in all arguments, there are $2^{(n+1)} - 1$ rules for potential forcified code variants for direct recursive functions.

4.3 Strictness Calculus

As already mentioned, optimized applications require the inference of strictness properties of function bodies. Again, the inference rules follow the cases of our programming language. The base cases treat the identity, the special case of the abstraction yielding $\perp$ and operations defined upon `strictify`.

$$\texttt{is\_strict}\ (\lambda x.\ x) \quad \texttt{is\_strict}\ (\lambda x.\ \perp)$$

$$\frac{f \equiv \texttt{strictify}\ g}{\texttt{is\_strict}\ f} \quad \frac{f \equiv \texttt{strictify}\ (\lambda x.\ \texttt{strictify}\ (g\ x))}{\texttt{is\_strict}\ (f\ c)}$$

Note, that the case for the lambda abstraction is omitted since

$$\texttt{is\_strict}\ (\lambda x.\ \texttt{LAM}\ y.\ (E\ x\ y))$$

simply does not hold: recall that a closure is a canonical form, hence a value different from $\perp$.

Since we suggest a source-to-source translation scheme, the calculus over strictness must cope with terms in which both strict and lazy applications may occur. Therefore, rules for both cases are needed. The inference reduces the applications to semantic functions and substitutes their denotation into it; in the case of the strict application, the argument must be strict in itself:

$$\frac{\texttt{is\_strict}\ (\lambda x.\ \lceil f\ x \rceil\ (a\ x))}{\texttt{is\_strict}\ (\lambda x.\ ((f\ x) \rhd_l (a\ x)))}$$

$$\frac{\texttt{is\_strict}\ (\lambda x.\ \lceil f\ x \rceil\ (a\ x)) \quad \texttt{is\_strict}\ (\lambda x.\ (a\ x))}{\texttt{is\_strict}\ (\lambda x.\ ((f\ x) \rhd_s (a\ x)))}$$

Note that the computation of the semantic functions $\lceil f\ x \rceil$ requires an own (trivial) side-calculus allowing to "push" $\lceil \_ \rceil$ inside; this side-calculus is not presented here.

With respect to the conditional, one gets two cases to establish strictness of the overall construct: either the condition is strict in x or both branches:

$$\frac{\texttt{is\_strict}\ f}{\texttt{is\_strict}\ (\lambda x.\ (\texttt{IF}\ (f\ x)\ \texttt{THEN}\ (g\ x)\ \texttt{ELSE}\ (h\ x)))}$$

$$\frac{\texttt{is\_strict}\ g \quad \texttt{is\_strict}\ h}{\texttt{is\_strict}\ (\lambda x.\ (\texttt{IF}\ (f\ x)\ \texttt{THEN}\ (g\ x)\ \texttt{ELSE}\ (h\ x)))}$$

The most technical proofs of this paper are behind the rules for inferring strictness of recursive schemes and definition constructs. These schemes — which perform an implicit induction — are consequences of the bi-simulation briefly presented in Section 2.3:

$$\frac{\texttt{cont}\ F \quad \bigwedge H.\ \begin{array}{c}[\texttt{is\_strict}\ H]\\ \vdots\\ \texttt{is\_strict}\ (F\ H)\end{array}}{\texttt{is\_strict}\ (\texttt{REC}\ F)}$$

This rule performs (for the 1-ary recursive function) a kind of specialized fixpoint induction proof: If we can establish strictness of the body F provided that a function H replaced in the recursive call is strict, then the recursor REC F yields a function that is strict in its first argument. Note, that for the n-ary cases similar rules are needed that are omitted here.

5 Examples

The calculi are grouped into several sets of rules which were inserted in the Isabelle rewriter. As a result, several tactics are available that perform the translation phases fully automatically.

5.1 Example 1

As a first example, we define a function in MiniHaskell whose body consists of a 2-ary lambda abstraction which is strict in its second argument. Its first argument represents an undefined value $\perp$:

```
fun f y = (LAM a b. b) $\triangleright_l$ (DIV x ZERO) $\triangleright_l$ y;
```

The first translation phase is able to derive the strictness in the second argument and replaces the second lazy application by a strict one:

```
fun f y = (LAM a b. b) $\triangleright_l$ (DIV x ZERO) $\triangleright_s$ y;
```

The next translation phase replaces the remaining lazy application by our default translation. Recall that a lazy application can always be converted into a strict one by delaying the argument and forcifying the function of the application. Furthermore, a forcification-propagation is performed:

```
fun f y =
  LAM a b. (LAM a. b $\triangleright_s$ delay a) $\triangleright_s$
               delay (DIV x ZERO) $\triangleright_s$ y;
```

A one-to-one translation is performed by the following translation phase. Each MiniHaskell construct is replaced by its MiniML counterpart yielding a pure MiniML-program:

```
fun' f y =
  LAM' a b.(LAM' a.b $\triangleright_s$ delay a) $\triangleright_s$
               delay (DIV' x ZERO') $\triangleright_s$ y;
```

The final translation phase performs an optimization by reducing the MiniML-program to the identity:

```
fun' f y = y;
```

5.2 Example 2

As a second example, we define the factorial function in MiniHaskell representing a recursive function:

```
fun fac x =
  IF (x ^=^ ZERO) THEN ONE ELSE x * (fac $\rhd_l$ (x - ONE));
```

Here, the first translation phase deduces that the function fac is strict in its argument and replaces the lazy application in the recursive call by the strict one:

```
fun fac x =
  IF (x ^=^ ZERO) THEN ONE
                  ELSE x * (fac $\rhd_s$ (x - ONE));
```

Finally, the next phase replaces each MiniHaskell-construct by its corresponding MiniML-counterpart:

```
fun' fac x =
  IF' (EQ' x ZERO')
     THEN' ONE'
     ELSE' TIMES' x (fac $\rhd_s$ (MINUS' x ONE'));
```

6 Conclusion

We address a well-known compilation problem of functional programming. We embed the semantics of both languages into a theory of denotational semantics and derive — as a check of these definitions — the corresponding operational semantics of these languages. The resulting strict semantics can be compared with the semantics of SML [6] and recognized as its abstracted version.

Finally, we derived a couple of rewrite rules that describe the translation of both languages as a source-to-source translation, which is prototypically implemented as a tactic-based compiler finally yielding executable code in SML.

Since the proofs of the translation rules are surprisingly simple (with few exceptions that are interesting in themselves), our approach yields a testbed for the implementation of compilers also for richer languages. Furthermore, it is feasible to develop typical libraries such as lists and compile them with our tactic-based compiler once and for all. Further, our approach may also be relevant to boot-strapping schemes when developing a proven correct compiler.

6.1 Further Work

We see the following issues for an extention of our work:

1. *Extending* MiniHaskell: a richer language comprising Cartesian products or lazy data types would help, in particular for the generation of concrete code.
2. *Low level target language*: In principle, our approach can also be applied for the generation of machine-code or JAVA byte-code.

References

[1] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.

[2] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, Dec. 1992.

[3] S. Glesner. Using program checking to ensure the correctness of compiler implementations. *JUCS*, 9(3), 2003.

[4] G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine and compiler. Technical report, TUM, March 2003.

[5] T. Meyer and B. Wolff. Correct code-generation in a generic framework. In M. Aargaard, J. Harrison, and T. Schubert, editors, *TPHOLs 2000: Supplemental Proceedings*, OGI Technical Report CSE 00-009, pages 213–230. Oregon Graduate Institute, Portland, USA, July 2000.

[6] R. Milner, M. Tofte, and R. Harper, editors. *The Definition of Standard ML (revised).* MIT Press, 1997.

[7] O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[8] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[10] H. Tej and B. Wolff. A corrected failure-divergence model for csp in isabelle/hol. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of the FME 97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer Verlag, 1997.

[11] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1993.

[12] L. Zuck, A. Pnueli, Y. Fang, and B. G. B. A methodology for the translation validation of optimizingcompilers. *JUCS*, 9(3), 2003.

Optimizing Proof for Replay

Malcolm C. Newey[1], Aditi Barthwal[1], and Michael Norrish[2]

[1] The Australian National University
Department of Computer Science, Canberra ACT 0200, Australia
[2] NICTA, Canberra ACT 0200, Australia

Abstract. The product of interactive theorem proving in an LCF-style prover such as the HOL system is an ML script that can be replayed as required. This paper describes an extension to the standard HOL system architecture whereby such replays may be executed in less time than is currently the norm. The savings, however, depend on expensive decision procedures being able to capture, for any input, sufficient information about the case that it will serve as a valuable hint for subsequent repeats. A case study is described wherein a decision procedure for BDD construction was tweaked to record hints that allow replay of some HOL proofs to be done dramatically faster.

1 Introduction

Technically, a proof in the context of an LCF-style theorem prover is just some sequence of primitive inferences that result in the creation of an appropriate object of abstract datatype *theorem*. Since this sequence (for interesting applications such as program verification and serious mathematics) is too long to be saved (let alone inspected), the ML commands that were used to create that proof, are used as a proxy for it. That is, when we say a proof is replayed it is this script that is executed again.

There are three important situations where proof replay occurs.

- A script is often developed iteratively over many sessions and its structure tends to evolve as proofs get improved and as new lemmas are found. With each change to the foundations of a theory and each generalization of theorems the script as it stands will need to be replayed. Moreover, the user is likely to start each session, by re-executing the script that has been developed so far.
- When any user does an installation of HOL (say) the libraries must be created afresh by executing a large collection of such scripts.
- When a proof is received in a proof-carrying-code situation it is checked; the whole point of such a proof is that it will be replayed by every 'customer'.

In each of these cases the time to replay a script is an important consideration for the user. A speed-up of just one order of magnitude makes a huge difference to someone waiting - it slashes times expressed in minutes to times in seconds (also hours to minutes and days to hours). In fact, we have found even greater factors can apply.

Although many (if not most) of the ML commands in a proof script will inevitably take the same time during replay, interactive theorem proving in many problem domains of interest depends on decision procedures that have exponential complexity. Typically it is calls on such decision procedures that will dominate replay time. Since one of the obstacles to more widespread adoption of formal methods is the ready applicability of theorem-proving, we can expect new and complex theories to be developed, with even greater dependence on efficient automation.

A key observation that leads to the suggestion that proofs can be optimized for replay is that they often involve search in a massive space before any actual formal inferences need be done. If the results of this search can be captured in some compact way then that same search need not be done next time around.

As an illustrative example, consider the case where a large search space is covered by a binary tree that is searched breadth first to some depth, d, before a suitable node is found and suppose the proof to be constructed is proportional to d (the length of the path back to the root). Further, suppose the cost of search is just one microsecond per node visited whereas the cost of constructing proof is 10 millisecond per node on the path back to the root. Table 1 shows the improvement we could expect with optimized replay, assuming only that the path which is discovered can be captured as a reasonably small string of text. In this case unoptimized replay time will be $(2^d + 10^4 \times d)$ microseconds.

Table 1. Binary search example – replay times

Depth (d)	Unoptimized replay	Optimized replay
10	11 msec	10 msec
20	1.05 secs	20 msec
30	18 mins	30 msec
40	30 hrs	40 msec

One could argue that the situation captured in the last line of Table 1 is most unlikely to occur in practice because the user is likely to factor the problem first. However one can easily imagine a script with several dozen invocations of this decision procedure where the search depths were 20 or more.

One lesson to be extracted from this example is that with such speed-ups feasible with exponential decision procedures, breadth-first search offers greater possibilities for dramatic optimizations.

2 The Architecture

In this section we discuss an extension of the standard HOL[3] system architecture. The ideas almost certainly apply to other LCF-style interactive theorem provers but it is left to the reader to adapt the mechanism appropriately.

The aim is to support decision procedures written in the style of a *memo function*[1]. The big distinction is that a decision procedure cannot work by just remembering the theorem that it returned previously, since that would avoid the proof activity that is its raison d'être. Rather it should commit to memory information that will make redoing the proof of the theorem much faster.

Because a typical memo function stores previous mappings in local variables such information is lost when the enclosing session ends. We don't just need a persistent database either, since the replay may be done by a distinct user on a different platform at a different time.

The memory mechanism proposed in this paper that supports optimized replay of proof scripts is termed a *hints file.* Such a file will accompany each proof script that contains calls on a decision procedure that can use it. Here are the rules for this 'memory device':

- If there is no hints file (not even an empty one) then no attempt to optimize is appropriate.
- If the input for a call to a decision procedure matches an entry in the hints file that accompanies the proof script then there is useful information in that entry.
- Otherwise, the decision procedure may add a mapping to the hints file to allow for speedup when this case re-occurs during replay.

Naturally, when a proof is replayed (such as in the case of proof-carrying code) there will be a significant overhead in accessing the hints file so that it is incumbent on the author of a decision procedure to avoid this mechanism if expected savings are outweighed by the cost of finding the hint and reading it in.

Since it is possible that a single proof script may contain invocations of multiple decision procedures that utilize the hints file, it is imperative that the association of input with hint must be decorated with other information that identifies which decision procedure produced (and will reuse) the hint.

On the basis of the above discussion, the reader might well conclude that an associative memory is the natural mechanism for implementing the hints file. It is certainly one possibility but that level of generality may be overkill. Implementing a true associative memory as a sequential file requires an initial index (probably a hash table) and the ability to randomly access the string data that follows. Since the hints file gets accessed sequentially on replay, in tandem

[1] "A memo function remembers which arguments it has been called with and if called with the same arguments again it will simply return the result from its memory rather than recalculating it." – FOLDOC (Free On-Line Dictionary of Computing), URL: `www.etext.org/Quartz/computer/internet/dictionary.gz`

with the proof script, a scheme that keeps hints in the order that they will be needed should be more efficient.

In the experiment described in the next section a simple sequentially accessed text file was used as the hints file. Because the focus of the project was simply to measure savings on replay, a hints file was written on one run and read on the next one. If our aim is to ship a hints file with proof script in the case of proof carrying code, that model of operation can work. However, in other settings where this architecture might be used, proof scripts can evolve and we can get hints in the file that need to be garbage collected. This is clearly the subject of further design experiments.

3 BDD Construction – A Case Study

A BDD (Binary Decision Diagram) can be thought of as the dag representation of a truth table for a propositional formula. The two BDDs shown in Figure 1 both represent the same propositional formula, $(a \vee b) \wedge (c \vee d)$ but the order of the variables is different; it is $[a \rightarrow b \rightarrow c \rightarrow d]$ in the first but $[d \rightarrow a \rightarrow c \rightarrow b]$ in the second.

In every path though a BDD the variables appear in the same order; this is just as it is for all the rows of a truth table. This constraint is adopted so that every propositional formula has a canonical representation, for each variable ordering.

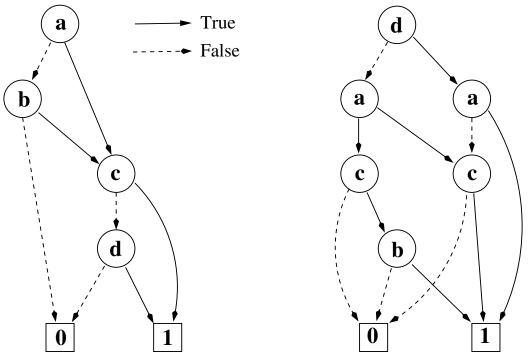


Fig. 1. Equivalent BDDs – same formula, different variable order

BDDs have wide application in hardware and software verification and consequently have been given serious attention in the world of higher order logic theorem proving. Harrison[4] made a three-way comparison of constructing a BDD deductively in HOL, using SML as the language and a C implementation. Gordon[2] interfaced the widely used BuDDy system to HOL to act as a BDD construction oracle; in this system theorems proved using BDDs therefore are tagged to indicate that they are dependent on this oracle.

A principal goal of any useful BDD construction algorithm is to choose a variable order that minimizes the number of nodes in the BDD. It has been observed that starting with a good order can reduce the running time of some BDD constructions by several orders of magnitude.

There are thus two obvious possibilities for the hint to speed up BDD construction on replay. We could use the variable order found on the first run as the starting point in replay or we could use the whole BDD produced. The variable order is textually compact; the BDD in the cases that matter is likely to be too large.

Barthwal, in her BSc honours thesis[1] explored in detail the option of using variable order as the hint for optimizing replay of the BDD construction algorithm as it might be invoked in decision procedures for tautology testing or equivalence of formulae.

3.1 Experimental Design

In this case study the BuDDy BDD engine was used to translate propositional formulae to their BDD representation in HOL. Comparisons were made of the performances of the algorithm in the absence of hints and on replay with a variable order specified.

The quantities measured were elapsed time and the number of nodes in the generated BDD. Various knobs that a BuDDy user can twiddle to aid performance, such as nodetable size and cache size, were set to recommended values.

There are two ways that BuDDy can be invoked to achieve an answer that hopefully has a good variable order. The build process can be invoked (possibly using a given order of variables) and the result can then be reduced using one of a number of reordering algorithms. Alternatively, reordering can be invoked automatically during the build process. The case study explored both options for the initial BDD build. Of course, on replay no reordering is done since the variable order from the hints file should be maintained.

The main choice that the user must make is that of reordering algorithm. In the experiment, all the five built-in ones were used to see if that choice might make a significant difference to the results. One of the five, providing a random reordering of variables, is part of BuDDy for testing purposes but provides a useful reference. The four smarter algorithms are called `Sift, Siftite, Win2` and `Win2ite.`

3.2 Test Data

John Harrison has produced a few little OCAML programs that generate families of tautologies. These programs, modified to produce HOL formulae, are described in Table 2. The upper limits on the number of variables were based on the practical limits of memory and time; runs exceeding 1 day were aborted.

Table 2. The Harrison Problem Set

Name	Description	Number of variables
Adder	Equivalence problems for Carry-Select vs Ripple-Carry adders	11-102
Mult	Equivalence problems for Carry-Select vs Ripple-Carry multipliers	16-138
Prime	Primality testing hardware	3-99
Ramsey	Formulae encoding assertion $R(s,t) < n$ for the Ramsey number $R(s,t)$	1-46

3.3 Highlights from Results

For any input formula, there is a baseline time of interest, τ_b; it is the time taken for the formula to be translated to a BDD without auto-reordering or subsequent reordering. Naturally, it costs more to do a subsequent reduction of the BDD produced by BDD Build. In fact, the penalty for doing such reordering is significant. For many big tests it was greater than $10 \times \tau_b$.

Of greater interest is the fact that, in most cases[2], auto-reorder also caused a slowdown. Again, there were cases where the slowdown was an order of magnitude.

The lesson to be drawn from these results is that there may be a substantial cost involved in optimizing the BDD in order to create a good hint.

In spite of the significant cost of finding a suitable variable ordering to record as a hint there were large savings to be made. The best stories to report are:

- the BDDs corresponding to the largest of the adders tested could be constructed in one hundredth the time if given the hint and if using the `Sift` or the `Siftite` reordering algorithms.
- the BDDs corresponding to the largest of the multipliers tested could be constructed in one fiftieth the time if given the hint and if using the `Sift` or the `Siftite` reordering.
- The shape of the curves suggests that greater savings are likely if the sizes of the adders were to grow.

On the other hand, there were disappointing results:

- constructing the BDDs corresponding to the primality, the non-primality and Ramsey tests actually took longer in the presence of the hints file.

Clearly the lesson here is that whether or not to use the hints file architecture depends on the domain in question.

[2] The only case in which auto-reorder sped up BDD construction was with the Adder tests involving large numbers of nodes.

4 Other Common Decision Procedures

Much of the progress made in theorem proving has depended on the discovery of decision procedures for areas of mathematics and computer science. Indeed, many of the HOL system libraries depend on them. Moreover, improving the uptake of formal methods in software engineering (where verifiability is greatly to be valued) is clearly dependent on achieving a degree of automation that enables reasoning about specifications, designs and programs to be done at a higher level than is currently the case. In this context, better automation really implies better theories of real world domains and better decision procedures for those theories.

Who can say whether our replay architecture will be a boon in new application domains but we are allowing for the possibility that it will. In this section we ask the question, for each of several decision procedures, "Could this function perform much better on some arguments in the presence of hints and will such improvement in speed, considering overheads, result in better performance on replay for proof scripts that utilize the hints file?"

4.1 Satisfiability

SAT solvers are very important decision procedures mainly because they can be applied in many different application areas. They make an ideal target for optimization because verifying that a substitution satisfies a formula takes time proportional to the length of that formula, whereas the complexity of finding such a mapping is exponential in the number of variables.

The hint that would be saved for replay is, of course, the substitution.

It might be argued that this decision procedure does not provide much support for the idea of the hints file since a user can take the output of the SAT solver and textually insert it into the script as a suitable witness. One downside of doing that is that it makes the proof script a bit less readable; but it also makes the proof more fragile in the sense that if changes are subsequently made to the script upstream then the proof script might break because the formula to be satisfied may have changed. Being more automatic, a system using a hints file can adapt to the change in the proof.

A bigger problem with the suggestion of explicitly including the output of the SAT solver in the proof script is that the satisfiability decision procedure may be being called by some other decision procedure. In this case, the hints file provides the memo-ization that is hard for the user to duplicate.

4.2 Tautology Checking

Although BDDs are sometimes the method of choice for checking tautologies[3], Stålmarck's algorithm as implemented by Harrison [5] has the advantage of generating an untagged theorem in HOL. Two matters that require experimentation

[3] especially if there are other uses for BDDs in the same project

are whether it is faster, even if the user is prepared to trust BuDDy, and whether it could be factored into a search phase and a proving phase connected by a pipe that carries a fairly small amount of information. The authors' guess is that the latter is not the case and so is not a candidate for proof optimization using the hints file mechanism.

4.3 First order logic

Decision procedures for first order logic are another very good fit for the hints architecture we propose. Such procedures typically spend a great deal of time exploring widely branching inference trees. Existing first order decision procedures in Isabelle and HOL (resolution, model elimination and tableau methods) all use the technique of performing this search outside the logic, performing only the successful chain of inferences in the trusted kernel. This provides a great deal of efficiency in itself. The hints architecture would further cache information to allow this chain to be recreated efficiently on replay.

One obvious implementation of the hints file would be to simply store a direct encoding of the primitive inferences themselves. Such a format need not be excessively verbose because the implementations of these decision procedures use a relatively narrow set of inferences, in very constrained ways. Indeed, because such an encoding is generated by the tool performing the search, the hints implementation may be little more than writing it to disk.

4.4 Arithmetic

Any method for finding satisfying assignments to systems of linear inequalities can clearly be augmented with a hints file where satisfying assignments exist. Additionally, when a method such as the Fourier-Motzkin technique finds that a system of inequalities is unsatisfiable, much of its work consists of enumerating consequences of the initial system. Only one of these consequences needs to be false for the system to be unsatisfiable, and again only the chain of inferences leading to this particular consequence needs to be performed by the trusted kernel.

As in the case of first order logic, existing implementations of these procedures in LCF-style systems perform the search for a refutation (or satisfying assignment) outside the logic, and then replay only those inferences absolutely necessary. Again, the hints architecture for this application could simply store an encoding of the proof.

5 Discussion

In this paper we have shown that in one significant case study the judicious use of a hints file can help a decision procedure cause invocations of itself on replay to be much faster than if the hints were unavailable. We have also suggested that in other cases the same dramatic increase in efficiency might be realized.

The fact that the provision of hints will make proof replay go faster prompts us to ask where the hints should go. Moreover, since an obvious place in many circumstances is the proof script itself, it begs the question of why go to the trouble of managing a companion file to the proof script.

We have argued above, in the case study, that there are problems of making the proof script less readable and less maintainable. These objections apply in general. One would hardly argue that the code of a program in a higher level language should be cluttered with automatically derivable optimization hints to make subsequent compilation more efficient.

5.1 Library Optimization

With current HOL system distribution, the replay of the proofs that make the theory library is not as time consuming for a user as it was a few years ago when it took a large fraction of a day. That is because the great increase in machine speed more than matches the relatively small increase in library complexity in that time. Although we are yet to see the demise of Moore's law, it would be defeatist to not allow for the possibility that a dramatic upturn in the acceptance of formal methods and the consequent expansion of the library support that should be shipped with HOL.

On the other hand, it doesn't actually follow that even with lots more invocations of decision procedures, the hints file architecture will provide such large savings in a library build. The question is complicated by the fact that in the future we expect to be able to prove the code for decision procedures to be correct, allowing theorem creation after search without requiring constructing a primitive proof. The elimination of the base proof construction obligation would make many library proof scripts more efficient (although the proofs of correctness of those proof procedures would have to be added to the library).

5.2 Program Verification

The higher order logic community perpetually believes that respectability for formal program verification is just around the corner. In fact, the growth of web technologies may engender acceptance of proof as the best form of guarantee to affix to code. In such a future world, the proofs sought will be ones that are inexpensive to transmit and check.

It's hard to even talk about what improvements would be made using self-conscious decision procedures in program verification without mounting a serious project in this area. On the other hand this is the area that optimizaton for proof carrying code depends on.

Homeier [6] has a verification condition generator that can be used to generate VCs in higher order logic. It seems like this tool presents an ideal opportunity to explore the extent to which optimization for replay can be expected to reduce the time taken to check a code-carried proof.

6 Conclusion

In this paper we have asserted that, at least in the case of the HOL system, extending the architecture to allow for hints files will enable faster replay of proof scripts. We suggest that the idea can also be adapted to other interactive theorem provers that might be part of the infrastructure for checking proof carrying code.

The empirical evidence for the claim was found in the many results produced in a systematic study of the BuDDy package as it is distributed with HOL. It was found that decision procedures that make use of BDDs can have a win if they use the hints file to record a 'good' variable ordering in limited circumstances:

- The size of the problem should not be too small or else the overhead of consulting a file is greater than the benefit that might flow from use of the hint.
- The decision procedure must be advised that the set of likely inputs are from a domain where savings are to had.
- The variable reordering algorithm adopted must be also appropriate to the domain.

The main conclusion that we draw at this point is that the idea is good but several more case studies need to be done to make the picture clearer and the case stronger.

References

1. Aditi Barthwal. Proof optimization for replay: A case study. Technical Report Honours Thesis, Computer Science Dept., The Australian National University, November 2004.
2. Mike Gordon. Reachability programming in hol98 using bdds. In Mark Aagaard and J. Harrison, editors, *13th International Conference on Theorem Proving in Higher Order Logics*, volume 1869 of *LNCS*, pages 179–196, portland, Oregon, USA, Aug 2000. Springer.
3. Mike Gordon and Tom Melham. *Introduction to HOL: a Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
4. John Harrison. Binary decision diagrams as a hol derived rule. *The Computer Journal*, 38(2):162–170, 1995.
5. John Harrison. Stålmarck's algorithm as a hol derived rule. In J. von Wright, J. Grundy, and J. Harrison, editors, *Ninth International Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *LNCS*, pages 221–234, Turku, Finland, Aug 1996. Springer.
6. Peter Homeier and David Martin. Mechanical verification of total correctness through diversion verification conditions. In J. Grundy and M. Newey, editors, *Eleventh International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 189–206, Canberra, Australia, Sep 1998. Springer.

A HOL Implementation of the ARM FP Coprocessor Programmer's Model

James Reynolds
jr291@cam.ac.uk
http://www.cl.cam.ac.uk/

University of Cambridge

Abstract. The ARM Vector Floating-point coprocessor is used by the ARM processor to perform floating-point arithmetic on single registers or vectors of registers. This paper details the development of a simulator written in HOL which uses the formally specified IEEE library created by John Harrison.

1 Introduction

This paper details the development of a *Higher Order Logic* model of the ARM Vector floating-point programmer's model. The model uses the HOL4 theorem proving environment to manipulate and prove theorems in higher order logic. In particular this system has methods to evaluate functions consistently with the axioms and rules defined for higher order logic, hence provided with a formal specification of desired behaviour this system can be used to create a 'Gold Standard' simulator for a processor.

1.1 The ARM VFP

The ARM processor is only capable of performing operations on integers in the range $[0, 2^{32})$, hence calculations involving non-integer rationals either require careful implementation in software or a specialised coprocessor. Implementation in software is much slower than implementation in dedicated hardware, so real-time applications will either have to be re-written using fixed-point arithmetic or make use of dedicated hardware, a floating-point coprocessor.

The *ARM Vector Floating-point* architecture is a coprocessor architecture designed to perform calculations on non-integer rationals on behalf on the ARM processor it is attached to. The VFP provides instructions to carry out arithmetic on a subset of rational numbers in the range $[2^{-149}, 2^{128})$ for *single precision values* and $[2^{-1074}, 2^{1024})$ for *double precision values.* The VFP Architecture also provides instructions which can perform operations between scalars and vectors, and vectors and vectors.

Coprocessor Communication

The ARM processor operates on a series of 32-bit long instructions, a subset of which are *floating-point* instructions, these cannot be executed on the ARM processor itself. Instead, these instructions raise an 'Undefined Instruction' exception, which is handled by the ARM processor by passing the instruction to the VFP coprocessor. In a VFP coprocessor implementation the coprocessor will have access to the parent processor's register file and the system memory.

1.2 Motivation

Creation of a 'Gold Standard' simulator from a formal specification has a number of uses: in creating the formal specification anything which is badly defined in the original specification will be easily spotted, and results obtained using the simulator are guaranteed to be correct. Generating results in this manner allows two very useful statements to be proved:

1. By running a program on the model we can prove that given a correct real world implementation of the architecture the programme produces the desired behaviour.
2. A real world implementation of the architecture may be compared against the model to show that it produces the correct results.

There already exists a model formulated in HOL of the ARM instruction set architecture [3] which defines the processor state space, consisting of registers and memory. The VFP coprocessor formulation in HOL was constructed as a function from a product of the ARM state space and the VFP state space and an instruction to a new product of state spaces. In this way it would be possible for the ARM processor model to call the VFP coprocessor model in the event of a floating-point instruction.

1.3 HOL

HOL [6] is an automated proof system for Higher Order Logic in which functions and data types may be defined and reasoned over. All theorems deduced using HOL must be be proved using well defined rules from a small set of axioms.

The HOL syntax is well-typed and incredibly rich, allowing universal, existential and unique existential quantification, lambda abstraction, Hilbert's choice operator, predicate sets and more. This allows the easy definition of a `VFP` function that calculates a new state given a previous state and an instruction.

Theorems in HOL are written as:

```
{Assumptions} |- Conclusion : thm
```

The assumption set is omitted when none are present.

Sequents

Theorems in HOL are manipulated using sequents; manipulation rules which maintain the consistency of theorems. For example, the conclusion $P \wedge Q$ is only provable from the assumptions if P and Q may be proved from the assumptions separately:

$$\frac{A \vdash P \quad A \vdash Q}{A \vdash P \wedge Q} \ .$$

Sequents can combined to create *tactics* and proof tools.

Evaluation

Evaluation in HOL takes place by adding functions to a *compset*, these functions are pattern matched to the sub-terms in the expression currently being evaluated and if they match a rewrite is performed. For instance:

```
compset := { |- a + a = 2 * a : thm }

EVAL ``5 + 5``;
val it  = |- 5 + 5 = 2 * 5 : thm
```

If no such theorem exists within the *compset*, then the sub-term cannot be reduced.

1.4 IEEE Floating-point

The IEEE [4] have informally specified a standard for floating-point arithmetic, which defines the results of floating-point operations; however, it does not specify how these results can be calculated.

Every floating-point number represents a rational number, so the IEEE specification defines the result of an operation to be the *'closest' floating point representation to the result of the calculation using the corresponding rationals.* John Harrison [1] created a formally specified version of this standard which uses the axiom of choice to formally define this relation.

The Axiom of Choice

The axiom of choice, is used to 'select' values which have the desired property, for example division could be written as:

$$a \div b = (\varepsilon x.\ a = x * b)$$

where ε is the Hilbert choice operator.

The axiom of choice is a controversial axiom for a number of reasons. For example, in the previous Equation, how can a value of x be selected if a is non-zero and b is zero? When creating a simulator, the axiom of choice causes problems as it allows equations to be formalized without the need to define how results are obtained.

The result of this is that a formal specification which uses the axiom of choice cannot be directly used to create a simulator, instead an algorithm to find the value selected must be formulated and proved to select the correct value.

2 The Programmer's Model [5]

The programmer's model, as mentioned in Section 1.2, provides a 'Gold Standard' operation for an architecture. The programmer's model for the VFP architecture defines how the results of a floating-point calculation relate to the real number calculation, and the instructions that may be used to perform these calculations.

2.1 Floating-point Formats

This Section outlines the general floating-point formats as defined by the IEEE 754 standard, specific instantiations of which are used in the VFP architecture, and details how operations are performed on numbers in these formats. In addition it describes details of these formats which are specific to the ARM Floating-Point coprocessor that had been left open in the standard.

Floating-point Numbers

The IEEE floating-point library defines a floating-point number format as a pair of `num` values; the 'width' of memory to store the fraction and to store the exponent. Floating point numbers within this format are therefore elements of the set:

$$\mathbb{FP}(E,F) \triangleq \{(s,e,f) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mid s \in \{0,1\} \wedge 0 \leq e < E \wedge 0 < f < 2^F\}$$

where E is the exponent width and F the fraction width. Values with $e = E$ represent special values: infinities or NaNs.

Floating point numbers within this set represent real numbers as follows:

$$\mathit{exponent} = 0 \Rightarrow \mathit{val} = -1^{\mathit{sign}} \times 2^{-(\mathit{bias}-1)} \times (0.\mathit{fraction}) \qquad (1)$$

$$0 < \mathit{exponent} \leq E \Rightarrow \mathit{val} = -1^{\mathit{sign}} \times 2^{\mathit{exponent}-\mathit{bias}} \times (1.\mathit{fraction}) \qquad (2)$$

where bias is defined as:

$$\mathit{bias} = 2^{E-1} - 1 \ .$$

This implementation presents two values for zero: plus zero, `(0,0,0)` and minus zero `(1,0,0)`. These two values behave identically in all but a few exceptional instructions, such as the sign of the infinity resulting from a *Division by Zero* exception.

HOL Representation

Values are stored in HOL as three tuples and formats as two tuples, with a predicate defined to constrain tuples to the format:

$$is\_valid\ (E, F)\ (s, e, f) \triangleq s < 2 \wedge e < 2^E \wedge f < 2^F \ .$$

Functions have also been defined to calculate values for the bias, the maximum exponent and other important format constants. The library then defines a `valof` function for converting a floating-point number to the real number it represents.

As mentioned in Section 1.4, the IEEE standard requires the result of a floating-point computation to be the closest floating-point representation, subject to rounding conditions, of the real number result of the calculation. In HOL this is defined using the axiom of choice:

```
round Format mode x ≜ εa.
        is finite and satisfies rounding conditions ∧
        ∀b. b is finite and satisfies rounding conditions ⇒
            $abs(valof\ Format\ a - result) \leq$
            $abs(valof\ Format\ b - result)$
```

where *finite* is a predicate that is true iff the floating-point number is a normal value, a denormal value, or a zero.

This may be read as: *select the floating-point number that satisfies the rounding conditions, and all other values that satisfy the rounding conditions are further away (produce a greater rounding error) from x.*

Special Values

Values with the exponent equal to the maximum exponent are special reserved values:

Exponent	**Fraction**	**Meaning**
0	$0 < f < 2^F$	Denormal numbers
$0 < e < E$	$0 < f < 2^F$	Normal numbers
E	0	$\pm\infty$ Depending on sign
E	$0 \leq f < 2^{F-1}$ (bit F - 1 not set)	Signaling NaNs
E	$2^{F-1} \leq f < 2^F$ (bit F - 1 set)	Quiet NaNs

The IEEE 754 standard defines the resulting NaN of an operation in which all operands are NaNs to be one of the operands. Unfortunately the IEEE library defines operations which produce an NaN to produce any NaN. Therefore all floating-point operations were wrapped with a function that selects the NaN according to the rules defined in the *ARM Vector floating-point* programmer's model [5].

2.2 Rounding Modes

The IEEE 754 standard defines four rounding modes; Round to Nearest (RN), Round towards Plus Infinity (RP), Round towards Minus Infinity (RM) and Round towards Zero (RZ). These are defined in the following fashion:

RN Minimises abs(rounding error), if two such minima exist then select the item with an even fraction component.
RP Minimises abs(rounding error) subject to the condition: $0 \leq$ rounding error If the result is greater than the largest possible floating-point number then the result is $+\infty$.
RM Minimises abs(rounding error) subject to the condition: rounding error ≤ 0 If the result is less than the smallest possible floating-point number then the result is $+\infty$.
RZ Minimises abs(rounding error) subject to the condition: abs(rounded result) $\leq$ abs(exact result).

These are defined using a function which selects a floating-point number which minimises the rounding error subject to an arbitrary condition:

$$\begin{array}{l} \text{is\_closest X x P a} \triangleq \\ \quad \text{P a} \wedge \\ \qquad (\forall \text{ b. P b} \Rightarrow \text{abs (valof X a - x)} \leq \text{abs (valof X b - x)}) \end{array}$$

$$\begin{array}{l} \text{closest X x P Q} \triangleq \\ \quad \varepsilon a.\ \text{is\_closest X x P a} \\ \qquad \wedge\ ((\exists b.\ \text{is\_closest X x P b} \wedge \text{Q b}) \Rightarrow \text{Q a}) \end{array}$$

where P is the condition predicate and Q is the selection predicate in the case of there existing two floating-point numbers that satisfy the condition. The latter of these may be read as: *select the number that is closest, and if there exists a number that is closest and satisfies Q then the selected number satisfies Q as well.*

The rounding function then uses the following selections along with some logic to ensure that the conditions outside the range of floating-point numbers are met:

Mode:	**P:**	**Q:**
RN	$(\lambda a.$ finite X a)	$(\lambda a.$ EVEN (fraction a))
RP	$(\lambda a.$ finite X a $\wedge$ valof X a $\leq$ x)	(λ a. T)
RM	$(\lambda a.$ finite X a $\wedge$ x $\leq$ valof X a)	(λ a. T)
RZ	$(\lambda a.$ finite X a $\wedge$ abs (valof X a) $\leq$ abs x)	(λ a. T)

2.3 Format of the General-purpose Registers

A VFP implementation contains 32 general purpose registers which are capable of holding a 32-bit word, hence a single precision, `(8,23)`, value. These registers are named S0–S31.

The double precision, `(11,52)`, registers overlap the single precision registers in memory using two single precision registers to hold a double precision register, for example: D0 is stored in S0 and S1, D1 is stored in S2 and S3 and so on. The programmer's model does not, however, define how a double precision register is stored in the two corresponding single precision registers.

As this system aims to be implementation independent this means that the method of storing and retrieving floating-point registers should enable us to prove:

$$\forall reg\ s\ data.\ (readsingle(writesingle\ s\ data)\ s = data)$$

where s and d are the single register to write to and the data to store in it respectively. However we should never be able to prove:

$$\forall reg\ d\ data\ f.\ \exists f.\ (f(readdouble(writesingle\ s\ data)\ d) = data)$$

where s and d are single and double floating point registers which overlap.

The VFP defines two instructions which must store then load a number of double precision registers regardless of what they contain in at most $2N + 1$ words. The VFP defines a standard format for storing registers with tags, in order for the above theorems to hold and correct memory operation to take place this format (or an equivalent implementation defined format) must be used.

2.4 Integer - Floating Point Conversions

Each single precision register may hold a 32-bit integer, the register contents are identical for a 32-bit integer and a single precision value represented by the same word. This means that a direct copy of a single precision register to memory can be used to store either integers or single precision values.

Conversion uses the same rounding technique as floating-point operations.

2.5 Vector Operations

Vector operations are constructed by defining the system parameters *length* and *stride* held in a special register. Instructions are then performed on a vector of registers, *len* long, moving on *stride* registers at a time, and starting from the register given by the instruction. Registers from the first register bank, S0–S7 for single precision registers and D0–D3 for double precision, are considered as *scaler* unless the instruction forces them to be otherwise.

Singles :	**Doubles** :
$v[0] = S$ $v[1] = (S + stride) mod\ 8$ ... $v[len] = (S + len \times stride) mod\ 8$	$v[0] = D$ $v[1] = (D + len \times stride) mod\ 4$ ... $v[len] = (D + len \times stride) mod\ 4$

Instructions come in three forms: one operand, two operands or three operands. In all cases, if the destination register lies within a 'scalar' register bank then the operation is automatically a scalar operation:

Destination:	**Monadic:**	**Non-monadic**		
Scalar	= Op(Scalar)	= Scalar	⊗ Scalar	⊕ Scalar
Vector [0]	= Op(ScalarA)	= Vector[0]	⊗ Scalar	⊕ Vector[0]
...	...	...		
Vector [len]	= Op(ScalarA)	= Vector[len]	⊗ Scalar	⊕ Vector[len]
Vector [0]	= Op(Vector[0])	= Vector[0]	⊗ Vector[0]	⊕ Vector[0]
...	...	...		
Vector [len]	= Op(Vector[len])	= Vector[len]	⊗ Vector[len]	⊕ Vector[len]

For non-monadic operations the destination register is not a 'scalar' register then the first source operand is automatically considered a vector *regardless of which bank it is in.*

Unpredictable Results

There are various cases when the VFP considers the results of an operation to be unpredictable; in cases where vectors overlap, such as *VectorA[i] = VectorB[j]*, then the result is unpredictable unless the start registers of the two vectors are equal. Similarly, if the a vector overlaps a scalar then the results are considered unpredictable.

Unpredictable results should be coded in such a way that they may be represented in calculations but very little can be proved about them for instance:

$$\vdash (u * 0 = 0) \wedge (0 * u = u) \text{ but not } \vdash (u * a = b)$$

where u is an unpredictable value.

2.6 Examples

The model is capable of calculating floating-point arithmetic and using the results of these to calculate the results of floating point instructions.

```
|- EVAL ``fadd (8,23) To_nearest (0,31,1048575) (0,31,525288)``;
> val it =
    |- fadd (8,23) To_nearest (0,31,1048575) (0,31,525288)
        = (0,31,786432) : thm

|- EVAL ``fmul (8,23) To_nearest (0,127,500) (0,128,715)``;
> val it =
    |- fmul (8,23) To_nearest (0,127,500) (0,128,715)
        = (0,128,1215) : thm
```

```
|- EVAL ``fdiv (8,23) float_To_zero (0,127,22) (0,127,7)``;
> val it =
    |- fdiv (8,23) float_To_zero (0,127,22) (0,127,7)
        = (0,127,14) : thm
```

Operations are completed in reasonable times, however for simulating larger programs more optimisation will be required:

a:	b:	result:	Time(s):
(0,15,524288)+	(0,15,262144)=	(0,16,393216)	9.3
(0,31,1048575)+	(0,31,525288)=	(0,31,786432)	11.7
(0,62,2097151)+	(0,62,1048575)=	(0,63,1572863)	9.7
(0,125,4194303)+	(0,125,2097151)=	(0,126,3145727)	22.0
(0,251,8388607)+	(0,251,4194303)=	(0,252,6291455)	87.1

3 Implementation

As mentioned earlier in Section 1.3 the model was implemented as a function from a system state and an instruction, to another system state. As the coprocessor requires access to the ARM register file, the memory [3] and the coprocessor registers all of these things must be included in the state:

```
|- FP_State: ARM mem reg psr, fp_reg, fp_status_reg
```

The model is a function with the type:

```
|- Processor: FP_State -> Instruction -> FP_State
```

Functions were created to perform loading and storing of results into registers, memory access, transfer of information between the VFP and the ARM processor and functions that use the IEEE library to calculate results.

3.1 Unpredictable Results and Register Storage

As mentioned in Section 2.5 and 2.3, the results of many operations will be ‘unpredictable’, and nothing should be provable about loading a single precision value from a register holding a double value. In Section 1.4 the axiom of choice was introduced, and it was stated that the axiom of choice can only be ‘resolved’ if an algorithm can be defined to implement the choice. There clearly exist no such values, and hence no such algorithm, in situations such as:

$$\varepsilon x.\, \frac{0}{x} = 1 \ .$$

Furthermore, there is no such algorithm when the choice is entirely arbitrary, so it would not be possible to resolve the axiom of choice in a case such as:

$$\varepsilon x.\, T$$

As such this value will pass unchanged through all equations and execution will be forced to stop when an attempt is made to evaluate an equation using it, since, as mentioned in Section 1.3, evaluation uses theorems supplied to the *compset*, and no such theorem can be proved.

3.2 Vector Arithmetic

Vector arithmetic was implemented by way of 'lifting' the standard arithmetic operations and register store operations to perform these operations on lists of numbers.

Vectors of registers were formulated as functions from numbers to registers, and operations as functions from vectors to vectors. In the case of two operand vector operations:

$$\begin{aligned} v1 \otimes v2 &\triangleq \lambda n.\,(v1\,n) \otimes (v2\,n) \\ Write\,v\,r &\triangleq \lambda n.\,write(v\,n)\ r \\ Add\ va\ vb &\triangleq\ (write\,vd\,(va + vb)) \end{aligned}$$

This results in an easy to read formulation of all operations, however six separate functions were required, as operations can operate on singles or doubles and take one, two or three arguments.

4 IEEE Standard Conformance

As the floating-point operations defined in the IEEE library make use of the axiom of choice it is not possible to execute them without providing an algorithm to calculate the correct value. There are a number of ways in which this can be achieved:

1. Provide algorithms for add, multiply etc... and prove they are correct.
2. Provide a general algorithm for rounding any real number and prove this is correct.
3. Provide a general algorithm but allow it to be optimised in the case of particular operations.

The final option was chosen here as it allows a single calculation of a real number value and optimisations to be added as and when they are required.

4.1 The Rounding Calculation

This was achieved by defining a `REAL2FLOAT` function and a `NEXTFLOAT` function, the former calculates the closest floating-point number to a real number in 'Round to Zero' mode, and the latter calculates the next floating-point number away from zero, if such a number exists.

Considering only the `s = 0` case Equations 1 and 2 may be reversed to give the following Equations:

$$\begin{aligned} 2^{exponent} &= val \times \frac{2^{biasX+fracwidthX}}{2^{fracwidthX} + fraction} \\ 0 < exponent < E \Rightarrow fraction &= val \times \frac{2^{biasX+fracwidthX}}{2^{exponent}} - 2^{fracwidthX} \\ exponent = 0 \Rightarrow fraction &= val \times 2^{biasX+fracwidthX-2} \end{aligned}$$

If δ is the smallest positive non-zero floating-point number (2^{-149} for single precision), then the rational value:

$$\left\lfloor \frac{val}{\delta} \right\rfloor \times \delta$$

will have the same floating-point representation in 'Round to Zero' mode as the real number it was derived from.

Functions to calculate the logarithm to the base 2 of a number, LN, and to round real numbers to natural numbers, $REAL2NUM$, were defined:

$$\begin{aligned} 0 < x \Rightarrow & \quad x\ DIV\ 2 < 2^{LN\ x} \leq x \\ 0 \leq x \Rightarrow & \ REAL2NUM(x) \leq x \leq REAL2NUM(x) + 1 \end{aligned}$$

These definitions were then used to create a `REAL2FLOAT` function that would round down a real number to a floating-point representation.

4.2 Round to Zero Proof

To prove that this function rounds to zero the following three conditions were proved as well as the $x < 0$ counterparts:

$$\begin{aligned} &\forall X x.\ -thresholdX \leq x \leq thresholdX \\ &\qquad \Rightarrow finite\ X\ (\texttt{REAL2FLOAT}\ X\ x) \end{aligned}$$

$$\forall X x.\ 0 < x \Rightarrow valofX(\texttt{REAL2FLOAT}\ X\ x) \leq x$$

$$\begin{aligned} &\forall X\ x\ b.\ 0 < x \land finite\ X\ b \land valofXb \leq x \\ &\Rightarrow valofXb \leq valofX(\texttt{REAL2FLOAT}\ X\ x) \end{aligned}$$

The first and second conditions were proved using repeated simplification [2] and application of the following transitive split theorems defined on the reals and natural numbers:

$$\forall ac.\ a < c = \exists b.\ a \leq b \wedge b < c$$
$$\forall ac.\ a < c = \exists b.\ a < b \wedge b \leq c$$
$$\forall ac.\ a \leq c = \exists b.\ a \leq b \wedge b \leq c$$

These theorems allowed the goal $a < c$ to be split into two goals, with a specified intermediary term.

The third was proved by forming a contradiction of the following:

$$\vdash valof\ X\ b <= r \wedge valof\ X\ (\texttt{REAL2FLOAT}\ X\ r) < valof\ X\ b$$

Then using the fact that an inequality between the value of two floating point numbers may be written as an inequality on their components:

$$\begin{aligned}
&\vdash \forall Xab.(sign\ a = 0) \wedge (sign\ b = 0) \Rightarrow\\
&\quad valofXa < valofXb =\\
&\qquad exponenta < exponentb \vee\\
&\qquad (exponenta = exponentb) \wedge fractiona < fractionb
\end{aligned}$$

The `NEXTFLOAT` function was defined by checking whether incrementing the fraction would lead to another finite floating-point number and incrementing the exponent otherwise. In the case of the maximum or minimum floating point number being reached the same number was retained. By using this method it was easily provable that the function always returned a finite number given a finite number.

The function `NEXTFLOAT X (REAL2FLOAT X x)` was then proved to be the minimum floating-point representation to a real number whose absolute value was larger than that of the real number. This was achieved by showing the following:

$$\vdash \neg(x = top\_float\ X) \Rightarrow valof\ X\ x < valof\ X(\texttt{NEXTFLOAT}\ X\ x)$$

Hence the value of this function is strictly greater than that of `REAL2FLOAT` and because this is the minimum representation less than or equal to the real number this value must be greater than or equal to the real number.

It was then proved that this function was the minimum such value by showing that any other floating-point representation would have to be greater than `REAL2FLOAT`, and `NEXTFLOAT` was, by definition, the minimum such representation.

The Final Rounding Function

Using the functions `REAL2FLOAT` and `NEXTFLOAT` a rounding function `ROUND` was created which followed the IEEE library definition. The function simply tests which function satisfies the criteria given and returns that function.

4.3 Uniqueness Proofs

In order to prove that `ROUND` was equivalent to the IEEE rounding function, $\varepsilon\, x.\, closest\ X\ x\ P\ Q$, the following had to be proved for each rounding mode and corresponding predicates P and Q:

$$\begin{array}{rl} \forall X\ x. & closest\ X\ (\texttt{ROUND}\ X\ x\ mode)\ P\ Q \\ \forall X\ x\ y. & (closest\ X\ x\ P\ Q) \wedge (closest\ X\ y\ P\ Q) \Rightarrow (x = y) \end{array}$$

These could then be resolved with the `SELECT_UNIQUE` theorem to show that the two were equivalent.

The latter proof presented a number of problems however, as the rounding function is not unique, as the standard defines both +0 and -0. For this reason the functions were proved over 12 separate regions (here δ is the smallest positive non-zero floating-point number as mentioned in Section 4.1):

Rounding :	**Unique negative**	**Zero**	**Unique Positive**
$Nearest$	$r < -\delta \div 2$	$-\delta \div 2 \leq r \leq \delta \div 2$	$\delta \div 2 < r$
$Zero$	$r \leq -\delta$	$-\delta < r < \delta$	$\delta \leq r$
$+\infty$	$r \leq -\delta$	$-\delta < r \leq 0$	$0 < r$
$-\infty$	$r < 0$	$0 \leq r < \delta$	$\delta \leq r$

The `closest` function cannot be proved to be unique in the regions around zero, for this reason the more simple goal `is_zero X (closest X x P Q)` was proved. Fortunately, all the floating-point functions are defined with an enclosing `zerosign` function, designed to select the correct sign for any resulting zero.

Using the 12 proofs the following could be proved:

```
|- !X x m. zerosign X s (ROUND X x m) = zerosign X s (round X x m)
```

where `ROUND` is the rounding function defined earlier and `round` is the rounding function using the `closest` definition.

Limits on Formats

In order to complete many of the proofs, it was necessary to place constraints upon the values which the fraction width and the exponent width can take. These limits however impose few practical limitations as merely require the fraction width to be non-zero and the bias to be greater than that.

These constraints are met by both single and double precision formats, as well as many of the extended formats.

4.4 Conversion to a Single Function With Constraints

At this point it is possible to evaluate a floating-point operation by evaluating the operation on the corresponding real numbers and rounding the result. Unfortunately this approach is incredibly inefficient as numbers will have to be multiplied by $2^{fracwidthX+biasX}$ or 2^{150} for single precision values as the table below shows:

a: b: result:	**Time(s)**
(0,15,524288)+ (0,15,262144)=(0,16,393216)	44.9
(0,31,1048575)+ (0,31,525288)=(0,31,786432)	53.3
(0,62,2097151)+ (0,62,1048575)=(0,63,1572863)	45.4
(0,125,4194303)+(0,125,2097151)=(0,126,3145727)	62.7
(0,251,8388607)+(0,251,4194303)=(0,252,6291455)	201.4

For this reason a `generic_float_op` function was defined, such that given functions which match certain criteria it could be proved to be equal to the rounding function:

```
|- !data real2float lethreshold nearest isexact r m s.
      P (real2float) /\ Q (lethreshold) /\ ...
      ==>
         (zerosign X s (round X m r) =
                zerosign X s
                  (generic_float_op X m r
                     (data,real2float,lethreshold,
                      nearest,isexact))))
```

The use of the `data` member allows functions to be passed the operands of an operation. In the case of add, the function `add_data` calculates the sum of the floating-point numbers prior to division by $2^{bias+fracwidth}$ which is used in the remaining functions:

```
|- (zerosign X s (round X m (valof X a + valof X b)) =
         zerosign X s
             (generic_float_op X m (valof X a + valof X b)
                   (add_data X a b,add_real2float,
                    add_lethreshold,add_nearest,
                    add_isexact)))
```

The only deviation from the original rounding function is in the calculation of whether the nearest float towards zero or away from zero should be used. Instead of calculating the difference in rounding error:

$$abs(valof\ X\ a - x) < abs(valof\ X\ b - x)$$

A calculation based on which side of the mid-point x lies was used, as finding the sum of two floating-point numbers is a problem that would have to be solved in creating functions for addition.

It is relatively simple to prove that for $0 < x$ and $a < b$ the previous equation is equal to:

$$\frac{(valof\ X\ a + valof\ X\ b)}{2} < x$$

5 Implementation of Optimisations

The add and multiply operations have been successfully optimised, such that operations take less than a second, rather than minutes. The square root function required slightly more work, but is on the verge of completion.

Calculation time is highly dependent upon the magnitude of intermediate results as the following table shows:

Calculation	Inference steps	Time (s)
2^{100}	52434	0.80
2^{200}	204449	3.05
2^{300}	456971	6.95
2^{400}	808464	12.2
2^{500}	1261328	19.1

There are two steps that are taken to keep the magnitude of calculations as low as possible, the first step was to create functions that calculate fractions and exponents without involving unnecessarily large powers and the second was to create rewrite rules that reduce the magnitude of the intermediate results of a calculation when possible.

This two step method ensures that the function definitions remain easy to reason about, but simulation runs quickly as the rewrite rules ensure that calculation is performed in a more efficient manner.

5.1 Calculation functions

Using addition as an example it is easy to see that by creating a function that uses numbers of a lower magnitude calculation is much faster.

Addition was optimised by creating a function that calculates the value of $(valof\ X\ a + valof\ X\ b) \div \delta$ without multiplying and dividing by large powers of two. This was used to calculate a fraction and exponent, and the resulting values proved to satisfy the conditions imposed in Section 4.4.

Example

Addition using just this technique is much faster, as the following table shows:

a:	**b:**	**result:**	**Time(s):**	
			Standard	Optimised
(0,15,524288)+	(0,15,262144)=	(0,16,393216)	44.9	**9.3**
(0,31,1048575)+	(0,31,525288)=	(0,31,786432)	53.3	**11.7**
(0,62,2097151)+	(0,62,1048575)=	(0,63,1572863)	45.4	**9.7**
(0,125,4194303)+	(0,125,2097151)=	(0,126,3145727)	62.7	**22.0**
(0,251,8388607)+	(0,251,4194303)=	(0,252,6291455)	201.4	**87.1**

5.2 Rewrite optimisations

Throughout calculation there are many sums of the form:

$$\frac{2^a(2^b + c) \pm 2^d(2^b + e)}{2^f}$$

where $c < 2^b, e < 2^b$ and $d < a$. These were optimised by creating rewrite rules that change this calculation to one in which the division is replaced by factorising and subtracting powers. Then, as it is known that $e < 2^b$, division of the constants c and e by a power of two larger than b can be shown to be simply zero.

Logarithms were optimised by using case statements to select values based on equations such as the following:

$$\forall a\ b.\ 0 \leq b < 2^a \Rightarrow log_2(2^a + b) = a$$

$$\forall a\ b.\ 2^a \leq b < 2^{a+1} \Rightarrow log_2(2^a + b) = a + 1\ \ .$$

Cached rewrites A further optimisation was added at this stage, as floating point calculations for a given accuracy will all involve calculation of similar powers of two these could be cached. This was achieved by creating a function that calculated powers of two, then creating a rewrite rule for this function that looked up the result in a hash-table performing the calculation and storing the result if it was not found. This caching improved performance by a factor of at least four.

These optimisations result in a simulator that can perform calculations in under half a second:

a:	**b:**	**result:**	**Time(s):**
(0,15,524288)+	(0,15,262144)=	(0,16,393216)	**0.374**
(0,31,1048575)+	(0,31,525288)=	(0,31,786432)	**0.481**
(0,62,2097151)+	(0,62,1048575)=	(0,63,1572863)	**0.380**
(0,125,4194303)+	(0,125,2097151)=	(0,126,3145727)	**0.399**
(0,251,8388607)+	(0,251,4194303)=	(0,252,6291455)	**0.441**

The resulting mean time taken, on a 3.2GHz machine for 5000 additions and subtractions, is 0.44 seconds, the average number of inference steps involved being 25700.

5.3 Multiply

The multiply functions work in much the same way as the add functions, except the additional optimisation that the multiple of two denormal numbers will be zero is used.

5.4 Square Root

A square root of an integer may be calculated by performing a bitwise algorithm that checks each bit of the result to see if it should be part of the root.

The natural number root, r, of another natural number, v is defined as:

$$r^2 \leq v < (r+1)^2 \tag{3}$$

The algorithm was then proved to be correct by showing that it satisfied inequality, Equation 3.

Unfortunately direct calculation of the square root proved to be particularly slow, even for single precision numbers. Therefore the logarithm to the base two of the square root was calculated directly using the same method. For this function the two following properties may be proved:

$$2^{2x} \leq v \land \forall y.2^{2y} \leq v \Rightarrow y \leq x$$

Using these this function it should be possible to divide values by the eventual exponent before the square root is taken providing a large speed up.

6 Further Work

The square root algorithm needs to be completed and division optimised, the coprocessor also needs to be able to convert integers to floating point numbers and back. The IEEE floating-point library only defines a function which rounds a number to the nearest integer valued floating point number, this can be proved to be equivalent to rounding to the nearest integer, but it is not immediately obvious.

Finally the coprocessor model should be integrated with the ARM model [3] mentioned in Section 1.2 allowing complete programs containing floating-point instructions to be simulated.

References

1. J. R. Harrison, "Floating-point verification in HOL light: the exponential function", *Technical Report number 428, University of Cambridge Computer Laboratory. UK*, June 1997.
2. J. R. Harrison, "Proving with the real Numbers", *Technical Report number 408, University of Cambridge Computer Laboratory. UK*, December 1996.

3. A. C.J. Fox, "A HOL specification of the ARM instruction set architecture", *Technical Report number 545, University of Cambridge Computer Laboratory. UK*, June 2001
4. IEEE Standards, "IEEE Standard for Binary Floating-Point Arithmetic, Std. 754-1985", *IEEE Standard, IEEE*, 1985.
5. D. Seal, "ARM Architecture Reference Manual, Second Edition", *Addison-Wesley*, June 2000
6. M. J. C. Gordon and T. F. Melham, editors. "Introduction to HOL: a theorem proving environment for higher order logic", *Cambridge University Press, New York* 1993.

Teaching a HOL Course: Experience Report

Konrad Slind, Steven Barrus, Seungkeol Choe, Chris Condrat, Jianjun Duan, Sivaram Gopalakrishnan, Aaron Knoll, Hiro Kuwahara, Guodong Li, Scott Little, Lei Liu, Steffanie Moore, Robert Palmer, Claurissa Tuttle, Sean Walton, Yu Yang, and Junxing Zhang

School of Computing, University of Utah
slind@cs.utah.edu

Abstract. Experience from teaching a course in Higher Order Logic theorem proving is recounted, from both the student and teacher perspective.

1 Introduction

Higher Order Logic is great fun to work with, but we all had to learn it somehow. In the past, highly talented and persistent researchers and PhD students struggled to learn the logic (the easy part) and the capabilities of the implementations (the hard part). In many cases, successful practitioners absorbed the necessary collection of incantatations, minutiae, tricks, and other dark arts by sitting at the feet (or at a desk in the same office) of a Master, who had quite often been trained in a similar fashion. Such apprenticeships are often successful, since they allow very efficient information exchange. However, they are rather less efficient when the number of students increases.

The more structured approach of designing and delivering a course in a classroom setting has been taken many times before, of course. The lead author himself learned HOL from Graham Birtwistle, in a course largely based on painstaking reading of hardware verification transcripts in **hol88**. We would pore over proofs, following each tactic application and its result. This approach has much to recommend it, but is perhaps overly monastic for the students of today. Plus, the range of technologies available for disseminating information is much greater today.

A successful course by Tom Melham was used to teach the fundamentals of HOL and the **hol88** system for many years. The course provided a thorough, bottom-up route to facility with HOL: first, a small amount of ML was taught, then the basics of the logic (types, term formation, primitive rules of inference), then more advanced topics such as tactics, then higher-level packages, and finally extended case studies. The course could be adjusted to various time frames, up to a full week. Both the Isabelle and PVS communities have also conducted similar courses, and trained many accomplished researchers as a result.

2 Course Structure

The course was advertised as a graduate-level course with the only requirement being "desire and ability to reason formally". In the end sixteen students (some undergraduates) completed the course, and a number of others sat in. None of the participants had any knowledge of mechanized theorem proving at the start of the course. Only a few had been exposed to ML.

The first month and a half of the course were devoted to learning the basics of the HOL-4 system. This involved approximately two weeks learning ML, and two assignments:

- A collection of proofs in the theories of lists and regular expressions.
- Proofs about summation (Σ), applications of decision procedures, and proofs in set theory

In all exercises in the assignments, the goals were explicitly given so that students did not have to formulate correct goals.

At the beginning, classroom lectures focused on gaining familiarity with how to make definitions and express statements in the HOL logic. After that, proof tools were discussed, beginning with high-level proof tools (the simplifier, first order proof search, and decision procedures) and finishing with low-level tactics. Only a slight amount of discussion was devoted to forward inference vs. tactics: the ongoing assumption was that most proofs would be performed by tactic application.[1] In general, high-level proof tools were given much more emphasis than low-level tools. In retrospect, the emphasis on highly automated tools was one of the more arguable choices.

Subsequently, the students submitted a project proposal and embarked on a project, which took the duration of the course. The remainder of the lectures in the course were given over to a survey of automated theorem proving. This included the following topics:

- Equational Logic
  - Unification and matching
  - Proofs in equational logic
  - Rewriting (unconditional and conditional)
  - How to write your own simplifier in ML
- Propositional Logic
  - Proof, satisfaction, and refutation
  - Normal forms
  - Soundness and Completeness
  - Proof procedures (truth tables, resolution, and DPLL)
- First Order Logic
  - Syntax, proofs, and Skolemization
  - Models (Soundness, Deduction Theorem, Compactness, Completeness (Henkin's proof))

[1] In some cases, this hampered students in their project work.

- Decision Procedures
  - Decidability, undecidability
  - Fourier-Motzkin (Dense Linear Orders)
  - Quantifier Elimination, Cooper's algorithm

3 Projects

One of the prime reasons for the spread of higher order logic theorem provers is their expressive power: since these systems can basically formalize 'anything of interest', there is scope for a wide range of formalizations. This is reflected in the following projects which were proposed and undertaken by the students. Some students originally proposed projects (*e.g.*, Lie Algebra) requiring resources beyond those of the current version of HOL-4; in those cases, the instructor tried to point students towards more immediately fruitful proposals. However, in general little control of content was exercised, other than that the student had to know the subject matter well.

3.1 Expression Compiler

Initially, a compiler for a language with assignments and While loops was proposed. Specifying the implementation and correctness took some time, but real problems only occurred when the proofs were started (goals were simply too large). This prompted a retreat to simpler languages. Eventually, a compiler for arithmetic expressions was proved correct. The correctness proof required a significant generalization of the goal, which required instructor intervention.

3.2 Gödel Encoding

Gödelizing a formula means to translate it to a unique number. The project was to define the encoding and decoding functions and prove that decoding inverts encoding. (The student also re-formalized much of basic arithmetic, in order to gain more experience in theorem proving.)

3.3 Regular Languages

Automata and regular languages were formalized, and standard meta-theorems were proved: closure of regular languages under boolean operations and Kleene star. Also the Pumping Lemma for regular languages was proved. The proof of the Lemma requires the PigeonHole Principle, in a slightly non-standard form, which was not already available in HOL-4. Finally, the Pumping Lemma was applied to show the non-regularity of some specific languages. An interesting aspect of the work is that the student discovered that the obvious definition of regular sets in HOL will not work, since the definition has two type variables occurring in the right-hand-side of the definition, and only one on the left-hand-side.

3.4 Attempting to Break SHA-1 using SAT

The SHA-1 cryptographic hash function is heavily used in security applications, although there have been recent reports that it has been broken. The first phase of the project formalized SHA-1 as a functional program and then attempted to find pairs of plaintext that would hash to the same value. This can be mapped to a SAT problem by symbolically executing SHA-1. The attempt was ultimately unsuccessful, and it is not clear this path is feasible. The first problem was that the result of symbolic execution was not in CNF, and the conversion to CNF using the existing HOL-4 mechanism was very slow. Since SHA-1 does 80 rounds of hashing, we reduced the number of rounds. At one round, things were feasible, but at two, they were not. Formulas became so large that it wasn't clear what was going on.

3.5 IDEA Crypto Algorithm

The IDEA encryption algorithm is used in PGP (Pretty Good Privacy) and provides another application of symbolic execution to prove functional correctness of encryption algorithms [4]. The correctness of IDEA requires that its version of multiplication ($\mathbf{mod}\ 2^{16} + 1$) has an inverse. This required much work, the full formalization is discussed in [5].

3.6 Hardware Adders

Hardware verification was a popular topic: several projects on proving correctness of circuits were undertaken. One reason for this popularity may have been that a course on fast hardware implementation of arithmetic operations was being delivered by Erik Brunvand in the same semester[2]. Basic implementations such as ripple-carry and carry-lookahead adders were proved correct. More advanced prefix adders (Ladner-Fisher and Brent-Kung) were also verified. Mostly, conventional $imp \supset spec$ proofs were performed; however, one project took a somewhat different tack, using a Perl module for parsing Verilog to parse (fixed width) Verilog circuit descriptions into HOL terms, and then checking their equivalence with a ripple-carry adder using `HolSatLib`, HOL-4's interface to commonly-used SAT solvers.

3.7 Hardware Multipliers

The class of *Carry/Save Array* multipliers were formalized, starting from the transistor level, and applied to prove the unsigned Baugh-Wooley multiplier correct. The definitions were quite general, and provide a platform from which the correctness of other fast multipliers can be proved with less effort.

[2] Webpage: `http://www.cs.utah.edu/classes/cs5830`

3.8 Abstract Interpretation of Analog/Mixed Signal Circuits

Analog/mixed circuits are becoming more prevalent in implementations. However, formal tools for dealing with their properties are few and far between and haven't managed to scale well yet. One solution to this problem is to discretize the behaviour, then run finite-state tools. But then the problem is correctness of the abstraction. The project was just to show the essence of this correctness argument, where behaviour is abstracted to 2D zones in the plane.

3.9 FFT over Polynomial Trees

This project worked from a paper by Venanzio Capretta on verifying the Fast Fourier Transform in Type Theory[1]. There was significant overhead to overcome, especially absorbing Type Theory after one had just managed to learn the basics of higher order logic. Ultimately, the major challenge was formulating and applying induction principles for the underlying tree representation used in the original paper.

3.10 Matrices

The theory of matrices was tackled. Initial attempts at defining a matrix as a list of lists led to complex proofs for simple goals, involving nested induction. This lead to an exploration of a number of other formalizations, ending with the representation of a matrix as a triple, consisting of row bound, column bound, and indexing function. Based on this, the theory was developed up to the associativity of matrix multiplication. A large part of the effort was taken up by defining and developing a theory of summations.

3.11 Groups

The algebra of groups was tackled, and progressed to the First Isomorphism Theorem. The definition of groups was simple, but significant problems were encountered in applying the usual proof tools (first order proof search and simplification) in algebraic derivations. Simple identities were indeed simple, but solving group membership side-conditions needs special proof support (naively applying first order proof search works only in simple settings). Another challenge was dealing with the quotient construction in the Isomorphism theorem. Initially, Hilbert's Choice operator was used, but this lead to the usual difficulties. Happily, Skolemization was applicable instead.

3.12 Mutual Exclusion

The mutual exclusion problem was formulated set-theoretically, and the theorem that at most one process could be in the critical section at one time was proved. Each process was formalized as a non-deterministic finite-state machine, and a global scheduler (an application of the Axiom of Choice) was used to arbitrarily pick from multiple processes attempting to enter the critical section. An interesting next step in this effort would be to deal with fairness and liveness.

3.13 Thread Interleaving and Transactions

The Reduction Theorem from the research paper [2] was formalized and proven. The effort to follow the informal presentation in the paper provided the main difficulty, but in the end the informal definitions and proof were clarified and improved by the HOL formalization. Unlike many of the other projects, the number of definitions and the size of the statement of the theorem were quite large; there was therefore a large component of making sure that definitions actually captured the intended ideas.

3.14 Program Transformation

It is well-known that, subject to some constraints, linear recursions can be translated to tail recursions; however, it has been an ongoing challenge to automate this. The project was to implement a simple version of the translation, supported by formal proof. The code was applied to common recursive functions in the theory of numbers and lists, *e.g.*, factorial, reverse, flattening a list of lists, *etc.* The final version of the tool took a linear recursion, analyzed it, defined the corresponding tail recursion, and used automated proof to return an equality relating the two.

3.15 Java Program Verification

Krakatoa [3] is a system for verifying Java programs annotated with JML assertions. It is parameterized with backends for various theorem provers (the Coq port is most highly developed, but ports also exist for PVS, Harvey, CVC-Lite, among others). Integrating a HOL-4 backend and pushing through some standard examples required overcoming a wide range of obstacles. The HOL-4 port has recently been integrated into the Krakatoa release.

4 Comments from the Instructor

The students came up with an impressive range of examples, far more—and better—than I would have been able to come up with alone. Letting students choose projects in the intersection of their interest and competence is a good idea, although it can mean that the instructor is ‘at sea’ in some discussions. However, since the instructor is there to provide theorem proving expertise, the lack of domain knowledge wasn’t usually a problem.

The main problem for the instructor was the sheer amount of time and energy required to teach students enough so they could ‘stand on their own two feet’. My office was packed for the entire semester with students seeking help. The lack of a Teaching Assistant was a contributing factor, but, had there been a TA, he/she would have been also completely swamped. The learning curve is very steep! This was mitigated by other factors, chief among them being that it is fun to help people learn theorem proving.

We have to admit that interactive theorem provers are very powerful but also bewildering for beginners:

- Simply managing to formulate correct statements (well-formed types, well-formed terms, sensible goals) is a significant hurdle. It is very easy to get into situations where complex formulations (*e.g.*, uniqueness, maximality, relativized statements, dealing with partial functions) require some logic sophistication.
- Finding the correct tool to use at any point can be difficult. Moreover, there can be a 'superstitious' aspect to tool use, wherein, once a tool is found to solve one goal, the user tries to use it in all situations, when in fact other tools would be more appropriate. For example, first order proof search can be magically useful, but it can flounder on goals easily dealt with by a simplifier; and *vice versa*.
- Finding pre-proved theorem to apply (out of the 11,000 or so provided by HOL-4), or even remembering how to look for existing theorems can be hard. This of course is an ongoing problem with all interactive proof systems.

A common response to the question *How do I do X in HOL?* is *Well, I know at least three ways*. This flexibility is good for experts, and frustrating for beginners.

Part of getting a novice proof tool user to stand on their own two feet is teaching them a workable proof methodology. Being able to decompose a verification into a series of definitions and lemmas is fundamental to success. Fortunately, this was not a difficult notion for the students and seemed to be naturally taken up.

There is an astonishing range of abilities when it comes to 'picking up' a piece of software and learning to use it effectively. Interactive theorem provers like HOL-4 provide a wide range of theories, features, tools, documentation, and so forth. Arriving at a mental picture of what the tool *is* was a struggle for some. (Simply saying that HOL-4 is a system for generating proofs in Higher Order Logic isn't enough, alas.) Some students had little trouble navigating the system, while others were quickly blocked. How to eliminate this difference is an important problem.

A common phenomenon is that of *dialing-back*: the original proposal was far too ambitious, even with the requirement that the project proposal had to include a fall-back plan. As the course played out, expectations had to be continuously revised. This happens with all projects, of course.

A more characteristic difficulty with theorem proving is the unpleasant phenomenon of *Being Stuck*. One can easily get into this situation by stating false goals, not knowing the proof, not being able to formalize a known informal proof, *etc.* It is not catastrophic to Be Stuck, provided it is a temporary state of affairs. However, some students were extremely uncomfortable with the idea. This seems to be a significant difference between the practices of programming and proving:

usually there is a way forward in programming, while Getting Stuck is always a danger in formal proofs.

A majority of the students became quite enthusiastic. Quite a few projects have continued after the end of the course, sometimes because the student wanted to 'finish that last theorem', and sometimes because the project lies within their wider research goals.

Finally, given the project nature of the course, it was unclear whether the supplementary lectures were useful. However, attendance was surprisingly good all the way through the course. It is not clear whether this is because the content was so interesting or because of inertia.

5 Comments from the Students

We now present a lightly edited list of comments from the students. Some of the comments reflect the students' lack of experience, but that's just the point: these are typical comments from beginners, just after they've finished the main task of learning a new system.

- The system is vast and frustrating, I can't get it to do what I want.
- The command-line interface is too primitive.
- It is easy to forget how to get things done.
- Using pre-proved theorems with automated theorem provers on some of the assignments was easy and fun.
- Searching for a suitable theorem for use by a proof procedure is a nightmare.
- I was surprised by case-sensitivity.
- Mostly the high-level tactics are not useful and low-level tactics had to be used for most proofs.
- It would be nice if tactics printed out useful information when they fail.
- It is amazing that the HOL system contains such an enormous set of theories and proofs.
- There should already be a library supporting verification of arithmetic circuits.
- More homework and representative examples before starting the project.
- Dealing with assumptions is often irritating: they manage to confuse the automated provers when there are many assumptions.
- I had difficulty understanding error messages.
- The documentation needs more examples and explanations for students.
- There is a lack of examples when developing proofs.
- Performing a proof step can cause 'things to magically change' and I don't understand how the change happened.
- I was unable to get automated reasoners to understand set notation.
- I thought the most difficult part would be to do the reasoning steps but in fact getting the definitions right was the hard part.

- HOL itself was agreeable once things got rolling. The tool support I found most lacking was creating tactics. This is because it was time consuming and confusing to figure out when and where the case splits were in larger proofs.
- It is good that each recursively defined function automatically gets its own induction theorem.
- It would be helpful if HOL supported the deletion of assumptions that are no longer needed.
- Seemingly obvious paths are not obvious to HOL.
- In many instances, the advantages of using case analysis vs. induction was not clear.
- Defining a procedural process as a provable statement of logic required a lot of effort. But once defined, the statement appeared simple, even elegant.
- While the proof clearly was evident on paper, translating the steps into HOL was not intuitive, it seemed arcane.
- The tutorial and reference guides were not helpful, and the online help was less so.
- Develop a 'How-To' for various commonly performed proof steps.
- Write a FAQ for converting ML to HOL.
- HOL has a well-formed set theory, but more theorems could be added to HOL, especially cardinality theorems and equalities between set expressions.
- Writing and proving the universe was, perhaps, too large a jump.
- I feel that I outperformed myself in this project. I didn't plan to complete the whole thing, because the verification of ... alone requires the effort of a project. But the topic of this project really intrigued me, so I spent double time and efforts on it.

6 Conclusion

In the instructor's view the course results were quite encouraging: he had been expecting much resistance to the idea and practice of formal methods as performed in HOL-4, but most of the projects were interesting and successful. Although it was hard work (more than one student called it the hardest course they had ever taken), the majority of students did manage to learn enough to make a large and complex proof system do what they wanted. The collection of projects listed here could serve as a basis for others who are designing a similar project-based theorem proving course.

One observation is that most of the successful projects started from previous formalizations, or at least a correct set of definitions. Perhaps attempting to solidify definitions and get substantial proofs done is too much of a burden for beginners. On the other hand, we don't want to stifle application of the proof tools to already settled mathematics; several of the most successful projects of the course did involve much wrestling with definitions.

One crucial issue in teaching interactive proof is the following: Given that there is a finite amount of time (and student patience), what proof tools should be emphasized?. This course took the approach of focusing on the most highly automated tools. However, once the going gets tough in a proof, more specialized steps may need to be taken (simplifying only one specific occurrence of a term, explicitly instantiating a theorem, throwing away assumptions after use, *etc*) and the students did not have an adequate basis for going ahead with such steps. Possibly an assignment explicitly addressing such thorny aspects could be constructed.

Next time the course is taught, there will be a more significant attempt at integrating the lectures with the content; otherwise, the lectures don't seem to bear much relationship with the verifications in progress. Possibly another assignment could be added so that students can gain experience with writing their own proof tool, which is one of the major applications of HOL-4.

Focusing on a particular proof language, as is done with the Isar language in Isabelle could also ease the burden on the students, although understanding the behaviour of complex simplifiers and decision procedures would still be required.

In many universities, first order logic and its meta-theory, such as the soundness and completeness theorems, are routinely taught to undergraduates. A tool-supported undergraduate course in Higher Order Logic would seem to offer new benefits, *e.g.*, more compelling examples. The tool support, thanks largely to the long-term effort of the TPHOLs community, is substantially there; all that needs to be produced is an attractive curriculum for undergraduates. A project-based course described here still seems best for the graduate level though.

References

1. Venanzio Capretta. Certifying the Fast Fourier Transform with Coq. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 154–168. Springer-Verlag, 2001.
2. Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Types in Language Design and Implementation (TLDI 2003)*, pages 1–12. ACM, 2003.
3. Claude Marche', Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 2005. To appear.
4. Konrad Slind. A verification of Rijndael in HOL. In V. A Carreno, C. A. Munoz, and S. Tahar, editors, *Supplementary Proceedings of TPHOLs 2002*, number CP-2002-211736 in NASA Conference Proceedings, August 2002.
5. Junxing Zhang and Konrad Slind. Verification of Euclid's algorithm for finding multiplicative inverses. In Joe Hurd, editor, *Emerging Trends: Theorem Proving in Higher Order Logics: 18th International Conference, TPHOLs 2005*, 2005. (If accepted).

Using a SAT Solver as a Fast Decision Procedure for Propositional Logic in an LCF-style Theorem Prover[*]

Tjark Weber

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
webertj@in.tum.de

Abstract. This paper describes the integration of a leading SAT solver with Isabelle/HOL, a popular interactive theorem prover. The SAT solver generates resolution-style proofs for (instances of) propositional tautologies. These proofs are verified by the theorem prover. The presented approach significantly improves Isabelle's performance on propositional problems, and furthermore exhibits counterexamples for unprovable conjectures.

1 Introduction

Interactive theorem provers like PVS [17], HOL [8] or Isabelle [18] traditionally support rich specification logics. Proof search and automation for these logics however is difficult, and proving a non-trivial theorem usually requires manual guidance by an expert user. Automated theorem provers 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.

The two paradigms can be combined to obtain the best from both worlds. By integrating automatic decision procedures with interactive provers, we can preserve the richness of our specification logic and still increase the degree of automation [20]. Most interactive theorem provers nowadays come with a variety of built-in decision procedures that were specifically developed for the particular prover. Such decision procedures are often based on complex algorithms. To ensure that a potential bug in the decision procedure does not render the whole prover unsound, theorems in Isabelle, like in other LCF-style [6] provers, can be derived only through a set of core inference rules. Therefore it is not sufficient for a decision procedure to return whether a formula is provable, but the decision procedure must also generate the actual proof, expressed in terms of the prover's inference rules.

[*] This work was supported by the PhD program Logic in Computer Science of the German Research Foundation.

Built-in decision procedures enhance the capabilities of interactive provers, but seldom do they incorporate the latest advances in automated theorem proving. Automated theorem provers have become complex systems in their own right. Integrating their algorithms into an interactive system is a tedious task that requires continual maintenance. Therefore interfacing the two systems might be the more economic solution: later improvements to the ATP system are obtained "for free" in the interactive prover, only changes to its interface – which should happen much more rarely than changes to the algorithm – still require maintenance. This is an idea that goes back at least to the early nineties [12].

Formal verification is an important application area of interactive theorem proving. Problems in verification can often be reduced to Boolean satisfiability (SAT), and recent SAT solver advances have made this approach feasible in practice. Hence the performance of an interactive prover on propositional problems may be of significant practical importance. In this paper we describe the integration of zChaff [15], a leading SAT solver, with the Isabelle/HOL [16] prover. We show that using zChaff to prove theorems of propositional logic dramatically improves Isabelle's performance on this class of formulas. Furthermore, while Isabelle's previous decision procedures simply fail on unprovable conjectures, zChaff is able to produce concrete counterexamples. This can again be particularly useful in the context of formal verification, where a counterexample amounts to input data exhibiting faulty behavior of the hardware or system under examination.

The next section describes the integration of zChaff with Isabelle/HOL in more detail. In Section 3 we evaluate the performance of our approach, and report on experimental results. Related work is discussed in Section 4. Section 5 concludes this paper with some final remarks and points out directions for future research.

2 System Description

To prove a propositional tautology ϕ in the Isabelle/HOL system, but with the help of zChaff, we proceed in several steps. First ϕ is negated, and the negation is converted into an equivalent formula ϕ^* in conjunctive normal form. ϕ^* is then written to a file in DIMACS CNF format [4], the input format supported by zChaff (and many other SAT solvers). zChaff, when run on this file, returns either "unsatisfiable", or a satisfying assignment for ϕ^*.

In the latter case, the satisfying assignment – restricted to the Boolean variables occuring in ϕ – is displayed to the user. The assignment constitutes a counterexample to the original conjecture. When zChaff returns "unsatisfiable" however, things are more complicated. If we have confidence in the SAT solver, we can simply trust its result and accept ϕ as a theorem in Isabelle. The theorem is tagged with an "oracle" flag to indicate that it was proved not through Isabelle's own inference rules, but by an external tool. In this szenario, a bug in zChaff could allow us to derive inconsistent theorems in Isabelle/HOL.

The LCF-approach instead demands that we verify zChaff's claim of unsatisfiability within Isabelle/HOL. While this is not as simple as the validation of a satisfying assignment, the increasing complexity of SAT solvers has before raised the question of support for independent verification of their results, and in 2003 zChaff has been extended by L. Zhang and S. Malik [25] to generate resolution-style proofs that can be verified by an independent checker.[1] Hence our main task boils down to using Isabelle/HOL as an independent checker for the resolution proof found by zChaff.

zChaff stores this proof in a text file that is read in by Isabelle, and the individual resolution steps are replayed in Isabelle/HOL. Section 2.1 describes the necessary preprocessing of the input formula, and details of the proof reconstruction are explained in Section 2.2. The overall system architecture is shown in Figure 1.

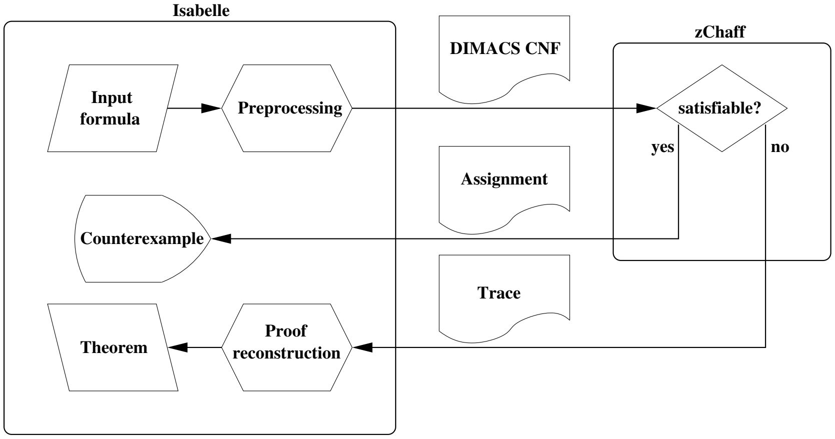


Fig. 1. System Architecture

2.1 Preprocessing

Isabelle/HOL offers higher-order logic (on top of Isabelle's meta logic), whereas zChaff only supports formulas of propositional logic in conjunctive normal form. Therefore the (negated) input formula ϕ must be preprocessed before it can be passed to zChaff.

[1] This is the very reason why we chose zChaff as the SAT solver to be integrated with Isabelle/HOL. Extending other DPLL-based solvers with proof-generating capabilities should be relatively simple [25], but despite some work in this direction [5], zChaff, to our knowledge, is currently the only proof-generating SAT solver that is publicly available.

First connectives of the meta logic, namely meta implication ($\Longrightarrow$) and meta equivalence ($\equiv$), are replaced by the corresponding HOL connectives $\longrightarrow$ and $=$. This is merely a technicality. Then the Boolean constants True and False are eliminated from ϕ, as are implication, $\longrightarrow$, and equivalence, $=$. The only remaining connectives are conjunction, disjunction, and negation. Finally ϕ is converted into negation normal form, and then into conjunctive normal form (CNF). The naive conversion currently implemented may cause an exponential blowup of the formula, but a Tseitin-style encoding [23] could easily be used instead. Quantified subformulas of ϕ are treated as atomic.

Note that it is not sufficient to convert ϕ into an equivalent formula ϕ' in CNF. Rather, we have to *prove* this equivalence inside Isabelle/HOL. The result is not a single formula, but a theorem of the form $\phi = \phi'$. Our main workhorse for the construction of this theorem is a generic function `thm_of`, proposed by A. Chaieb and T. Nipkow [3]:

```
thm_of decomp t =
  let
    (ts, recomb) = decomp t
  in recomb (map (thm_of decomp) ts)
```

It takes a decomposition function `decomp` of type $\alpha \to \alpha\, list \times (\beta\, list \to \beta)$ and a problem `t` of type α, decomposes `t` into a list of subproblems `ts` and a recombination function `recomb`, solves the subproblems recursively, and uses `recomb` to combine the recursive solutions into an overall solution. In our setting, `t` is a formula, `decomp` will look at its syntactic structure, and β is the type of theorems. We use reflexivity of $=$ when `t` is just a literal, and tautologies like $\neg P = P' \Longrightarrow \neg Q = Q' \Longrightarrow \neg(P \wedge Q) = P' \vee Q'$ (which are easily provable in Isabelle/HOL) to implement the recombination function. All of the conversions mentioned above can then be handled with proper instantiations for `decomp`.

zChaff treats clauses as sets of literals, making implicit use of associativity, commutativity and idempotence of disjunction. Therefore some further preprocessing is necessary, aside from conversion to CNF. Using associativity of conjunction and disjunction, we rewrite ϕ' into an equivalent CNF formula with unnecessary parentheses removed. In a second step, we remove duplicate literals, so that every clause contains each literal at most once. Finally, using $P \vee \neg P =$ True, we remove every clause that contains both a literal and its negation. Each preprocessing step yields an equivalence theorem that was proved in Isabelle/HOL, and transitivity of $=$ allows us to combine these theorems into a single theorem $\phi = \phi^*$, where ϕ^* is the final result of our conversion. Unless ϕ^* is already equal to True or False, it is then written to a file in DIMACS CNF format, and zChaff is invoked on this file.

2.2 Proof Reconstruction

When zChaff returns “unsatisfiable”, it also generates a resolution-style proof of unsatisfiability and stores the proof in a text file [25]. This file consists of three sections: clauses derived from the original problem by resolution, the values of variables implied by these clauses, and a conflict clause, i.e. a derived clause in which all literals are false. The text file is parsed by Isabelle, and the relevant information contained in it – i.e. the resolvents of newly generated clauses, the antecedents of variables, and the conflict clause's ID – is used to reconstruct the unsatisfiability proof in Isabelle/HOL. Proof reconstruction is based on two simple functions: one that uses resolution to derive new theorems of the form $\phi^* \longrightarrow c$ from existing theorems $\phi^* \longrightarrow c_1$, …, $\phi^* \longrightarrow c_n$ (where c and c_1, …, c_n are single clauses), and another function that proves $\phi^* \longrightarrow l$ (where l is a single literal) from l's antecedent $\phi^* \longrightarrow c$. Here c must be a clause that contains l, and for all other literals l' in c a theorem of the form $\phi^* \longrightarrow \neg l'$ must be provable. These functions correspond to the first and second section, respectively, of the text file generated by zChaff.

```
prove_clause clause_id =
  resolution (map prove_clause (resolvents_of clause_id))

prove_literal var_id =
  let
    th_ante = prove_clause (antecedent_of var_id)
    var_ids = filter (fn i => i <> var_id)
                     (var_ids_in_clause th_ante)
  in resolution (th_ante :: map prove_literal var_ids)
```

Resolution between two clauses c_1 and c_2 is always performed with the first literal in c_1 that occurs negated in c_2. Note that `resolution` must internally use associativity and commutativity of disjunction to reorder clauses, and idempotence to ensure that the resulting clause contains each literal at most once.

Proof reconstruction proceeds in three steps. First the conflict clause is proved by a call to `prove_clause`. Then `prove_literal` is called for every literal in the conflict clause, to show that the literal must be false. Finally resolving the conflict clause with these negated literals yields the theorem $\phi^* \longrightarrow \text{False}$.

For efficiency reasons, the actual implementation is slightly different from what is shown above. Some clauses that were derived by zChaff may be used many times during the proof, while others are perhaps not used at all. Theorems that were proved once are therefore stored in two arrays (one for clauses, one for literals), and simply looked up – rather than reproved – should they be needed again. Hence our implementation is not purely functional.

2.3 A Simple Example

In this section we illustrate the proof reconstruction using a small example. Consider the following input formula

$$\phi \equiv (\neg v1 \lor v2) \land (\neg v2 \lor \neg v3) \land (v1 \lor v2) \land (\neg v2 \lor v3).$$

Since ϕ is already in conjunctive normal form, preprocessing simply yields the theorem $\phi = \phi$. The corresponding DIMACS CNF file, aside from its header, contains one line for each clause in ϕ:

```
-1  2 0
-2 -3 0
 1  2 0
-2  3 0
```

zChaff easily detects that this problem is unsatisfiable, and creates a text file with the following data:

```
CL: 4 <= 2 0
VAR: 2 L: 0 V: 1 A: 4 Lits: 4
VAR: 3 L: 1 V: 0 A: 1 Lits: 5 7
CONF: 3 == 5 6
```

This tells Isabelle that first a new clause (with ID 4) is derived by resolving clause 2, $v1 \lor v2$, with clause 0, $\neg v1 \lor v2$. The first variable that occurs both positively and negatively in clause 2 and clause 0 is $v1$; this variable is eliminated by resolution.

Now the value of variable 2 (`VAR: 2`) can be deduced from clause 4 (`A: 4`). $v2$ must be true (`V: 1`). Clause 4 contains only one literal (`Lits: 4`), namely $v2$ (since $4 \div 2 = 2$), occuring positively (since $4 \bmod 2 = 0$). This decision is made at level 0 (`L: 0`), before any decision at higher levels.

Likewise, the value of variable 3 can then be deduced from clause 1, $\neg v2 \lor \neg v3$. $v3$ must be false (`V: 0`).

Finally clause 3 is our conflict clause. It contains two literals, $\neg v2$ (since $5 \div 2 = 2$, $5 \bmod 2 = 1$) and $v3$ (since $6 \div 2 = 3$, $6 \bmod 2 = 0$). But we already know that both literals must be false, so this clause is not satisfiable.

Note that information concerning the level of decisions, the actual value of variables, or the literals that occur in a clause is redundant in the sense that it is not needed by Isabelle to validate zChaff's proof. This information can always be reconstructed from the original problem.

3 Evaluation

Isabelle/HOL offers three major automatic proof procedures: *auto*, which performs simplification and splitting of a goal, *blast* [19], a tableau-based prover, and *fast*, which searches for a proof using standard Isabelle inference. Details can be found in [16]. We compared the performance of our approach to that of Isabelle's existing proof procedures on all 42 problems contained in version 2.6.0 of the TPTP library [22] that have a representation in propositional logic. The problems were negated, so that unsatisfiable problems became provable. All benchmarks were run on a machine with a 3 GHz Intel Xeon CPU and 1 GB of main memory.

Problem	Status	auto	blast	fast	zChaff
MSC007-1.008	unsat.	**x**	**x**	**x**	726.5
NUM285-1	sat.	**x**	**x**	**x**	0.2
PUZ013-1	unsat.	0.5	**x**	5.0	0.1
PUZ014-1	unsat.	1.4	**x**	6.1	0.1
PUZ015-2.006	unsat.	**x**	**x**	**x**	10.5
PUZ016-2.004	sat.	**x**	**x**	**x**	0.3
PUZ016-2.005	unsat.	**x**	**x**	**x**	1.6
PUZ030-2	unsat.	**x**	**x**	**x**	0.7
PUZ033-1	unsat.	0.2	6.4	0.1	0.1
SYN001-1.005	unsat.	**x**	**x**	**x**	0.4
SYN003-1.006	unsat.	0.9	**x**	1.6	0.1
SYN004-1.007	unsat.	0.3	822.2	2.8	0.1
SYN010-1.005.005	unsat.	**x**	**x**	**x**	0.4
SYN086-1.003	sat.	**x**	**x**	**x**	0.1
SYN087-1.003	sat.	**x**	**x**	**x**	0.1
SYN090-1.008	unsat.	13.8	**x**	**x**	0.5
SYN091-1.003	sat.	**x**	**x**	**x**	0.1
SYN092-1.003	sat.	**x**	**x**	**x**	0.1
SYN093-1.002	unsat.	1290.8	16.2	1126.6	0.1
SYN094-1.005	unsat.	**x**	**x**	**x**	0.8
SYN097-1.002	unsat.	**x**	19.2	**x**	0.2
SYN098-1.002	unsat.	**x**	**x**	**x**	0.4
SYN302-1.003	sat.	**x**	**x**	**x**	0.4

Table 1. Running times (in seconds) for TPTP problems

19 of these 42 problems are rather easy, and were solved in less than a second each by both the existing procedures and the SAT solver approach. Table 1 shows the times in seconds required to solve the remaining 23 problems. An **x** indicates that the procedure ran out of memory or failed to terminate within an hour.

Proof reconstruction in Isabelle/HOL is currently several orders of magnitude slower than proof verification with an external checker [25] written in C++. While there may still be potential for optimization in the Isabelle/HOL implementation, profiling indicates that this difference must mainly be attributed to the data structures and functions provided by Isabelle's LCF-style kernel, which are not geared towards clausal reasoning.

The SAT solver approach dramatically outperforms the automatic proof procedures that were previously available in Isabelle/HOL. The other procedures combined solved only 8 of the harder problems. Running times between the different procedures vary wildly, and they all fail to terminate for the 7 satisfiable (i.e. unprovable) problems. In contrast, the SAT solver approach solves *all* problems, takes less than two seconds on all but two problems, and provides actual counterexamples for the unprovable problems. Furthermore, the rightmost column of Table 1 already shows the total (combined) time for the invocation of zChaff and the following proof reconstruction in Isabelle/HOL. zChaff alone terminates after a fraction of this time, at which point a definite answer can already be displayed to the user – a feature that is particularly useful in our interactive setting.

4 Related Work

Michael Gordon has implemented *HolSatLib* [7], a library which is now part of the HOL 4 theorem prover. This library provides functions to convert HOL 4 terms into CNF, and to analyze them using a SAT solver. In the case of unsatisfiability however, the user only has the option to trust the external solver. No proof reconstruction takes place, "since there is no efficient way to check for unsatisfiability using pure Hol98 theorem proving" [7]. A bug in the SAT solver could ultimately lead to an inconsistency in HOL 4.

Perhaps closer related to our work is the integration of automated first-order provers, recently further explored by Joe Hurd [10, 11] and Jia Meng [13, 14]. Proofs found by the automated system are either verified by the interactive prover immediately [10], or translated into a proof script that can be executed later [14]. The main focus of their work however is on the necessary translation from the interactive prover's specification language to first-order logic. In contrast our approach is so far restricted to instances of propositional tautologies, but it avoids difficult translation issues, and uses a SAT solver, rather than a first-order prover.

Other applications of SAT solvers in the context of theorem proving include SAT-based decision procedures for richer logics (e.g. [2, 21]), as well as SAT-based model generation techniques (e.g. [1, 24]). These applications again require involved translations, and a correctly implemented SAT solver is usually taken for granted.

5 Conclusions and Future Work

Our results show that the zChaff-based tactic is clearly superior to Isabelle's built-in tactics for propositional formulas. With the help of zChaff, many formulas that were previously out of the scope of Isabelle's built-in tactics can now be proved – or refuted – automatically, often within seconds. Isabelle's applicability as a tool for formal verification, where large propositional problems occur in practice, has thereby improved considerably.

However, it is also important to note that Isabelle's performance is still not sufficient for problems with thousands of clauses, like some of those found in the SATLIB library [9]. While zChaff and other recent SAT solvers may well be able to decide these problems in practice, their sheer size currently does not permit an efficient treatment in Isabelle/HOL. Further work is necessary to investigate if this issue can be resolved by relatively minor optimizations to Isabelle's kernel, or if a kernel extension with optimized data structures and algorithms for propositional logic is more promising.

The approach presented in this paper has applications beyond propositional reasoning. The decision problem for (fragments of) richer logics can be reduced to SAT [2, 21]. Consequently, proof reconstruction for propositional logic can serve as a foundation for proof reconstruction for other logics. Based on our work, one only needs a proof-generating implementation of the reduction to integrate the whole SAT-based decision procedure with an LCF-style theorem prover.

Acknowledgments The author would like to thank Sharad Malik and Zhaohui Fu for their help with zChaff, and Tobias Nipkow for his valuable suggestions.

References

[1] Pranav Ashar, Malay Ganai, Aarti Gupta, Franjo Ivancic, and Zijiang Yang. Efficient SAT-based bounded model checking for software verification. In *1st International Symposium on Leveraging Applications of Formal Methods, ISOLA 2004*, 2004.

[2] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 195–210, Copenhagen, Denmark, July 2002. Springer.

[3] Amine Chaieb and Tobias Nipkow. Generic proof synthesis for Presburger arithmetic. Technical report, Technische Universität München, October 2003.

[4] DIMACS satisfiability suggested format, 1993. Available online at ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc.

[5] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE 2003)*, pages 10886–10891. IEEE Computer Society, 2003.

[6] M. J. C. Gordon. From LCF to HOL: a short history. In G. Plotkin, Colin P. Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*. MIT Press, 2000.

[7] M. J. C. Gordon. HolSatLib documentation, version 1.0b, June 2001. Available online at http://www.cl.cam.ac.uk/˜mjcg/HolSatLib/HolSatLib.html.

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

[9] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT 2000*, pages 283–292. IOS Press, 2000. Available online at http://www.satlib.org/.

[10] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321, Nice, France, September 1999. Springer.

[11] Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Copenhagen, Denmark, July 2002. Springer.

[12] R. Kumar, T. Kropf, and K. Schneider. Integrating a first-order automatic prover in the HOL environment. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J.

Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 170–176, Davis, California, USA, August 1991. IEEE Computer Society Press, 1992.

[13] Jia Meng. Integration of interactive and automatic provers. In Manuel Carro and Jesus Correas, editors, *Second CologNet Workshop on Implementation Technology for Computational Logic Systems, FME 2003*, September 2003.

[14] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning: Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 372–384. Springer, 2004.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, Las Vegas, June 2001.

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[17] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer.

[18] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[19] Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–83, 1999.

[20] Natarajan Shankar. Using decision procedures with a higher-order logic. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 5–26. Springer, 2001.

[21] Ofer Strichman. On solving Presburger and linear arithmetic with SAT. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design: 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*, volume 2517 of *Lecture Notes in Computer Science*, pages 160–169. Springer, 2002.

[22] Geoff Sutcliffe and Christian Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. Available online at http://www.cs.miami.edu/~tptp/.

[23] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation Of Reasoning: Classical Papers On Computational Logic, Vol. II, 1967-1970*, pages 466–483. Springer, 1983. Also in *Structures in Constructive Mathematics and Mathematical Logic Part II*, ed. A. O. Slisenko, 1968, pp. 115–125.

[24] Tjark Weber. Bounded model generation for Isabelle/HOL. In *2nd International Joint Conference on Automated Reasoning (IJCAR 2004) Workshop W1: Workshop on Disproving – Non-Theorems, Non-Validity, Non-Provability*, Cork, Ireland, July 2004.

[25] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE 2003)*, pages 10880–10885. IEEE Computer Society, 2003.

Liveness Proof of An Elevator Control System [⋆]

Huabing Yang, Xingyuan Zhang, and Yuanyuan Wang

PLA University of Science and Technology, P.R. China
yanghuabing@gmail.com, xyzhang@public1.ptt.js.cn

Abstract. In [11], a method used to prove liveness properties in the setting of inductive protocol verification is proposed. As a companion paper, this paper shows the practical aspects of the proposed method. Elevator control system, a benchmark problem in formal specification and verification, has been chosen as a case study. A liveness property could be difficult to prove when it concerns non-trivial loops in the underlying state transition diagram. The liveness property proved in this paper is such a difficult property, which has never been proved before for an elevator control system with arbitrarily large state space. We show that using the method proposed in [11], such difficult liveness properties can be proved with reasonable human effort.

Keywords: Inductive Protocol Verification, Temporal Reasoning, Isabelle, Elevator Control System

1 Introduction

As a companion paper of [11], this paper describes how the method proposed there is used to verify the liveness of an elevator control system. Elevator control system is a benchmark problem in the area of formal specification and verification. The liveness of elevator control system is non-trivial. In a number of efforts to formalize elevator control systems [7,10,9,3,4,2,12,6,1,5,8], only [12], [7], [1], [8] and [5] treat liveness. Since we can not find paper [1] and [5], a comparison of this paper with only [12], [7] and [8] is given in Table 1.

Work	**For arbitrarily large state space?**	**Conclusion**
[12]	No	$\Box(\langle(\!\vert \mathit{Down}\ n \vert\!)\rangle \hookrightarrow \Diamond\ \langle(\!\vert \mathit{Stop}\ n \vert\!)\rangle);$ $\Box(\langle(\!\vert \mathit{Up}\ n \vert\!)\rangle \hookrightarrow \Diamond\langle(\!\vert \mathit{Stop}\ n \vert\!)\rangle);$ $\Box(\langle(\!\vert \mathit{To}\ n \vert\!)\rangle \hookrightarrow \Diamond\langle(\!\vert \mathit{Stop}\ n \vert\!)\rangle)$
[7]	No	$\Box(\langle(\!\vert \mathit{Down}\ n \vert\!)\rangle \hookrightarrow \Diamond\ \langle(\!\vert \mathit{Stop}\ n \vert\!)\rangle);$ $\Box(\langle(\!\vert \mathit{Up}\ n \vert\!)\rangle \hookrightarrow \Diamond\langle(\!\vert \mathit{Stop}\ n \vert\!)\rangle)$
[8]	Yes	$\Box(\mathit{Down}\ n \hookrightarrow \Diamond\ \mathit{Stop}\ n);$ $\Box(\mathit{Up}\ n \hookrightarrow \Diamond\ \mathit{Stop}\ n)$
This paper	Yes	$\Box(\langle(\!\vert \mathit{Arrive}\ p\ m\ n \vert\!)\rangle \hookrightarrow \Diamond\ \langle(\!\vert \mathit{Exit}\ p\ m\ n \vert\!)\rangle)$

Table 1. A comparison with previous works

It can be seen from Table 1 that the liveness conclusion in this paper is more realistic than those in [12,7,8], because *Arrive p m n* represents the event that

[⋆] This research was funded by National Natural Science Foundation of China, under grant 60373068 ‘Machine-assisted correctness proof of complex programs’

person p arrives at floor m wanting to go to floor n and *Exit* p m n represents the event that person p gets off at floor n. The meaning of events in an elevator control system is explained in section 2.1. The works in [12,7,8] only prove that if a button is pressed at floor n(represented by event *Up* n or *Down* n), the elevator will eventually stop at floor n, or if the button n inside the elevator is pressed(represented by event *To* n), the elevator will eventually reach and stop at floor n. These properties do not necessarily guarantee that a person who want to go to a certain floor will eventually reach there, considering the cases when a person temporarily forgets to get onboard when the elevator arrives or forgets to get off when the elevator reaches his destination. The consideration of human behavior gives rise to non-trivial loops in the underlying state transition diagram. Such non-trivial loops make liveness proof difficult.

It is shown in this paper that such difficult liveness properties can be established with reasonable human labor, using the method proposed in [11], under stronger non-standard fairness constraints. Compared with [8], the effort spent in this paper is greater, but the conclusion is stronger as well. Compared with [12,7], this paper can deal with systems with arbitrarily large state space. This is because the theorem proving technique used in this paper overcomes the state space explosion problem, which is intrinsic to the model checking technique used in [12,7].

As an illustration of the method in [11], a detailed explanation is given on how an informal state-transition diagram can help in the construction of a liveness proof.

The rest of this paper is organized as the following: section 2 gives the formal definition of the elevator control system in full detail; section 3 describes the liveness proof; section 4 concludes.

2 Formalizing the elevator control system

2.1 Events

The elevator control system to be formalized is shown in Figure 1. We consider systems with only one elevator. The elevator can be in stationary state, as shown in Figure 1, or on the move, as shown in Figure 2. There are two buttons on each floor, ▲ and ▼, used by users to signal to the elevator control system their requests to go up or down. There is a control panel inside the elevator. After getting into the elevator, users can signal their destinations by pressing number buttons on the control panel. We do not consider the open and close of elevator, neither the on and off of various kind of lights. People can get on and off the elevator as long as the elevator is in stationary state. There is no limit on the number of people the elevator can hold.

Without loss of generality, both floor and user are represented as natural number:

types *floor* = *nat* — type for floors

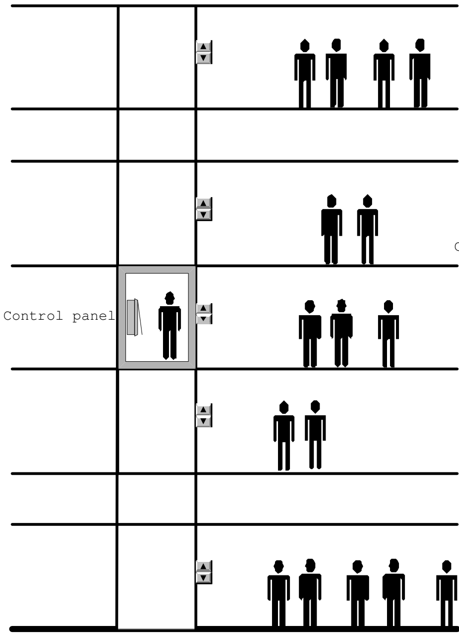


Fig. 1. Elevator in stationary

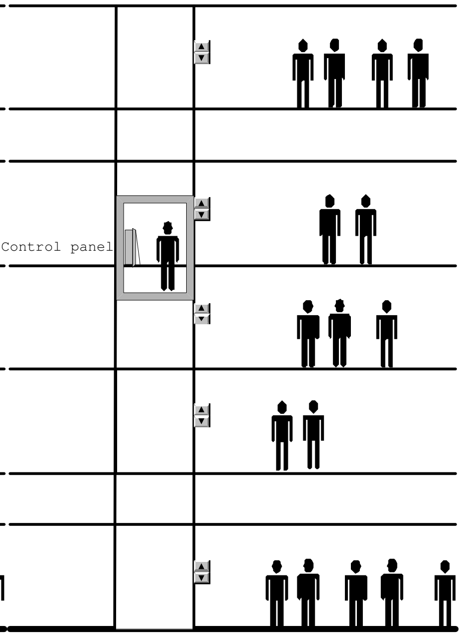


Fig. 2. Elevator on the move

types *user* = *nat* — type for users

The type of events that may happen in an elevator control system is defined as:

datatype *event* =

Arrive user floor floor — *Arrive p m n*: User *p* arrives at floor *m*, planning to go to floor *n*.

| *Enter user floor floor* — *Enter p m n*: User *p* enters elevator at floor *m*, planning to go to floor *n*.

| *Exit user floor floor* — *Exit p m n*: User *p* gets off elevator at floor *n*, *m* is the floor, where user *p* entered elevator.

| *Up floor* — *Up n*: A press of the ▲-button on floor *n*.

| *Down floor* — *Down n*: A press of the ▼-button on floor *n*.

| *To floor* — *To n*: A press of button *n* on elevator's control panel.

| *StartUp floor* — *StartUp n*: Elevator starts moving upward from floor *n*.

| *StartDown floor* — *StartDown n*: Elevator starts moving down from floor *n*.

| *Stop floor floor* — *Stop n m*: Elevator stops at floor *m*. Before stopping, the elevator is moving from floor *n* to floor *m*.

| *Pass floor floor* — *Pass n m*: Elevator passes floor n without stopping. Before passing floor m, the elevator is moving from floor n to floor m.

2.2 Observation functions

Observation functions compute values from current state τ. Concurrent system uses these observation values when deciding which events are eligible to happen next.

The observation function *now-flr* computes the state of elevator. The definition of *now-flr* is:

consts $\textit{now-flr} :: \textit{event list} \Rightarrow (\textit{floor} \times \textit{floor})$
  $\textit{now-flr}\ [] = (0,\ 0)$
  $\textit{now-flr}\ (\textit{Stop}\ n\ m\ \#\ \tau) = (m,\ m)$
  $\textit{now-flr}\ (\textit{StartUp}\ n\ \#\ \tau) = (n,\ n + 1)$
  $\textit{now-flr}\ (\textit{StartDown}\ n\ \#\ \tau) = (n,\ n - 1)$
  $\textit{now-flr}\ (\textit{Pass}\ n\ m\ \#\ \tau) = (\textit{if}\ n < m\ \textit{then}\ (m,\ m + 1)\ \textit{else}\ (m,\ m - 1))$
  $\textit{now-flr}\ (e\ \#\ \tau) = \textit{now-flr}\ \tau$

$\textit{now-flr}\ \tau = (n,\ n)$ means the elevator is static at floor n. $\textit{now-flr}\ \tau = (n,\ m)$, $(n{\neq}m)$ means the elevator is moving from floor n to floor m.

Observation function *up-set* computes the set of floors, on which there is a ”go-up” request not served. The definition of *up-set* is:

consts $\textit{up-set} :: \textit{event list} \Rightarrow \textit{floor set}$
  $\textit{up-set}\ [] = \{\}$
  $\textit{up-set}\ (\textit{Up}\ n\ \#\ \tau) = \{n\} \cup (\textit{up-set}\ \tau)$
  $\textit{up-set}\ (\textit{StartUp}\ n\ \#\ \tau) = \textit{up-set}\ \tau - \{n\}$
  $\textit{up-set}\ (e\ \#\ \tau) = \textit{up-set}\ \tau$

A request to go up is signaled when the ▲-button is pressed. The request is *served* when the elevator starts moving up from the requesting floor.

Observation function *down-set* computes the set of floors, on which there is a ”go-down” request not served. The definition of *down-set* is:

consts $\textit{down-set} :: \textit{event list} \Rightarrow \textit{floor set}$
  $\textit{down-set}\ [] = \{\}$
  $\textit{down-set}\ (\textit{Down}\ n\ \#\ \tau) = \{n\} \cup (\textit{down-set}\ \tau)$
  $\textit{down-set}\ (\textit{StartDown}\ n\ \#\ \tau) = \textit{down-set}\ \tau - \{n\}$
  $\textit{down-set}\ (e\ \#\ \tau) = \textit{down-set}\ \tau$

A request to go down is signaled when the ▼-button is pressed. The request is *served* when the elevator starts moving down from the requesting floor.

Observation function *dest* computes the set of floors, which users want to arrive, but have not been arrived. The definition of *dest* is:

consts $\textit{dest} :: \textit{event list} \Rightarrow \textit{floor set}$
  $\textit{dest}\ [] = \{\}$

$\mathit{dest}\ (\mathit{To}\ n\ \#\ \tau) = \{n\} \cup \mathit{dest}\ \tau$
$\mathit{dest}\ (\mathit{Stop}\ n'\ n\ \#\ \tau) = \mathit{dest}\ \tau - \{n\}$
$\mathit{dest}\ (e\ \#\ \tau) = \mathit{dest}\ \tau$

After entering elevator, a user signals a request to go to floor n by pressing button n on the control panel. This request is considered served when the elevator stops at floor n.

Observation function *arr-set* computes the set of people, who are still waiting to enter the elevator. The definition of *arr-set* is:

consts $\mathit{arr\text{-}set} :: \mathit{event\ list} \Rightarrow \mathit{event\ set}$
$\mathit{arr\text{-}set}\ [] = \{\}$
$\mathit{arr\text{-}set}\ (\mathit{Arrive}\ p\ m\ n\ \#\ \tau) = \{\mathit{Arrive}\ p\ m\ n\} \cup \mathit{arr\text{-}set}\ \tau$
$\mathit{arr\text{-}set}\ (\mathit{Enter}\ p\ m\ n\ \#\ \tau) = \mathit{arr\text{-}set}\ \tau - \{\mathit{Arrive}\ p\ m\ n\}$
$\mathit{arr\text{-}set}\ (e\ \#\ \tau) = \mathit{arr\text{-}set}\ \tau$

$\mathit{Arrive}\ p\ m\ n \in \mathit{arr\text{-}set}\ \tau$ means user p is waiting at floor m and he is planning to go to floor n.

Observation function $\mathit{ent\text{-}set}\ \tau$ computes the set of people, who have entered the elevator, but yet to exit. The definition of $\mathit{ent\text{-}set}\ \tau$ is:

consts $\mathit{ent\text{-}set} :: \mathit{event\ list} \Rightarrow \mathit{event\ set}$
$\mathit{ent\text{-}set}\ [] = \{\}$
$\mathit{ent\text{-}set}\ (\mathit{Enter}\ p\ m\ n\ \#\ \tau) = \mathit{ent\text{-}set}\ \tau \cup \{\mathit{Enter}\ p\ m\ n\}$
$\mathit{ent\text{-}set}\ (\mathit{Exit}\ p\ m\ n\ \#\ \tau) = \mathit{ent\text{-}set}\ \tau - \{\mathit{Enter}\ p\ m\ n\}$
$\mathit{ent\text{-}set}\ (e\ \#\ \tau) = \mathit{ent\text{-}set}\ \tau$

$\mathit{Enter}\ p\ m\ n \in \mathit{ent\text{-}set}\ \tau$ means user p has entered the elevator, but has not exited yet, moreover, user p entered the elevator at floor m and he is planning to go to floor n.

The motion of elevator alternates between *Up* and *Down* pass. Observation function *is-up* computes the current pass of the elevator.

consts $\mathit{is\text{-}up} :: \mathit{event\ list} \Rightarrow \mathit{bool}$
$\mathit{is\text{-}up}\ [] = \mathit{True}$
$\mathit{is\text{-}up}\ (\mathit{StartUp}\ m\ \#\ \tau) = \mathit{True}$
$\mathit{is\text{-}up}\ (\mathit{StartDown}\ m\ \#\ \tau) = \mathit{False}$
$\mathit{is\text{-}up}\ (e\ \#\ \tau) = \mathit{is\text{-}up}\ \tau$

$\mathit{is\text{-}up}\ \tau$ means the elevator is in *Up*-pass; otherwise, the elevator is in *Down*-pass. If the elevator is currently in *Up*-pass, a happening of *StartDown* changes it into *Down*-pass, and it remains in *Down*-pass until the next happening of *StartUp*.

2.3 User and elevator rules

The height of the building, in which the elevator is mounted, is assumed to be no more than H floors, where H is an arbitrary natural number greater than 0. There are $H+1$ floors served by the elevator. The set of these floors is $\{0 \ldots H\}$.

The elevator control system is formalized as the following concurrent system *elev-cs*, which contains 16 rules, each describing either the behaviour of users or the elevator. To understand the definition of *elev-cs*, it should be noticed that the elevator needs to go to floor n only when there are some requests at floor n, or there is a user inside the elevator, who wants to go to floor n. Therefore, the set of floors to which the elevator needs to go, is naturally written as: $\mathit{up\text{-}set}\ \tau \cup \mathit{down\text{-}set}\ \tau \cup \mathit{dest}\ \tau$.

consts $\mathit{elev\text{-}cs} :: (\mathit{event\ list} \times \mathit{event})\ \mathit{set}$
inductive *elev-cs* **intros**

e0: $[\![m \neq n;\ m \leq H;\ n \leq H]\!] \Longrightarrow (\tau,\ \mathit{Arrive}\ p\ m\ n) \in \mathit{elev\text{-}cs}$

> — Rule *e0* describes the behaviour of users. If user p arrives at level m and wants to go to level n, both m and n should be less than H, and m should be different from n. This way, the height of the building is implicitly constrained to be no more than H floors.

e1: $[\![\mathit{Arrive}\ p\ m\ n \in \mathit{arr\text{-}set}\ \tau;\ n < m]\!] \Longrightarrow (\tau,\ \mathit{Down}\ m) \in \mathit{elev\text{-}cs}$

> — Rule *e1* describes the behaviour of users. When a user p is waiting at level m and he wants to go to level n, if $n < m$, he will press the ▼-button at floor m.

e2: $[\![\mathit{Arrive}\ p\ m\ n \in \mathit{arr\text{-}set}\ \tau;\ m < n]\!] \Longrightarrow (\tau,\ \mathit{Up}\ m) \in \mathit{elev\text{-}cs}$

> — Rule *e2* describes the behaviour of users. When a user p is waiting at level m and he wants to go to level n, if $m < n$, he will press the ▲-button at floor m.

e3: $[\![\mathit{Enter}\ p\ m\ n \in \mathit{ent\text{-}set}\ \tau]\!] \Longrightarrow (\tau,\ \mathit{To}\ n) \in \mathit{elev\text{-}cs}$

> — Rule *e3* describes the behaviour of users. After a user p enters elevator and he wants to go to floor n, he will press button n on the control panel.

e4: $[\![\mathit{now\text{-}flr}\ \tau = (m,\ m);\ \mathit{Arrive}\ p\ m\ n \in \mathit{arr\text{-}set}\ \tau]\!]$
$\Longrightarrow (\tau,\ \mathit{Enter}\ p\ m\ n) \in \mathit{elev\text{-}cs}$

> — Rule *e4* describes the behaviour of users. If a user p is waiting at floor m, and he wants to go to floor n, if the elevator is now stopped at floor m, then user p can get onboard the elevator.

e5: $[\![\mathit{now\text{-}flr}\ \tau = (n,\ n);\ \mathit{Enter}\ p\ m\ n \in \mathit{ent\text{-}set}\ \tau]\!]$
$\Longrightarrow (\tau,\ \mathit{Exit}\ p\ m\ n) \in \mathit{elev\text{-}cs}$

> — Rule *e5* describes the behaviour of users. If a user p is inside the elevator, and he wants to go to floor n, if the elevator is now stopped at floor n, then user p can get off the elevator.

e6: $[\![\mathit{now\text{-}flr}\ \tau = (n,\ n);\ \mathit{is\text{-}up}\ \tau;$
$\exists\ x \in (\mathit{up\text{-}set}\ \tau \cup \mathit{down\text{-}set}\ \tau \cup \mathit{dest}\ \tau).\ n < x]\!]$
$\Longrightarrow (\tau,\ \mathit{StartUp}\ n) \in \mathit{elev\text{-}cs}$

> — Rule *e6* describes the behavior of the elevator. The set $\mathit{up\text{-}set}\ \tau \cup \mathit{down\text{-}set}\ \tau \cup \mathit{dest}\ \tau$ represents all requests in the system. The elevator is currently is *Up* pass, and stopped at floor n. If there exist requests which are above floor n (specified as $\exists\ x \in (\mathit{up\text{-}set}\ \tau \cup \mathit{down\text{-}set}\ \tau \cup \mathit{dest}\ \tau).\ n < x$), then, the elevator should start moving up.

e7: $[\![\mathit{now\text{-}flr}\ \tau = (n,\ n);\ \neg\ \mathit{is\text{-}up}\ \tau;$
$\exists\ x \in (\mathit{up\text{-}set}\ \tau \cup \mathit{down\text{-}set}\ \tau \cup \mathit{dest}\ \tau).\ n < x;$
$\forall\ x \in (\mathit{up\text{-}set}\ \tau \cup \mathit{down\text{-}set}\ \tau \cup \mathit{dest}\ \tau).\ n \leq x]\!]$
$\Longrightarrow (\tau,\ \mathit{StartUp}\ n) \in \mathit{elev\text{-}cs}$

— Rule *e7* describes the behaviour of the elevator. The elevator is currently in *Down* pass and stopped at floor *n*. If all requests in the system are no lower than floor *n* (specified by the $\forall\ x \in (\textit{up-set}\ \tau \cup \textit{down-set}\ \tau \cup \textit{dest}\ \tau).\ n \leq x$), and there are some requests which are strictly above floor *n* (specified by $\exists\ x \in (\textit{up-set}\ \tau \cup \textit{down-set}\ \tau \cup \textit{dest}\ \tau).\ n < x$)), then the elevator should turn around, start moving up.

$$\begin{aligned}&\textit{e8}\text{: } [\![\textit{now-flr}\ \tau = (n,\ n);\ \neg\ \textit{is-up}\ \tau;\\ &\qquad \exists\ x \in (\textit{up-set}\ \tau \cup \textit{down-set}\ \tau \cup \textit{dest}\ \tau).\ x < n]\!]\\ &\qquad\qquad \Longrightarrow (\tau,\ \textit{StartDown}\ n) \in \textit{elev-cs}\end{aligned}$$

— Rule *e8* describes the behavior of the elevator. It is the dual of *e6*, but is used to generate *StartDown n*.

$$\begin{aligned}&\textit{e9}\text{: } [\![\textit{now-flr}\ \tau = (n,\ n);\ \ \textit{is-up}\ \tau;\\ &\qquad \exists\ x \in (\textit{up-set}\ \tau \cup \textit{down-set}\ \tau \cup \textit{dest}\ \tau).\ x < n;\\ &\qquad\quad \forall\ x \in (\textit{up-set}\ \tau \cup \textit{down-set}\ \tau \cup \textit{dest}\ \tau).\ x \leq n]\!]\\ &\qquad\qquad \Longrightarrow (\tau,\ \textit{StartDown}\ n) \in \textit{elev-cs}\end{aligned}$$

— Rule *e9* describes the behavior of the elevator. It is the dual of *e7*, but is used to generate *StartDown*.

$$\begin{aligned}&\textit{e10}\text{: } [\![\textit{now-flr}\ \tau = (n,\ n{+}1);\ (n{+}1) \in \textit{up-set}\ \tau \cup \textit{dest}\ \tau]\!]\\ &\qquad\qquad \Longrightarrow (\tau,\ \textit{Stop}\ n\ (n{+}1)) \in \textit{elev-cs}\end{aligned}$$

— Rule *e10* describes the behavior of the elevator. When the elevator is moving from floor *n* to floor *n+1*, the elevator will stop at floor *n+1* if there is somebody inside the elevator who wants to get off at floor *n+1* or there is somebody at floor *n+1* who wants to go upward.

$$\begin{aligned}&\textit{e11}\text{: } [\![\textit{now-flr}\ \tau = (n,\ n{+}1);\ \forall\ x \in \textit{up-set}\ \tau \cup \textit{down-set}\ \tau \cup \textit{dest}\ \tau.\ x \leq (n{+}1)]\!]\\ &\qquad\qquad \Longrightarrow (\tau,\ \textit{Stop}\ n\ (n{+}1)) \in \textit{elev-cs}\end{aligned}$$

— Rule *e11* describes the behavior of the elevator. It describes the case when the elevator is going up from floor *n* to *n+1*, and the premises in *e10* do not hold. If, in this case, all requests in the system are from floors no higher than *n+1*, there is no need for the elevator to go pass floor *n+1*. Therefore, it should stop at floor *n+1*. In this case, there must be somebody at floor *n+1*, who want to go down.

$$\begin{aligned}&\textit{e12}\text{: } [\![\textit{now-flr}\ \tau = (n\ +\ 1,\ n);\ n \in \textit{down-set}\ \tau \cup \textit{dest}\ \tau]\!]\\ &\qquad\qquad \Longrightarrow (\tau,\ \textit{Stop}\ (n\ +\ 1)\ n) \in \textit{elev-cs}\\ &\textit{e13}\text{: } [\![\textit{now-flr}\ \tau = (n\ +\ 1,\ n);\ \forall\ x \in \textit{up-set}\ \tau \cup \textit{down-set}\ \tau \cup \textit{dest}\ \tau.\ n \leq x]\!]\\ &\qquad\qquad \Longrightarrow (\tau,\ \textit{Stop}\ (n\ +\ 1)\ n) \in \textit{elev-cs}\end{aligned}$$

— Rule *e12* and rule *e13* describe the behavior of the elevator. They are the counterparts of rule *e10* and rule *e11*, but for the case when the elevator is moving down.

$$\begin{aligned}&\textit{e14}\text{: } [\![\textit{now-flr}\ \tau = (n,\ n{+}1);\ (n{+}1) \notin \textit{up-set}\ \tau \cup \textit{dest}\ \tau;\\ &\qquad \exists\ x \in \textit{up-set}\ \tau \cup \textit{down-set}\ \tau \cup \textit{dest}\ \tau.\ (n{+}1) < x]\!]\\ &\qquad\qquad \Longrightarrow (\tau,\ \textit{Pass}\ n\ (n{+}1)) \in \textit{elev-cs}\end{aligned}$$

— Rule *e14* describes the behaviour of the elevator. When the elevator is moving from floor *n* to floor *n+1*, the elevator will not stop at floor *n+1* if nobody inside the elevator who wants to get off at floor *n+1* and there is nobody at floor *n+1* who wants to go upward.

$$\begin{aligned}&\textit{e15}\text{: } [\![\textit{now-flr}\ \tau = (n\ +\ 1,\ n);\ n \notin \textit{down-set}\ \tau \cup \textit{dest}\ \tau;\\ &\qquad \exists\ x \in \textit{up-set}\ \tau \cup \textit{down-set}\ \tau \cup \textit{dest}\ \tau.\ x < n]\!]\end{aligned}$$

$$\Longrightarrow (\tau,\ \mathit{Pass}\ (n + \mathit{1})\ n) \in \mathit{elev\text{-}cs}$$

— Rule *e15* describes the behaviour of the elevator. It is the dual of rule *e14*, but for the case when the elevator is moving down.

3 Liveness proof

3.1 Overview

The proved liveness property for the elevator control system is lemma *arrive-will-exit*:

$$\begin{array}{l}\llbracket \mathit{elev\text{-}cs} \vdash \sigma;\\ \ \mathit{PF\ elev\text{-}cs}\\ \ \ (\{\!|F\ p\ m\ n,\ E\ p\ m\ n,\ \mathit{4} * H + \mathit{3}|\!\}\ @\ \{\!|FT\ p\ m\ n,\ ET\ p\ m\ n,\ \mathit{4} * H + \mathit{3}|\!\})\ \sigma\rrbracket\\ \Longrightarrow \sigma \models \Box\langle(\!|\mathit{Arrive}\ p\ m\ n|\!)\rangle\hookrightarrow\Diamond\langle(\!|\mathit{Exit}\ p\ m\ n|\!)\rangle\end{array}$$

the conclusion of *arrive-will-exit* says: if a user p arrives at floor m and wants to go to floor n (represented by the happening of event *Arrive p m n*), then he will eventually get there (represented by the happening of event *Exit p m n*).

The premise of *arrive-will-exit* is a *parametric fairness* assumption. The explanation of *parametric fairness* and the definition of *PF* are given in [11]. In unfair executions, if *StartUp* and *StartDown* are given higher priority than *Enter*, a user will keep missing the elevator. On the other hand, if *Enter* is given higher priority than *StartUp* and *StartDown*, the elevator will fail to move. When the elevator reaches a user's destination, if *StartUp* and *StartDown* have higher priority than *Exit*, the user may fail to get off the elevator. The fairness assumption is necessary to prevent such occasions from happening infinite many times.

The proof of *arrive-will-exit* is split into two lemmas, lemma *will-enter*:

$$\begin{array}{l}\llbracket \mathit{elev\text{-}cs} \vdash \sigma;\ \mathit{PF\ elev\text{-}cs}\ \{\!|F\ p\ m\ n,\ E\ p\ m\ n,\ \mathit{4} * H + \mathit{3}|\!\}\ \sigma\rrbracket\\ \Longrightarrow \sigma \models \Box\langle(\!|\mathit{Arrive}\ p\ m\ n|\!)\rangle\hookrightarrow\Diamond\langle(\!|\mathit{Enter}\ p\ m\ n|\!)\rangle\end{array}$$

which says: if a user p arrives, he will eventually get into the elevator, and lemma *will-exit*:

$$\begin{array}{l}\llbracket \mathit{elev\text{-}cs} \vdash \sigma;\ \mathit{PF\ elev\text{-}cs}\ \{\!|FT\ p\ m\ n,\ ET\ p\ m\ n,\ \mathit{4} * H + \mathit{3}|\!\}\ \sigma\rrbracket\\ \Longrightarrow \sigma \models \Box\langle(\!|\mathit{Enter}\ p\ m\ n|\!)\rangle\hookrightarrow\Diamond\langle(\!|\mathit{Exit}\ p\ m\ n|\!)\rangle\end{array}$$

which says: if the user p gets into the elevator, he will eventually get off. The combination of *will-enter* and *will-exit* into *arrive-will-exit* is done using lemma *trans-resp*:

$$\llbracket\sigma \models \Box\varphi\hookrightarrow\Diamond\psi\ ;\ \sigma \models \Box\psi\hookrightarrow\Diamond\omega\ \rrbracket \Longrightarrow \sigma \models \Box\varphi\hookrightarrow\Diamond\omega$$

and lemma *PF-app-eq*:

$$\mathit{PF\ cs}\ (\mathit{pel}_1\ @\ \mathit{pel}_2)\ \sigma = (\mathit{PF\ cs\ pel}_1\ \sigma \wedge \mathit{PF\ cs\ pel}_2\ \sigma)$$

The real hard work hides in the proof of *will-enter* and *will-exit*, both use the proof rule *resp-rule*, which is derived in [11]:

$[\![RESP\ cs\ F\ E\ N\ P\ Q;\ cs \vdash \sigma;\ PF\ cs\ \{\!|F,\ E,\ N|\!\}\ \sigma]\!] \Longrightarrow \sigma \models \Box\langle P\rangle \hookrightarrow \Diamond\langle Q\rangle$

The application of *resp-rule* requires us to find two functions *?F*, *?E* and a natural number *?N*, such that the *RESP* premise is satisfied. The formal definition of *RESP* is given in [11]. The definition of *RESP* expresses some requirements on the underlying state-transition system. The definition of *RESP* requires *?F* to be a measuring function which returns the distance from the current state to the desired *Q*-state. The *RESP* also requires function *?E* to be a strategy for choosing the next eligible event to happen, so that the happening of the selected *?E*-event will decrease the *?F*-measurement. The *?N* is an upper bound of *?F*. The existence of *?F*, *?E* and *?N* will ensure the desired liveness property.

For the proof of *will-enter*, the assignment of *?F*, *?E* and *?N* is:

$$?F \mapsto F\ p\ m\ n,\ ?E\ \mapsto E\ p\ m\ n,\ ?N \mapsto 4{*}H\ +\ 3$$

where, *H* is the building height. It has been formally proved (in lemma *FE-resp*) that:

$RESP\ elev\text{-}cs\ (F\ p\ m\ n)\ (E\ p\ m\ n)\ (4\ *\ H\ +\ 3)\ (\!|Arrive\ p\ m\ n|\!)\ (\!|Enter\ p\ m\ n|\!)$

In subsection 3.2, we are going to explain how *F* and *E* are found, with the help of state-transition diagram, using the heuristic described in [11]. And then, in subsection 3.3, the proof structure of *FE-resp* is explained briefly.

For the proof of *will-exit*, the assignment of *?F*, *?E* and *?N* is:

$$?F \mapsto FT\ p\ m\ n,\ ?E\ \mapsto ET\ p\ m\ n,\ ?N \mapsto 4{*}H\ +\ 3$$

and it has been formally proved (in lemma *FET-resp*) that:

$RESP\ elev\text{-}cs\ (FT\ p\ m\ n)\ (ET\ p\ m\ n)\ (4\ *\ H\ +\ 3)\ (\!|Enter\ p\ m\ n|\!)\ (\!|Exit\ p\ m\ n|\!)$

The finding of *FT* and *ET* and the proof of *FET-resp* are omitted, because they are similar to the finding of *F* and *E* and the proof of *FE-resp*.

3.2 The finding of F and E

The state-transition diagram, which inspired us to find *F* and *E*, is shown in Figure 3. Figure 3 is for a 5-floor system (floors *0*, *1*, *2*, *3*, *4*). The diagram is from the view of a user *p*, who arrives at floor *2* and plans to go to floor *4*. The happening of *Arrive p 2 4* brings the system from state $Arrive\ p\ 2\ 4 \notin arr\text{-}set\ \tau$ to $Arrive\ p\ 2\ 4 \in arr\text{-}set\ \tau \wedge 2 \notin up\text{-}set\ \tau$. The latter state is the one after user *p*'s arrival, but before he presses the ▲-button on floor *2*. The next event to happen is the pressing of ▲-button on floor *2* (the event *Up 2*), which brings the system into the macro-state $Arrive\ p\ 2\ 4 \in arr\text{-}set\ \tau \wedge 2 \in up\text{-}set\ \tau$.

The macro-state $Arrive\ p\ 2\ 4 \in arr\text{-}set\ \tau \wedge 2 \in up\text{-}set\ \tau$ is refined into many sub-states, with each sub-state reflects the current pass and the value of *now-flr*. When the elevator is on the move, the value of *now-flr* uniquely identifies the state. For example, the state marked with (*n*, *n+1*) means the elevator is in

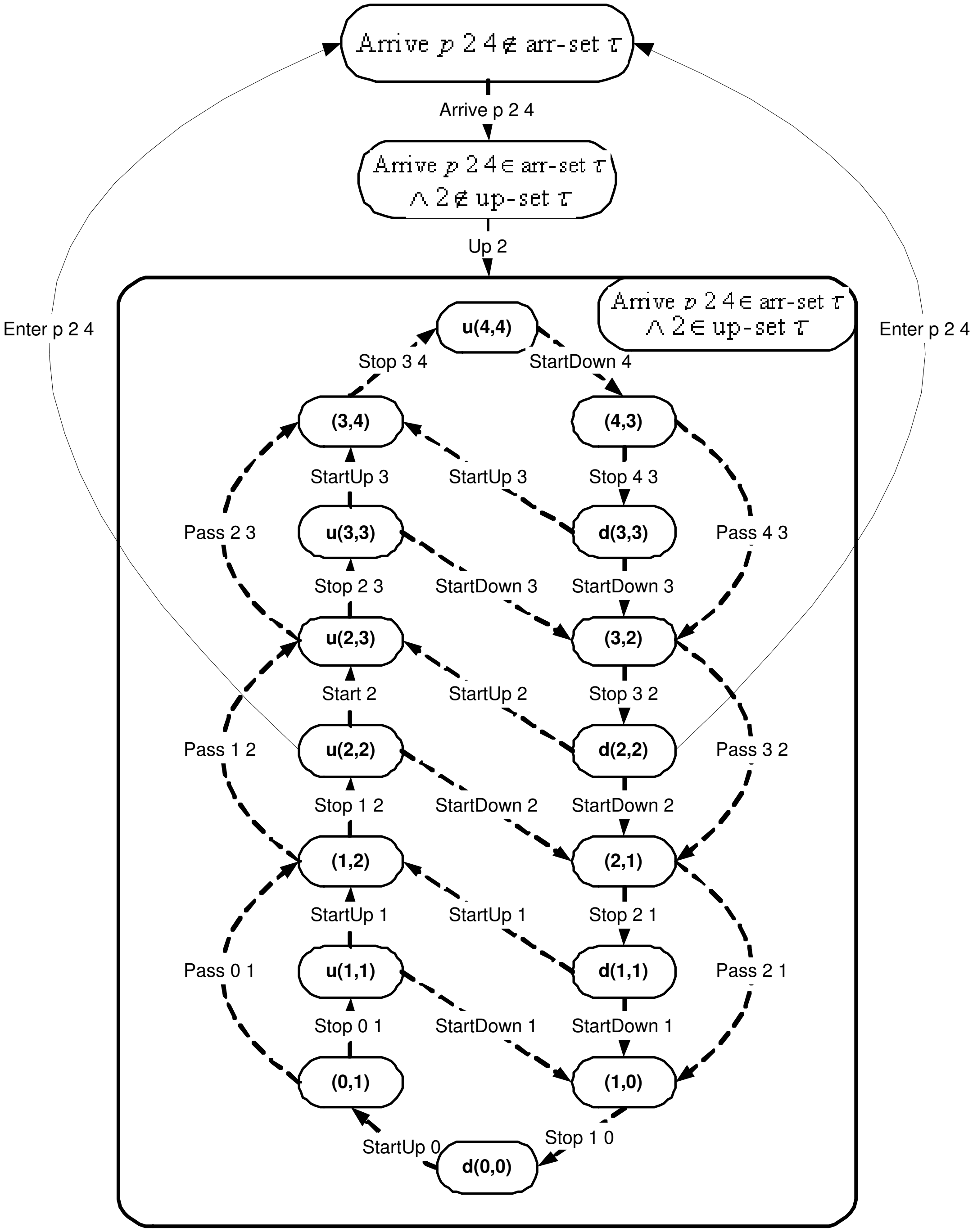


Fig. 3. State-transition diagram for *Arrive p 2 4*

Up-pass and is moving from floor n to floor $n+1$. Similarly, the state marked with $(n,\ n-1)$ means the elevator is in *Down*-pass and is moving from floor n to floor $n-1$. When the elevator is stopped, the value of *now-flr* is $(n,\ n)$ for some n. In this case, the value of *now-flr* alone can not identify the current pass. Therefore, an u or d is added before $(n,\ n)$ to identify the current pass. The state marked with $u(n,\ n)$ means the elevator is stopped at floor n, and is currently in *Up*-pass. The state marked with $d(n,\ n)$ means the elevator is stopped at floor n, and is currently in *Down*-pass. For top floor *4*, there is no state $d(4,\ 4)$. Similarly, for ground floor *0*, there is no state $u(0,\ 0)$.

In Figure 3, the outgoing edges attached to each state represents all events, the happening of which will lead the system out of the state. Those events, which do not change system state are omitted. Once the system gets into a state, there must be at least one outgoing edge, which is eligible to happen next. The subset of outgoing edges, which are eligible to happen next, may change with the happening of events. For example, suppose the system is currently in state $u(2,\ 2)$, then, either *StartUp 2* or *StartDown 2* is eligible to happen next. At first, if there is no request for the elevator to go higher than floor *2*, the next event eligible to happen is *StartDown 2*. Then, the happening of a *Up 3* event will change the next eligible event from *StartDown 2* to *StartUp 2*, because, now, there is a request at floor *3*. At state $u(2,\ 2)$, in addition to *StartDown 2* and *StartUp 2*, the event *Enter p 2 4* is also eligible to happen, because user p wants to go from floor *2* to floor *4* and the happening of *Enter p 2 4* will get user p onboard the elevator. The happening of *Enter p 2 4* will lead the system into the desired state: $\textit{Arrive p 2 4} \notin \textit{arr-set}\ \tau \wedge \textit{Enter p 2 4} \in \textit{ent-set}\ \tau$. In state $u(2,\ 2)$, the happening of *StartUp 2* or *StartDown 2* means user p missed the elevator. The fairness assumption ensures that user p can not always miss the elevator.

The spanning tree of Figure 3, which inspires the definition of F and E is shown in Figure 4. It is actually a directed acyclic graph (DAG), the spanning tree with some short cuts added. A top-sorting of the graph gives F-values, as marked to the left of each state. The DAG is formed by retaining the shortest path from each state to the desired state. Some states in Figure 3 have more than one outgoing edges, because the shortest path from that state depends on more refined information. Suppose the system is now at state $u(3,\ 3)$, if there are some needs above floor *3*, then event *StartUp 3* is the one on shortest path; otherwise, *StartDown 3* is the one on shortest path. Function E chooses the right event based on these more refined information.

In Figure 4, some states are assigned F-values higher than necessary. This is just to make the formal definition of F more concise. The increase of F-values will not affect the desired properties of $F\ p\ 2\ 4$.

The correctness of the heuristic to construct F and E will finally be judged by formal proofs. The formal definitions of $F\ p\ m\ n$ and $E\ p\ m\ n$ are shown in Figure 5 and Figure 6 respectively. The m and n in Figure 5 and Figure 6 can

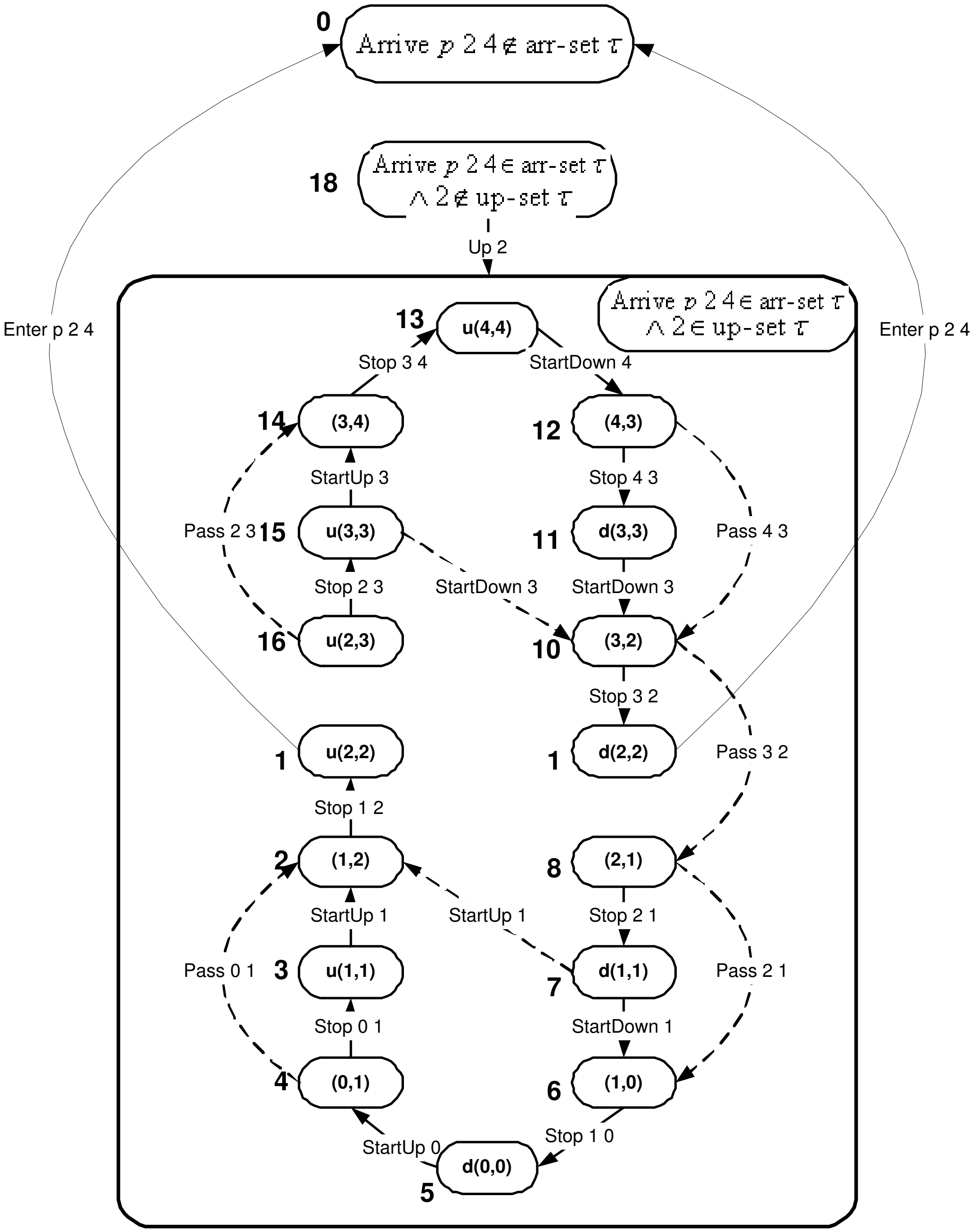


Fig. 4. Spanning tree for *Arrive p 2 4*

be viewed as the generalization of 2 and 4 in *Arrive p 2 4*. The H can be viewed as the generalization of the building height 4 above.

It can be checked by the reader that the definition of $F\ p\ m\ n$ in Figure 5 gives exactly the values illustrated in Figure 4. It is also noticeable that the evaluation of $F\ p\ 2\ 4$ only involves the upper-half of Figure 5. The lower-half of Figure 5 deals with the case when $n < m$, for example: $n \mapsto 3$, $n \mapsto 2$. However, the spirit of the lower half of Figure 5 is still the same as upper half, and therefore, will not be explained further.

```
constdefs F :: user ⇒ nat ⇒ nat ⇒ event list ⇒ nat
  F p m n τ ≡
  ( if Arrive p m n ∉ arr-set τ then 0
    else if now-flr τ = (m, m) then 1
    else if (m < n ∧ m ∉ up-set τ) then 4∗H + 2
    else if (¬ m < n ∧ m ∉ down-set τ) then 4∗H + 2
    else let (k1, k2) = now-flr τ in
         if m < n then
            if (¬ is-up τ) then
               if k1 = k2 then 2∗m + 2∗k1 + 1
                       else 2∗m + 2∗k1
            else if k1 < m then
                   if k1 = k2 then 2∗m − 2∗k1 + 1
                           else 2∗m − 2∗k1
                 else if k1 = k2 then 2∗m + 4∗H − 2∗k1 + 1
                           else 2∗m + 4∗H − 2∗k1
         else if (¬ is-up τ) then
                 if k1 ≤ m then
                   if k1 = k2 then 4∗H − 2∗m + 2∗k1 + 1
                           else 4∗H − 2∗m + 2∗k1
                 else if k1 = k2 then 2∗k1 − 2∗m + 1
                           else 2∗k1 − 2∗m
              else if k1 = k2 then 4∗H − 2∗m − 2∗k1 + 1
                         else 4∗H − 2∗m − 2∗k1)
```

Fig. 5. The definition of *F*

```
constdefs E :: user ⇒ nat ⇒ nat ⇒ event list ⇒ event
  E p m n τ ≡
  ( if Arrive p m n ∉ arr-set τ then (StartUp m)
    else if now-flr τ = (m, m) then (Enter p m n)
    else if (m < n ∧ m ∉ up-set τ) then (Up m)
    else if (¬ m < n ∧ m ∉ down-set τ) then (Down m)
    else let (k1, k2) = now-flr τ in
         if (¬ is-up τ) then
             if k1 = k2 then
                 if (∃ x ∈ (up-set τ ∪ down-set τ ∪ dest τ). x < k1)
                          then (StartDown k1)
                          else (StartUp k1)
             else
                 if (k2 ∉ down-set τ ∪ dest τ) ∧
                    (∃ x ∈ up-set τ ∪ down-set τ ∪ dest τ. x < k2)
                          then (Pass k1 k2)
                          else (Stop k1 k2)
         else if k1 = k2 then
                 if (∃ x ∈ (up-set τ ∪ down-set τ ∪ dest τ). k1 < x)
                          then (StartUp k1)
                          else (StartDown k1)
             else
                 if (k2 ∉ up-set τ ∪ dest τ) ∧
                    (∃ x ∈ up-set τ ∪ down-set τ ∪ dest τ. k2 < x)
                          then (Pass k1 k2)
                          else (Stop k1 k2))
```

Fig. 6. The definition of *E*

The definition of $E\ p\ m\ n$ follows the same structure as $F\ p\ m\ n$, except that the decisions in $E\ p\ m\ n$ does not depend on whether $n < m$ or $m < n$. Given the state-transition diagram in Figure 3, the definition of $E\ p\ m\ n$, at lowest level, is just to describe which outgoing edge to choose depending on more refined state information. The $E\ p\ m\ n$ is such defined that the returned event always follows the shortest path. Such a strategy will ensure that the happening of an E-event will always lead to a state with lower F-value.

3.3 The proof of *FE-resp*

Using the introduction rule of *RESP*, the proof of *FE-resp* splits into two lemmas, lemma *F-mid*:

$$\llbracket elev\text{-}cs \vdash \tau;\ \lceil (\!|Arrive\ p\ m\ n|\!) \longmapsto \neg (\!|Enter\ p\ m\ n|\!) * \rceil\ \tau;\ \neg\ (\!|Enter\ p\ m\ n|\!)\ \tau \rrbracket$$
$$\Longrightarrow 0 < F\ p\ m\ n\ \tau \wedge F\ p\ m\ n\ \tau < 4 * H + 3$$

corresponding to the *mid* assumption in locale *RESP*, and lemma *FE-fd*:

$$\llbracket elev\text{-}cs \vdash \tau;\ 0 < F\ p\ m\ n\ \tau \rrbracket$$
$$\Longrightarrow \tau\ [elev\text{-}cs{>}\ E\ p\ m\ n\ \tau \wedge F\ p\ m\ n\ (E\ p\ m\ n\ \tau\ \#\ \tau) < F\ p\ m\ n\ \tau$$

corresponding to the *fd* assumption in locale *RESP*.

The proof of *F-mid* further splits into two lemmas, lemma *pre-nar*:

$$\llbracket \lceil (\!| \mathit{Arrive}\ p\ m\ n |\!) \longmapsto \neg (\!| \mathit{Enter}\ p\ m\ n |\!) * \rceil\ \tau;\ \neg\ (\!| \mathit{Enter}\ p\ m\ n |\!)\ \tau \rrbracket$$
$$\Longrightarrow \mathit{Arrive}\ p\ m\ n \in \mathit{arr\text{-}set}\ \tau$$

which reduces the states between the happening of *Arrive p m n* and the happening of *Enter p m n* to state: *Arrive p m n* ∈ *arr-set* τ, and lemma *ar-mid*:

$$\llbracket \mathit{elev\text{-}cs} \vdash \tau;\ \mathit{Arrive}\ p\ m\ n \in \mathit{arr\text{-}set}\ \tau \rrbracket$$
$$\Longrightarrow \mathit{0} < F\ p\ m\ n\ \tau \wedge F\ p\ m\ n\ \tau < \mathit{4} * H + \mathit{3}$$

The proof of both *FE-fd* and *ar-mid* follow the structure of *F p m n* using a series of *cases* commands. The proof is straight forward, with the state-transition diagram of Figure 3 and 4 in mind. The proof of *FE-fd* is the lengthiest one, which takes *1269* lines of Isabelle/Isar and several days to complete. The *arith* command is heavily used to solve elementary number theory goals.

4 Conclusion

The formal verification described in this paper is written in a structured, human readable style, using Isabelle/HOL/Isar. Our experiment shows that: using the method proposed in [11], the liveness problem of elevator control system can be treated with a reasonable amount of human efforts. Table 2 lists the composition of the formal proof. The effort to derive *arrive-will-exit* from *will-enter* and *will-exit* takes only a dozen of lines, therefore, is not listed in Table 2. In Table 2, the working days used to construct definitions is marked as ∗, because we can not calculate accurately how many days are spent. The liveness proof of the elevator control system started in June 2004. It has gone through several major revisions. It has been several years since we started considering liveness proof for inductive protocol verification in general.

Contents	**Nr. of lines**	**Nr. of lemmas**	**Nr. of working days**
Definitions of the elevator control system, *F*, *E*, *FT* and *ET*	≤ 200	–	*
Safety properties	579	12	2
Proof of *will-enter*, including some preliminary lemmas	2012	10	5
Proof of *will-exit*, including some preliminary lemmas	1029	8	3

Table 2. Summary of Isabelle/Isar proof scripts

Our experience shows, once an understanding of the underlying problem is obtained, construction of formal proof doesn't take much time. The most difficult part of a formal proof is to come up with a proper abstraction, which can keep the complexity of formal proof tractable. The finding of such an abstraction has to be carried out case by case, and each may take indefinitely long time. We believe that we have found such an abstraction for the liveness of elevator control system, perhaps, for the liveness verification in general.

References

1. F. Andersen, U. Binau, K. Nyblad, K. D. Petersen, and J. S. Pettersson. The HOL-UNITY verification system. In *Proc. of the Sixth International Joint Conference CAAP/FASE: TAPSOFT'95-Theory and Practice of Software Development*, pages 795–796, Aarhus, Denmark, 1995.
2. J. S. Dong, L. Zucconi, and R. Duke. Specifying parallel and distributed systems in Object-Z. In G. Agha and S. Russo, editors, *The 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 140–149, Boston, Massachusetts, 1997. IEEE Computer Society Press.
3. D. N. Dyck and P. E. Caines. The logical control of an elevator. *IEEE Trans. on Automatic Control*, 40(3):480–486, 1995.
4. M. M. Gallardo, M. Martinez, P. Merino, and E. Pimentel. αSPIN: Extending SPIN with abstraction. *Lecture Notes in Computer Science*, 2318:254–??, 2002.
5. B. Heyd and P. Cregut. A modular coding of Unity in Coq. *Lecture Notes in Computer Science*, 1125:251–266, 1996.
6. M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *Proceedings: 17th International Conference on Software Engineering*, pages 15–24. IEEE Computer Society Press / ACM Press, 1995.
7. M. Kaltenbach. Interactive verification exploiting program design knowledge: A model-checker for UNITY. Technical Report CS-TR-96-22, The University of Texas at Austin, Department of Computer Sciences, January 1 1997. Mon, 28 Apr 103 21:06:36 GMT.
8. L. Paulson. Mechanizing UNITY in Isabelle. *ACM Trans. Comput. Logic*, 1(1):3–32, 2000.
9. Frank Strobl and Alexander Wisspeintner. Specification of an elevator control system – an AutoFocus case study. Institutsbericht, Technische Universitaet Muenchen, Institut fuer Informatik, March 1999.
10. W. G. Wood. Temporal logic case study. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 257–263, Berlin, June 1990. Springer.
11. X.Zhang, H.Yang, and Y.Wang. Liveness reasoning for inductive protocol verification. In *The 'Emerging Trend' proceeding of TPHOLs 2005*, 2005.
12. Ying Zhang and Alan K. Mackworth. Design and analysis of embedded real-time systems: An elevator case study. Technical report, March 01 1994.

Verification of Euclid's Algorithm for Finding Multiplicative Inverses

Junxing Zhang and Konrad Slind

School of Computing, University of Utah
junxing@cs.utah.edu
slind@cs.utah.edu

Abstract. Multiplicative inverses have been widely used in cryptography. The basic method to find multiplicative inverses is to use Euclid's Algorithm. When used for this purpose, several changes have to be made to the algorithm in order to make it more effective in finding these inverses. In this work, we use HOL-4 theorem prover to specify and verify Euclid's Algorithm as it is used in finding multiplicative inverses.

1 Introduction

Although Euclid's Algorithm is best known for finding greatest common divisors, it can also be used to find multiplicative inverses. For example, the secret-key cryptographic algorithm IDEA [1991] uses it for this purpose. However, the general Euclid's Algorithm doesn't guarantee the existence of the multiplicative inverse for any given number with respect to any modulus. Therefore, cryptographic algorithms specialize Euclid's Algorithm with certain parameter and modulus values to ensure the existence of multiplicative inverses for a range of input values. Further, in order to use the specialized Euclid's Algorithm in computer computations, additional changes also have to be made to the algorithm.

In this paper, we use the HOL-4 [Kananaskis 3] theorem prover to verify the correctness of the general Euclid's Algorithm, specialized Euclid's Algorithm and the "computerized" version of the algorithm as they are used in the calculation of multiplicative inverses. The specialized and "computerized" versions are based on IDEA. With the proof of Euclid's Algorithm, the verification of IDEA is trivial. We will show it in the appendix. The proof in this work not only can aid in the verification of IDEA, but also can shed light for the verification of other algorithms that use multiplicative inverses such as RSA [1977].

2 Euclid's Algorithm

Euclid's Algorithm is originally used to find the greatest common divisor of two integers. With some extension, however, we can use it to find multiplicative inverses modulo n. The *multiplicative inverse* of a number is defined as the number whose product with the original number equals to one under the modulus

n. We describe the general Euclid's Algorithm used for finding inverses as follows. We start with two natural numbers r_1 and r_2, and they can be represented as:

$$r_1 = u_1 * x + v_1 * y \tag{1}$$

$$r_2 = u_2 * x + v_1 * y \tag{2}$$

u_1, u_2, v_1, v_2, x, and y are integers. First, if $r_1 < r_2$, then we subtract equation 1 from equation 2; otherwise, we subtract 2 from 1. Second, we use the resulting equation and one of the previous equations that has the smaller r as the new equations 1 and 2. We repeat these two steps until r_1 or r_2 equals to zero. This scenario will happen because if a natural number continuously decreases it will become zero. Another fact is the original r_1 and r_2 have the same greatest common divisor as the final r_1 and r_2. Because one of the final r_1 and r_2 is zero, the other must be the greatest common divisor. This is the rationale that ensures Euclid's Algorithm can find the greatest common divisor of any r_1 and r_2. Then, how do we use it to find multiplicative inverses? Well, we use a special case of the original rationale. If r_1 and r_2 are relatively prime, their greatest common divisor will be one. One of the final equations will be of the form $1 = u * x + v * y$. If we mod both sides of the equation with y, it will become $1\%y = (u * x)\%y$, so we find u is the multiplicative inverse of x modulo y. Now suppose we want to find the multiplicative inverse of x modulo y, and x and y are relatively prime. In order to use Euclid's Algorithm, we need a way to construct the equations 1 and 2. We simply assign the values 1, 0, 0, 1, x and y to variables u_1, u_2, v_1, v_2, r_1 and r_2. So, the equations become:

$$x = 1 * x + 0 * y \tag{3}$$

$$y = 0 * x + 1 * y \tag{4}$$

In the equations 3 and 4, x and y are used as natural numbers at the left-hand side (due to the assignment to r_1 and r_2) but integers at the right-hand side. To meet both requirements, they should be natural numbers, because natural numbers can be used as integers not otherwise. This is one difference between 1, 2 and 3, 4. We will also show this difference in the general and specialized specifications of Euclid's Algorithm. It should be noted that x has the multiplicative inverse under the modulus y only if they are relatively prime.

2.1 Specification

We divide the specification into two parts. The general specification is based on equations 1 and 2, and it is mainly used for reasoning about the greatest common divisor. The specialized specification is based on equations 3 and 4, and it is used for reasoning about the multiplicative inverse.

General Euclid's Algorithm. We specify the general Euclid's Algorithm as two functions, one is the simple subtraction between the equations 1 and 2, and another is the iteration of the first function. This separation makes the specification more clear, and verification easier because the verification of the two functions can focus on different aspects of the algorithm.

```
val dec_def =
 Define
   `dec (r1:num, r2:num, u1:int, u2:int, v1:int, v2:int) =
      if r1 < r2
        then (r1, r2 - r1, u1, u2 - u1, v1, v2 - v1)
        else (r1 - r2, r2, u1 - u2, u2, v1 - v2, v2)`;
val (inv_def,inv_ind) =
  Defn.tprove
    (Hol_defn
      "inv"
      `inv (r1, r2, u1, u2, v1, v2) =
      if ((FST (r1, r2, u1, u2, v1, v2) = 1) \/
        (FST (SND (r1, r2, u1, u2, v1, v2)) = 1) \/
        (FST (r1, r2, u1, u2, v1, v2) = 0) \/
        (FST (SND (r1, r2, u1, u2, v1, v2)) = 0))
        then (r1, r2, u1, u2, v1, v2)
        else inv (dec (r1, r2, u1, u2, v1, v2))`,
  WF_REL_TAC `measure (\(a,b,c,d,e,f). a+b)` THEN
  RW_TAC arith_ss [dec_def]);
val P_def = Define
  `P ((r1:num, r2:num, u1:int, u2:int, v1:int, v2:int), x:int, y:int) =
  (((int_of_num r1) = u1*x + v1*y) /\ ((int_of_num r2) = u2*x + v2*y))`;
```

The two functions, *dec* and *inv*, only define the operations on u_1, u_2, v_1, v_2, r_1 and r_2, but they cannot specify the relations among them. Thus, we need the predicate P to show that the equations 1 and 2 exists among these variables. It should be noted that the induction function *dec* stops when r_1 or r_2 equals to zero or one, but for equations 1 and 2, the induction should stop only when r_1 or r_2 equals to zero. The reason for this change is that our ultimate goal is to find the multiplicative inverse, not the greatest common divisor. As discussed in the beginning of this section, for searching the multiplicative inverse, it is enough to stop when r1 or r2 equals to one. We still need to use r_1 or r_2 equals to zero as another stopping condition, because the algorithm doesn't ensure that r_1 or r_2 will become one for any given x and y. Our use of FST and SND on the list (r1, r2, u1, u2, v1, v2) seems unnecessary, but it helps HOL's symbolic expansion in the verification when the input of *inv* is not a tuple, but a function that returns a tuple.

Specialized Euclid's Algorithm The specialized algorithm is based on the equations 3 and 4, but it has one change. It uses a specific value 65537 for the variable y, because we want to verify multiplicative inverses for this modulus only. We choose this special modulus, because it is a prime. Since any nonzero natural number that is less than a prime is relatively prime to this prime, we can ensure that all natural numbers from 1 to 65536 have multiplicative inverses modulo 65537. Multiplicative inverses of other numbers or under other modulus can be verified with the similar approach as the one we will give, provided numbers and modulus are relatively prime.

```
val ir1_def = Define `(ir1 x) = FST (inv (x, 65537, 1, 0, 0, 1))`;
val ir2_def = Define `(ir2 x) =
  FST (SND (inv (x, 65537, 1, 0, 0, 1)))`;
val iu1_def = Define `(iu1 x) =
  FST (SND (SND (inv (x, 65537, 1, 0, 0, 1))))`;
val iu2_def = Define `(iu2 x) =
  FST (SND (SND (SND (inv (x, 65537, 1, 0, 0, 1)))))`;
```

The specialized algorithm is defined on top of the general algorithm, because it calls the function *inv*. The definition only extracts the final values of u_1, u_2, r_1 and r_2, because only they are useful in the verification of inverses.

2.2 Verification

The verification has three parts. In the first part, we create lemmas from theorems in several standard theories. These lemmas help the verification of the algorithm, but their definitions are very general, so we don't group them together with proofs that are specific to the algorithm. In the second part, we prove theorems of the general algorithm. In the third part, we specialize some theorem in the second part and extend them to show more attributes of the specialized algorithm.

Extension to Standard Theories First, we extend the mod operation in the Integer Theory. i16_Lemma3 shows that $(a + b * c)\%c = a\%c$. It is used to prove the equivalence of $r = u * x + v * y$ and $r\%y = (u * x)\%y$ when y is not zero. Then, we extend the theory *divides* and *gcd*. minv_Lemma1, minv_Lemma2, and minv_Lemma3 show that the greatest common divisor of any number from 1 to 65536 and 65537 is 1. gcd_Lemma1 and gcd_Lemma2 are used to prove the greatest common divisor of r_1 and r_2 is always the same no matter how *inv* and *dec* functions change them.

```
i16_Lemma3:
   `!a:int b:int c:int. ~(c = 0) ==> (((a + b * c) % c) = (a % c))`
minv_Lemma1:
    `!a b. b < a ==> ~(0 < b) \/ ~divides a b`
minv_Lemma2:
    `!x:num. ((0 < x) /\ (x < 65537)) ==> ~(divides 65537 x)`
minv_Lemma3:
    `!x:num. ((0 < x) /\ (x < 65537)) ==> ((gcd x 65537) = 1)`
gcd_Lemma1:
    `!a b. b <= a ==> (gcd (a-b) b = gcd a b)`
gcd_Lemma2:
    `!a b. a <= b ==> (gcd a (b-a) = gcd a b)`
```

General Algorithm Verification The theorems in this part prove the following facts:

1. decP_Theorem and invP_Theorem show that if (r1, r2, u1, u2, v1, v2), x and y satisfy the equations 1 and 2, they still satisfy these equations after their values have been changed by the functions *dec* and *inv*.
2. i16_Lemma4 and i16_Lemma7 prove that if (r1, r2, u1, u2, v1, v2), x and y satisfy the equations 1 and 2 and y is not zero then $r1\%y = (u1 * x)\%y$ and $r2\%y = (u2 * x)\%y$ even after (r1, r2, u1, u2, v1, v2) are changed by the function *inv*.
3. i16_Lemma5 shows that the *inv* function changes r1 or r2 to 1 or 0.
4. minv_Lemma4 and minv_Lemma6 prove that the r1 and r2 keep the same greatest common divisor even after they have been changed by *dec* and *inv* functions.

With the above facts 2 and 3, we can draw the conclusion that $(u1*x)\%y = 1$ or $(u2 * x)\%y = 1$ or $(u1 * x)\%y = 0$ or $(u2 * x)\%y = 0$, which is very close to the proof of the multiplicative inverse that we want. But we don't want to draw this conclusion here, because our ultimate goal to prove the correctness of multiplicative inverses produced by the specialized algorithm, and thus we delay this conclusion until the next subsection. The fact 1 and 4 will also be used by the next subsection.

```
decP_Theorem:
    `!r1 r2 u1 u2 v1 v2 x y. (P ((r1, r2, u1, u2, v1, v2), x, y))
    ==> (P (dec (r1, r2, u1, u2, v1, v2), x, y))`
invP_Theorem:
    `!r1 r2 u1 u2 v1 v2 x y. (P ((r1, r2, u1, u2, v1, v2), x, y))
    ==> (P (inv (r1, r2, u1, u2, v1, v2), x, y))`
i16_Lemma4:
    `!r1 r2 u1 u2 v1 v2 x y. (P ((r1, r2, u1, u2, v1, v2), $& x, y)
    /\ ~(y = 0))  ==>
    (((int_of_num (FST (r1, r2, u1, u2, v1, v2))) % y =
    (FST (SND (SND (r1, r2, u1, u2, v1, v2))) * $& x) % y) /\
    ((int_of_num (FST (SND (r1, r2, u1, u2, v1, v2)))) % y =
    (FST (SND (SND (SND (r1, r2, u1, u2, v1, v2)))) * $& x) % y))`
i16_Lemma7:
    `!r1 r2 u1 u2 v1 v2 x y. (P ((r1, r2, u1, u2, v1, v2), $& x, y)
    /\ ~(y = 0)) ==>
    (((int_of_num (FST (inv(r1, r2, u1, u2, v1, v2)))) % y =
    (FST (SND (SND (inv (r1, r2, u1, u2, v1, v2)))) * $& x) % y) /\
    ((int_of_num (FST (SND (inv (r1, r2, u1, u2, v1, v2))))) % y =
    (FST (SND (SND (SND (inv (r1, r2, u1, u2, v1, v2))))) * $& x) % y))`
i16_Lemma5:
    `!r1 r2 u1 u2 v1 v2.
    (FST (inv (r1,r2,u1,u2,v1,v2)) = 1) \/
    (FST (inv (r1,r2,u1,u2,v1,v2)) = 0) \/
    (FST (SND (inv (r1,r2,u1,u2,v1,v2))) = 1) \/
    (FST (SND (inv (r1,r2,u1,u2,v1,v2))) = 0)`
```

```
minv_Lemma4:
    `!r1:num r2:num u1:int u2:int v1:int v2:int. gcd r1 r2 =
    gcd (FST(dec(r1, r2, u1, u2, v1, v2)))
        (FST(SND(dec(r1, r2, u1, u2,v1, v2))))`
minv_Lemma6:
    `!r1:num r2:num u1:int u2:int v1:int v2:int. gcd r1 r2 =
    gcd (FST(inv(r1, r2, u1, u2, v1, v2)))
    (FST(SND(inv(r1, r2, u1, u2,v1, v2))))`
```

Specialized Algorithm Verification The theorems in this part prove the following facts:

1. i16_Lemma1 shows that the parameters that specialize the algorithm satisfy the equation 1 and 2 (They turn 1 and 2 into 3 and 4). i16_Lemma2 proves that even after these parameters are changed by the function *inv* they still satisfy these equations. This is a special case of the fact 1 proved in the previous subsection.
2. i16_Lemma8 shows that after r1, r2, u1 and u2 are changed by the function *inv* they still satisfy $r1\%65537 = (u1 * x)\%65537$ and $r2\%65537 = (u2 * x)\%65537$. This is a special case of the fact 2 proved in the previous subsection.
3. i16_Lemma6 proves that the *inv* function changes r1 or r2 to 1 or 0. This is a special case of the fact 3 proved in the previous subsection.
4. With the above facts 2 and 3, we draw the conclusion that $(u1*x)\%65537 = 1$ or $(u2*x)\%65537 = 1$ or $(u1*x)\%65537 = 0$ or $(u2*x)\%65537 = 0$ after r1, r2, u1 and u2 are changed by the function *inv*. It is proved in i16_Lemma9. i16_Lemma10 just exchanges the positions of u1 (or u2) and x in i16_Lemma9. It provides some convenience to the later verification of multiplicative inverse related attributes, because u1 or u2 is the inverse found by the algorithm and the inverse is usually multiplied after the original number.
5. Based on the fact 4 proved in the previous subsection, we conclude that after r1 and r2 are changed by the function *inv* their greatest common divisor still equals to the greatest common divisor of the initial r1 and r2 that equal to x and 65537.
6. From section 2.2.1 we know that the greatest common divisor of any number x from 1 to 65536 and 65537 is 1, so based on the fact 5 above we can conclude that, for these x values, after r1 and r2 are changed by the function *inv* their greatest common divisor is 1. From here we can further conclude that r1 or r2 equals to 1, because the fact 3 above says the *inv* function changes r1 or r2 to 1 or 0. This is proved in minv_Lemma8.
7. For any number x from 1 to 65536, the above fact 6 says the *inv* function changes r1 or r2 to 1. If we reapply the fact 2, we will have $(u1*x)\%65537 = 1$ or $(u2 * x)\%65537 = 1$. This means that if r1 is changed to 1 then u1 is the multiplicative inverse of x modulo 65537; otherwise u2 is the inverse. Therefore, we have proved that the specialized Euclid's Algorithm can find the multiplicative inverse of any number from 1 to 65536!

The first four facts show that u1 or u2 looks like the multiplicative inverse of x modulo 65537, but there is one difference. Their production with x modulo 65537 doesn't always equal to 1, and it could equal to 0, as shown in the fact 4. To make one of them the real inverse, we need to eliminate the possibility in which their production equals to 0. It turns out that we can eliminate this possibility only for x ranging from 1 to 65536, as we demonstrated in the last three facts.

```
i16_Lemma1:
  `!x. P((x, 65537, 1, 0, 0, 1), (int_of_num x), (int_of_num 65537))`
i16_Lemma2:
  `!x. P(inv(x, 65537, 1, 0, 0, 1), (int_of_num x), 65537)`
i16_Lemma6:
  `!x. ((ir1 x) = 1) \/ ((ir1 x) = 0) \/ ((ir2 x) = 1) \/ ((ir2 x) = 0)`
i16_Lemma8:
  `!x. ((int_of_num (ir1 x)) % 65537 =
  ((iu1 x) * (int_of_num x)) % 65537 ) /\
  ((int_of_num (ir2 x)) % 65537 = ((iu2 x) * (int_of_num x)) % 65537 )`
i16_Lemma9:
  `!x. (((iu1 x) * (int_of_num x)) % 65537 = 0) \/
  (((iu1 x) * (int_of_num x)) % 65537 = 1) \/
  (((iu2 x) * (int_of_num x)) % 65537 = 0) \/
  (((iu2 x) * (int_of_num x)) % 65537 = 1) `
i16_Lemma10:
  `!x. (((int_of_num x) * (iu1 x)) % 65537 = 0) \/
  (((int_of_num x) * (iu1 x)) % 65537 = 1) \/
  (((int_of_num x) * (iu2 x)) % 65537 = 0) \/
  (((int_of_num x) * (iu2 x)) % 65537 = 1)`
minv_Lemma7:
  `!x. gcd (ir1 x) (ir2 x) = gcd x 65537`
minv_Lemma8:
  `!x. ((0 < x) /\ (x < 65537)) ==> (((ir1 x) = 1) \/ ((ir2 x) = 1))`
minv_Lemma9:
  `!x. ((0 < x) /\ (x < 65537)) ==>
  (((int_of_num x) * (iu1 x)) % 65537 = 1) \/
  (((int_of_num x) * (iu2 x)) % 65537 = 1)`
```

3 Euclid's Algorithm in Computer Computation

The specialized Euclid's Algorithm has been adapted to find multiplicative inverses using computers. In this section, we will verify that this adapted version is still correct. The adapted version works with 16-bit words and uses them as unsigned numbers. It makes three changes to the specialized Euclid's Algorithm. First, to ensure the inverse is not negative it returns the inverse modulo 65537 as the inverse. Second, it treats 0 as 65536 to make sure that any 16-bit word value has the inverse and to be able to use the specialized algorithm. Finally, it converts 16-bit values to another type of values to calculate inverses and then converts inverses back to 16-bit values, and it also make these type conversions for multiplication. This change is made because 65537 cannot be represented as a 16-bit word, so another type that can represent 65537 must be used in the inverse calculation and multiplication. We will use integer for this purpose. In the following, we verify the adapted version in two steps. In the first step, we verify the first two changes. And in the second step we verify the third change.

3.1 New Inverse Definition And Encoding

Specification As mentioned at the beginning of the section, the multiplicative inverse is redefined as the original inverse modulo 65537. This change is specified by piu1_def, piu2_def, and minv_def. encode_def and decode_def specify the second change.

```
val piu1_def = Define ‘piu1 x = (iu1 x) % 65537‘;
val piu2_def = Define ‘piu2 x = (iu2 x) % 65537‘;
val minv_def =
  Define ‘minv x =
    if ((int_of_num x) * (piu1 x)) % 65537 = 1
        then piu1 x
        else if ((int_of_num x) * (piu2 x)) % 65537 = 1
          then piu2 x
          else 0‘;
val encode_def = Define ‘encode x:num = if x = 0n then 65536n else x‘;
val decode_def = Define ‘decode x:num = if x = 65536n then 0n else x‘;
```

Verification The first change is proved by the first two facts below. The algorithm can treat 0 as 65536, because the input 16-bit values to the algorithm are from 0 to 65535 and the output values from the algorithm are from 1 to 65536 and thus there are no 65536 in the input and no 0 in the output. This encoding is the second change. It is proved by the facts from 3 to 6.

1. mod_Lemma1 and mod_Lemma2 extend the integer theory for the mod operation. They are used to prove that after u1 and u2 are taken mod of 65537 (becoming pu1 and pu2) one of them is still the inverse of x. This is proved in minv_Lemma10, which is a restatement of minv_Lemma9 using the new u1 and u2 (i.e. pu1 and pu2).
2. The minv is defined to choose the inverse from pu1 and pu2, and thus its return value is the real inverse. minv_Theorem proves that it satisfies the definition of multiplicative inverse.
3. piu_Lemma1, piu_Lemma2, minv_Corollary1, minv_Corollary2 and minv_Corollary3 prove that the output of the algorithm is from 1 to 65536.
4. encode_Lemma1 and encode_Lemma3 show that the input 16-bit values are from 0 to 65535 and they can be encoded into 1 to 65536.
5. encode_Lemma2 proves that the encoding works with the inverse algorithm.
6. decode_Lemma1, decode_Lemma2 and decode_Lemma3 show that encoding and decoding revert each other. This fact is useful to prove the correctness of inverses in the modular multiplication in the next subsection.

```
mod_Lemma1:
  `!a c. (~(c=0) /\ (a % c = 0)) ==> (~a % c = 0)`
mod_Lemma2:
  `!a b c. ~(c=0) ==> ((a * b % c) % c = (a * b) % c)`
minv_Lemma10:
  `!x. ((0 < x) /\ (x < 65537)) ==>
  (((int_of_num x) * (piu1 x)) % 65537 = 1) \/
  (((int_of_num x) * (piu2 x)) % 65537 = 1)`
minv_Theorem:
  `!x. ((0 < x) /\ (x < 65537)) ==>
  (((int_of_num x) * (minv x)) % 65537 = 1)`
piu_Lemma1:
  `!x. (0 <= piu1 x) /\ (piu1 x < 65537)`
piu_Lemma2:
  `!x. (0 <= piu2 x) /\ (piu2 x < 65537)`
minv_Corollary1:
  `!x. ((0 < x) /\ (x < 65537)) ==>
  ((0 <= (minv x)) /\ ((minv x) < 65537))`
minv_Corollary2:
  `!x. ((0 < x) /\ (x < 65537)) ==> ~((minv x) = 0)`
minv_Corollary3:
  `!x. ((0 < x) /\ (x < 65537)) ==> ((0 < (minv x)) /\
  ((minv x) < 65537))`
encode_Lemma1:
  `!x. (x < 65536) ==> ((0 < (encode x)) /\ ((encode x) < 65537))`
encode_Lemma2:
  `!x. (x < 65536) ==>
  (((int_of_num (encode x)) * (minv (encode x))) % 65537 = 1)`
encode_Lemma3:
  `!w. ((0 < (encode (w2n w))) /\ ((encode (w2n w)) < 65537))`
decode_Lemma1:
  `!x. ((0 < x) /\ (x < 65537)) ==> ((decode x) < 65536)`
decode_Lemma2:
  `!x. ((0 < x) /\ (x < 65537)) ==> ((encode (decode x)) = x)`
decode_Lemma3:
  `!x. (x < 65536) ==> ((decode (encode x)) = x)`
```

3.2 Working With 16-bit Words

Specification *winv* is defined to get the inverse of the word16 type for a given word16 value. This function first converts the input word16 value into a natural number, encodes it, finds its inverse, and then decodes the inverse and converts it back to word16. *wmul* defines the modular multiplication. It converts and encodes two word16 values to integers, performs the modular multiplication, and converts and decodes the product back into word16. MOD_EQ0 is a specialized theorem from MOD_P in arithmetic theory. It states "$!p\,q.0 < q ==> ((pMODq = 0) = ?k.p = k * q)$". It is used in proving the range of the modular multiplicative product is from 1 to 65536 in the next subsection.

```
val winv_def = Define ‘winv (w:word16) =
  n2w (decode (Num (minv (encode (w2n w)))))‘;
val wmul_def = Define ‘wmul x y =
  n2w (decode (Num (((int_of_num (encode (w2n x))) *
  (int_of_num (encode (w2n y)))) % 65537)))‘;
val _ = set_fixity "wmul"  (Infixr 350);
val MOD_EQ0 =
  (SIMP_RULE arith_ss [] (BETA_RULE (Q.SPEC ‘\x. x=0‘ MOD_P)));
```

Verification The theorems in this part prove the following facts:

1. Num_Lemma1 infers the less relation between two natural numbers from the less relation between the corresponding positive integers. It is an extension to the integer theory, and used in the proof of type conversions.
2. wmul_Lemma1 and wmul_Lemma2 help wmul_Theorem to prove the correctness of word16 multiplicative inverse (*winv*) under the modular multiplication (*wmul*).
3. wmul_ASSOC shows the modular multiplication (*wmul*) is associative. It needs the proof of wmul_Lemma4, which proves the product of the modular multiplication won't equal to zero. Because the multiplication takes the modulus 65537, we can conclude the multiplicative product is from 1 to 65536. This fact helps prove the associative property.
4. wmul_Mul1 proves any word16 value multiplies one equals to itself. This is a property of the modular multiplication we defined, so we need prove it here.

With wmul_Theorem, we have proved the adapted algorithm is correct! However, most applications that use multiplicative inverses also require the multiplication has the associative property and returns the multiplicand when it is multiplied with one. Therefore, we went extra miles to prove our modular multiplication has these two properties.

```
Num_Lemma1:
  ‘!x y. ((0 <= x) /\ (0 <= y) /\ (x < y)) ==> ((Num x) < (Num y))‘
wmul_Lemma1:
  ‘!w. (((int_of_num (encode (w2n w))) *
  (minv (encode (w2n w)))) % 65537 = 1)‘
wmul_Lemma2:
  ‘!x. ((0 < x) /\ (x < 65537)) ==>
  (int_of_num (encode (w2n (n2w (decode (Num x))))) = x)‘
wmul_Theorem:
  ‘!w. w wmul winv(w) = 1w‘
wmul_Lemma4:
  ‘!x:word16 y:word16.
  ~((($& (encode (w2n x)) * $& (encode (w2n y))) % 65537) = 0)‘
wmul_ASSOC:
  ‘!x:word16 y:word16 z:word16. (x wmul y) wmul z = x wmul (y wmul z)‘
wmul_Mul1:
  ‘!w:word16. w wmul 1w = w‘
```

4 Conclusions

We have formulated the specification and proof of Euclid's Algorithm that is used for the calculation of multiplicative inverses. We did this in a gradual manner. First, we specify the general Euclid's Algorithm and verify some of its properties, then we define the specialized Euclid's Algorithm and prove its related properties, and finally we show the functional correctness of the algorithm as it is used in computer computation.

The proof we have provided is an important corner stone in verifying the correctness of IDEA. We also expect it to be helpful in adding multiplicative inverse related evidence in the proof of RSA. Moreover, we hope it can illuminate the verification of other algorithms that use multiplicative inverses.

In the use of HOL-4 system for this work, we feel that theorems in some basic theories are not rich enough. We have to prove quite some theorems that are very general. Sections 2.2, 3.1 and 3.2 all have some extended theorems from standard theories.

References

[1977] R. Rivest, A. Shamir, L. Adleman: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, Vol. 21 (2), 1978, pages 120–126.

[1991] C. Kaufman, R. Perlman, and M. Speciner: Network Security - Private Communication in a Public World. Second Edition, Prentice Hall (ISBN 0-13-040619-2)

[2002] Konrad Slind: A Verification of Rijndael in HOL. Supplementary Proceedings of TPHOLs'02, NASA Conference Proceedings CP-2002-211736

[1976] Witfield Diffie, Martin Hellman: Exhaustive Cryptanalysis of the NBS Data Encryption Standard. IEEE Computer 10(6), June 1977, pp74-84

A Formal Verification of IDEA with HOL

IDEA (International Data Encryption Algorithm) is a secret-key cryptographic algorithm. It was published in 1991 by Xuejia Lai and James L. Massey of ETH Zurich. It's original name is IPES (Improved Proposed Encryption Standard). This algorithm encrypts a 64-bit block of plain text into a 64-bit block of cypher text using a 128-bit key. It has been studied by cryptoanalysts since its publication. So far, no weakness has been identified in publications. In this section, we will first specify IDEA in HOL-4 [Kananaskis 3], and then provide the proof of its correctness. The proof of Euclid's Algorithm given above lays the foundation of this proof.

The state of IDEA is initialized with the input data, and then transformed by primitive operations in several rounds. Although IDEA operates on 64-bit blocks, its primitive operations work with 16-bit quantities, and thus the state is represented with four 16-bit words. The HOL library has a data type word16 that suits our need, so we define the state as a 4-tuple of word16. The algorithm uses a 128-bit quatity as the input key, so we define it as an 8-tuple of word16. The input key is then expanded into 52 16-bit quantities. Each of these quantities are defined as word16 values. We use a list to hold these values, and thus needn't provide the container type. One special characteristic of IDEA is that it distinguishes between even rounds and odd rounds and different rounds use different types of keys. The even rounds use the key type that has four 16-bit values, while the odd rounds uses the type that has only two 16-bit values. As a result, we have the different EvenKey and OddKey types. We also need two container types to hold all even-round keys and all odd-round keys. These types provide convenience to the round definitions such that each round can rotate keys within them and always use the first key inside. EvenKeySched holds 8 EvenKeys, and OddKeySched holds 9 OddKeys.

In DES [1976], each Sbox maps a 6-bit value into a 4-bit value. In IDEA, however, each primitive operations maps two 16-bit values into one 16-bit value. IDEA has three primitive operations. They are bitwise exclusive or (xor), modulo addition, and modulo multiplication. The modulo addition is the addition under the modulus of 2^{16}. It is the same as the addition provided in HOL word16Theory. The exclusive or is also the same as the one in this theory. Thus, we can use the two existent operations directly. On the other hand, the modulo multiplication used in IDEA is different from the one provided in word16Theory. It is under the modulus of $2^{16} + 1$, but the one in word16Theory is under the modulus of 2^{16}. Therefore, we have to define our own modulo multiplication. This is done in the proof of Euclids' Algorithm (wmul_def).

IDEA expands one 128-bit input key into fifty-two 16-bit encryption keys. The expansion is done by chopping off eight 16-bit keys from the input key seven times, and each time starts with a different offset that is a multiple of 25. For the first time, we start from the beginning (bit 0) and chop the input key until the end. For the second time, we offset the starting position by 25 bit, i.e. start the chopping from the bit 25, and wrapping around to the beginning when the end is reached. Next time, we offset another 25 bit from the previous starting position, and so on. Because we only need fifty-two encryption keys, we chop off four 16-bit keys for the last time. Because there is no "chopping" operation in HOL, we use the shifting and bitwise xor operations to simulate it. For example, the new KEY1 is the combination of the last 7 bits of the old KEY2 and the first 9 bits of the old KEY3, due to the effect of rotating the input key to the right for 25 bits from its previous state. To get this result, we left shift the old KEY2 for 9 bits, right shift the old KEY3 for 7 bits, and xor them together. Please note that we use the logical right shift (>>>) instead of the arithmetic right shift (>>) in order to make sure that the vacated high bits are filled with zeroes. After the key expansion, the expanded keys are grouped into the key schedule for the odd rounds and the key schedule for the even rounds. This is done with two operations. The first operation uses the first four keys in every six keys in the list to make odd-round keys. Because there are totally fifty-two keys in the list, four keys will be left after extracting keys in the group of six, these four keys are used to make the last odd-round key. The second operation uses the last two keys in every six keys in the list to make even-round keys. The decryption keys are made by inversing the encryption keys and (or) reversing the order in which these key are used. For the odd rounds, the encryption keys are inverted and then used in the reverse order. The first and fourth keys of every odd-round key are the multiplicative inverses, and the second and third keys are the additive inverse. These definitions are shown in the InverseKey_def and InverseKeys_def below. For the even rounds, the decryption keys are made by simply reversing the order of the encryption keys as shown in ReverseKeys_def.

IDEA has seventeen rounds. Among them, nine rounds are odd rounds and eight are even rounds (The round number starts from one). Even rounds and odd rounds are different because they are designed differently and they use different types of keys. The odd round is relatively simple. There are four 16-bit values in the odd-round key, and there are also four 16-bit values in the input block, so every pair of these 16-bit values are added or multiplied together to produce one 16-bit value in the output block. This is defined by OddRound_def. The even round is more complicated. It uses two mangler functions to generate two intermediate values, Yout and Zout, from the input key and block values. Then, these two intermediate values are xor'ed with the input block values to produce the output block values. In the original protocol specification [1991] there is only one mangler function, which generates Yout and Zout in two steps. We define two steps in the original mangler function as two separate functions to simplify the verification. First, the transformation of the first two 16-bit values in the input block is only affected by the first mangler step, so separating the two steps is good for verifying this transformation. Second, the second mangler step takes the output of the first step as one of its inputs, so separating the two steps modularized operations, i.e. we can prove certain properties of the first step and use these properties in the verification of the second step. The seventeen rounds are executed by calling the Round function with 17 as the round number value (the input parameter n). The Round function then recursively calls itself with decreased round number n, new key from the rotated key schedule, and transformed state. It calls EvenRound or OddRound function based on the round number. The rotation of the key schedule is done by RotateOddKeys and RotateEvenKeys.

One amazing feature of IDEA is that it can perform either encryption or decryption without requiring any change to the primitive operations, rounds or the orders in which they are carried out. In othere words, for the software implementation of IDEA, we can use the same code to perform either operation. Which operation is performed depends on which key is used. If we use the encryption key, the code will encrypt the input block; if we use the decryption key, it will decrypt the input block. In our specification, IdeaFwd is defined to run seventeen rounds without distinguishing encryption from decryption. Then, the encryption is defined by passing the expanded encryption key to IdeaFwd, and the decryption is defined by passing the decryption key to it.

The specification is given below:

```
val _ = type_abbrev("Block", Type`:word16 # word16 # word16 # word16`);
val _ = type_abbrev("InputKey", Type`:word16 # word16 # word16 #
        word16 # word16 # word16 # word16 # word16`);
val _ = type_abbrev("EvenKey", Type`:word16 # word16`);
val _ = type_abbrev("OddKey", Type`:word16 # word16 # word16 # word16`);
val _ = type_abbrev ("EvenKeySched", Type`:EvenKey#EvenKey#EvenKey#
        EvenKey#EvenKey#EvenKey#EvenKey#EvenKey`);
val _ = type_abbrev ("OddKeySched", Type`:OddKey#OddKey#OddKey#OddKey#
        OddKey#OddKey#OddKey#OddKey#OddKey`);
val (MakeEnKeys_def,MakeEnKeys_ind) =
  Defn.tprove (Hol_defn "MakeEnKeys"
    `MakeEnKeys n (K8::K7::K6::K5::K4::K3::K2::K1::rst) =
    let (NK1, NK2, NK3, NK4, NK5, NK6, NK7, NK8) =
      ((K2<<9) # (K3>>>7), (K3<<9) # (K4>>>7),
      ((K4<<9) # (K5>>>7), (K5<<9) # (K6>>>7),
      (K6<<9) # (K7>>>7), (K7<<9) # (K8>>>7),
      (K8<<9) # (K1>>>7), (K1<<9) # (K2>>>7))
    in if n = 0 then
          (NK4::NK3::NK2::NK1::K8::K7::K6::K5::K4::K3::K2::K1::rst)
       else
          MakeEnKeys (n-1) (NK8::NK7::NK6::NK5::NK4::NK3::NK2::NK1
            ::K8::K7::K6::K5::K4::K3::K2::K1::rst)`,
        WF_REL_TAC `measure (FST)`);
val MakeKeys_def = Define
  `MakeKeys ((K1, K2, K3, K4, K5, K6, K7, K8):InputKey) =
  MakeEnKeys 6 [K8;K7;K6;K5;K4;K3;K2;K1]`;
val ListToOddKeys_def =
  Define `(ListToOddKeys [] oddkeys = oddkeys) /\
  (ListToOddKeys ((k1::k2::k3::k4::k5::k6::t): word16 list)
    ((ok1,ok2,ok3,ok4,ok5,ok6,ok7,ok8,ok9): OddKeySched) =
  ListToOddKeys t ((k1,k2,k3,k4),ok1,ok2,ok3,ok4,ok5,ok6,ok7,ok8)) /\
  (ListToOddKeys ((k1::k2::k3::k4::t): word16 list)
    ((ok1,ok2,ok3,ok4,ok5,ok6,ok7,ok8,ok9): OddKeySched) =
  ListToOddKeys t ((k1,k2,k3,k4),ok1,ok2,ok3,ok4,ok5,ok6,ok7,ok8))`;
val ListToEvenKeys_def =
  Define `(ListToEvenKeys [] evenkeys = evenkeys) /\
  (ListToEvenKeys ((k1::k2::k3::k4::k5::k6::t): word16 list)
    ((ek1,ek2,ek3,ek4,ek5,ek6,ek7,ek8): EvenKeySched) =
  ListToEvenKeys t ((k5,k6),ek1,ek2,ek3,ek4,ek5,ek6,ek7))`;
val InverseKey_def = Define `InverseKey (k1,k2,k3,k4) =
    ((winv k1), ~k3, ~k2, (winv k4)) : OddKey`;
val InverseKeys_def =
  Define `InverseKeys (ok1,ok2,ok3,ok4,ok5,ok6,ok7,ok8,ok9) =
  (InverseKey ok9,InverseKey ok8,InverseKey ok7,InverseKey ok6,
  InverseKey ok5,InverseKey ok4,InverseKey ok3,InverseKey ok2,
  InverseKey ok1) : OddKeySched`;
val ReverseKeys_def =
  Define `ReverseKeys (ek1,ek2,ek3,ek4,ek5,ek6,ek7,ek8) =
  (ek8,ek7,ek6,ek5,ek4,ek3,ek2,ek1) : EvenKeySched`;
val OddRound_def = Define
  `OddRound ((Ka, Kb, Kc, Kd):OddKey) ((Xa, Xb, Xc, Xd):Block) =
  (Xa wmul Ka, Xc + Kc, Xb + Kb,Xd wmul Kd ):Block‘;
val Mangler1_def = Define ‘Mangler1 ((Yin:word16), (Zin:word16),
  (Ke:word16), (Kf:word16)) = ((Ke * Yin) + Zin) * Kf‘;
val Mangler2_def = Define ‘Mangler2 ((Yin:word16), (Ke:word16),
  (Yout:word16)) = (Ke * Yin) + Yout‘;
val EvenRound_def = Define
  ‘EvenRound ((Ke, Kf):EvenKey) ((Xa, Xb, Xc, Xd):Block) =
  let Yout = Mangler1 ((Xa # Xb), (Xc # Xd), Ke, Kf) in
  let Zout = Mangler2 ((Xa # Xb), Ke, Yout) in
  (Xa # Yout, Xb # Yout, Xc # Zout, Xd # Zout):Block‘;
val RotateOddKeys_def =
  Define ‘RotateOddKeys (k1,k2,k3,k4,k5,k6,k7,k8,k9) =
  (k2,k3,k4,k5,k6,k7,k8,k9,k1) : OddKeySched‘;
val RotateEvenKeys_def =
  Define ‘RotateEvenKeys (k1,k2,k3,k4,k5,k6,k7,k8) =
  (k2,k3,k4,k5,k6,k7,k8,k1) : EvenKeySched‘;
val (Round_def,Round_ind) =
  Defn.tprove (Hol_defn "Round"
  ‘Round n (oddkeys: OddKeySched) (evenkeys: EvenKeySched)
  (state:Block) = if (n = 0) then state else
    if (EVEN n) then Round (n-1) oddkeys
      (RotateEvenKeys evenkeys) (EvenRound (FST evenkeys) state)
    else Round (n-1) (RotateOddKeys oddkeys) evenkeys
      (OddRound (FST oddkeys) state)‘,
  WF_REL_TAC ‘measure (FST)‘);
val IdeaFwd_def = Define ‘IdeaFwd oddkeys evenkeys =
  Round 17 oddkeys evenkeys‘;
val IDEA_def = Define ‘IDEA key =
  let oddkeys = ListToOddKeys (MakeKeys key) ZeroOddKeys in
  let evenkeys = ListToEvenKeys (MakeKeys key) ZeroEvenKeys in
  (IdeaFwd oddkeys evenkeys, IdeaFwd (InverseKeys oddkeys)
  (ReverseKeys evenkeys))‘;
```

We defined many types that are aggregations of 16-bit values, so we need theorems to dissolve them into word16. The theorem FORALL_ODDKEYSCHED is proved to dissolve the odd key schedule into odd keys, and the theorem FORALL_ODDKEY is proved to further dissove an odd key into four word16 values. We also defined the similar theorems for the even key schedule, even key and block.

For odd rounds, the most important proof is to show that the addition or multiplication with inverses cancells the effect of the previous addition or multiplication with original values. For the addition, this proof is given by OddRound_Lemma1. For the multiplication, it is given by wmul_Theorem, wmul_AS SOC and wmul_Mul1 in the proof of Euclids' Algorithm. As defined in the section 3.2, wmul_Theorem proves that the mulitiplication of one 16-bit value with its inverse equals to one; wmul_ASSOC proves that the modulo muliplication (wmul) has the associative attribute, and thus the 16-bit value can be multiplied with its inverse before it is multiplied with the input value; wmul_Mul1 proves that the input value doesn't change when multiplied with one.

For even rounds, we need work with the mangler functions first. Mangler1_Lemma1 and Mangler2_Lemma1 prove that if two 16-bit values are multiplied with the same mangler function individually and then multiplied together the result equals to multiplying them directly. Mangler1_Lemma2 and Mangler2_Lemma2 show that the effect of multiplying a mangler function can be cancelled by multiplying the same function again. With the assistance of these lemmas, we can prove that the effect of a even round operation can be inversed by simply run the same operation with the same key again. What a magic!

Since we have shown that both even rounds and odd rounds are inversable, it is easy to prove the whole seventeen rounds are inversable as shown in IDEA_LEMMA below. To prove the correctness of the encryption and decryption, we simply apply the definition of the ecryption and decryption to the goal, and then apply the lemma we just proved. In the verification of IDEA we used many techniques learned from the proof of AES [2002].

```
FORALL_ODDKEYSCHED:
  ‘(!x:OddKeySched. Q x) = !k1 k2 k3 k4 k5 k6 k7 k8 k9.
  Q(k1,k2,k3,k4,k5,k6,k7,k8,k9)‘
FORALL_ODDKEY:
  ‘(!x:OddKey. Q x) = !kw1 kw2 kw3 kw4. Q(kw1,kw2,kw3,kw4)‘
OddRound_Lemma1:
  ‘!w1:word16 w2:word16. w1 + w2 + ~w2 = w1‘
OddRound_Inversion:
  ‘!s:Block k:OddKey. OddRound (InverseKey k) (OddRound k s) = s‘
Mangler1_Lemma1:
  ‘!w1 w2 w3 w4 w5 w6. w5 # Mangler1 (w1, w2, w3, w4) #
  (w6 # Mangler1 (w1, w2, w3, w4)) = w5 # w6‘,
Mangler2_Lemma1:
  ‘!w1 w2 w3 w4 w5. w4 # Mangler2 (w1, w2, w3) #
  (w5 # Mangler2 (w1, w2, w3)) = w4 # w5‘
Mangler1_Lemma2:
  ‘!w1 w2 w3 w4 w5. w5 # Mangler1 (w1, w2, w3, w4) #
  Mangler1 (w1, w2, w3, w4) = w5‘
Mangler2_Lemma2:
  ‘!w1 w2 w3 w4. w4 # Mangler2 (w1, w2, w3) #
  Mangler2 (w1, w2, w3) = w4‘
EvenRound_Inversion:
  ‘!s:Block k:EvenKey. EvenRound k (EvenRound k s) = s‘
IDEA_LEMMA:
  ‘!plaintext:Block oddkeys:OddKeySched evenkeys:EvenKeySched.
  IdeaFwd (InverseKeys oddkeys) (ReverseKeys evenkeys)
  (IdeaFwd oddkeys evenkeys plaintext) = plaintext‘
IDEA_CORRECT:
  ‘!key plaintext. ((encrypt,decrypt) = IDEA key) ==>
  (decrypt (encrypt plaintext) = plaintext)‘
```

Liveness Reasoning for Inductive Protocol Verification *

Xingyuan Zhang, Huabing Yang, and Yuanyuan Wang

PLA University of Science and Technology, P.R. China
xyzhang@public1.ptt.js.cn,yanghuabing@gmail.com

Abstract. This paper describes an extension of Paulson's inductive protocol verification approach. With this extension, liveness properties can be verified. The extension requires no change of the system model underlying the original inductive approach. Therefore, all the advantages, which makes Paulson's approach successful in safety reasoning are kept, while liveness reasoning capability is added. The liveness approach itself is reasonably convenient, as shown by its application to a non-trivial benchmark example – the liveness of elevator control system[18]. This work constitutes the first step towards developing inductive protocol verification into a general approach for the verification of concurrent systems.

Keywords: Inductive Protocol Verification, Temporal Reasoning, Isabelle

1 Introduction

Paulson's inductive approach for protocol verification[13] has been used to verify fairly complex security protocols [15,14]. The success gives incentives to extend this approach to a general approach for concurrent system verification. To achieve this goal, a method for the verification of liveness properties is needed. This paper proposes such a method.

The original inductive approach is only used to prove safety properties, i.e. properties about finite execution traces. In this paper, liveness properties are expressed as predicates on infinite execution traces. Infinite traces are represented as functions of the type: $nat \Rightarrow {'a}$, where the $'a$ is the type of events. To make sure what is proved is indeed liveness properties, a shallow embedding of LTL (Linear Temporal Logic)[8] is given. According to Manna and Puneli[8], temporal properties can be classified into three classes: safety properties, response properties and reactivity properties. The proof of safety properties is well solved by the inductive approach. In this paper, proof rules for liveness properties (both response and reactivity) are derived. The proof rules are used to reduce the proof of liveness properties to the proof of safety properties, so that

* This research was funded by National Natural Science Foundation of China, under grant 60373068 'Machine-assisted correctness proof of complex programs'

the original inductive approach's advantages in the proof of safety properties can be fully exploited. The application of this approach to the liveness verification of an elevator control system is presented in a companion draft paper[18]. Our experiences convinced us of the practicability of this approach.

The proof rules are derived based on a new notion of fairness, *parametric fairness*, which is a adaption of the α-fairness [17,2,6] to the setting of HOL. *Parametric fairness* is properly stronger than standard fairness notions such as *weak fairness* and *strong fairness*. We will explain why the use of *parametric fairness* can deliver more liveness results through simpler proofs.

There have been a lot of works on the embedding of temporal logics and I/O automata in theorem proving systems, [4,11,10,12,5,1] are just a few of them. It is not obvious how these works can work together with Paulson's inductive approach. This paper seems to be the first effort towards this direction.

The paper is organized as the following: section 2 presents the system model used in inductive approach, as well as a set of syntactic sugars to make later developments prettier; section 3 gives a shallow embedding of LTL; section 4 introduces and justifies parametric fairness; section 5 explains the liveness proof rules; section 6 discusses related works; section 7 concludes.

2 Concurrent systems

In inductive approach, system states are identified with finite executions , which are represented as lists of events. Events in a state list are arranged in reverse order of happening. The decision of which event to happen next is made according to the current system state. Definitions based on this view is given in Figure 1.

Since the set of events depends on specific systems, the type of events is defined as a polymorphic type $'a$ and the type of system states is defined as $'a\ list$. System states are written as τ. In this paper, *finite trace*, *event trace*, *finite execution* and *system state* are used interchangeably. *Infinite executions* (or *infinite traces*) are written as σ, which is of type $nat \Rightarrow\ 'a$. Therefore, the event happened at step i is $\sigma\ i$ and it is usually abbreviated as σ_i. The first i events of an infinite execution σ can be packed into a list in reverse order to form a finite execution and such a packing is written as $[\![\sigma]\!]_i$.

The type of concurrent systems is $('a\ list \times\ 'a)\ set$ and concurrent systems are written as *cs*. The expression $(\tau,\ e) \in cs$ means that the event e is legitimate to happen under state τ, according to *cs*. The notation $(\tau,\ e) \in cs$ is abbreviated as $\tau\ [cs\!>\ e$. The set of valid finite executions of a concurrent system *cs* is written as *vt cs*, which is inductively defined. The expression $\tau \in vt\ cs$ means that the finite execution τ is a valid finite execution of *cs*. The expression $\tau \in vt\ cs$ is usually abbreviated as $cs \vdash \tau$. The operator $\vdash$ is overloaded, so that the fact that σ is a valid infinite execution of *cs* can be written as $cs \vdash \sigma$. It can be seen

```
constdefs i-th :: (nat ⇒ 'a) ⇒ nat ⇒ 'a (-_ [64, 64] 1000)
  σ_i ≡ σ i

consts prefix :: (nat ⇒ 'a) ⇒ nat ⇒ 'a list (⟦-⟧- [64, 64] 1000)
primrec ⟦σ⟧_0 = []
        ⟦σ⟧_(Suc i) = σ_i # ⟦σ⟧_i

constdefs may-happen ::
  'a list ⇒ ('a list × 'a) set ⇒ 'a ⇒ bool (- [-> - [64, 64, 64] 50)
  τ [cs> e ≡ (τ, e) ∈ cs

consts vt :: ('a list × 'a) set ⇒ 'a list set
inductive vt cs
  intros
  vt-nil [intro] : [] ∈ vt cs
  vt-cons [intro] : ⟦τ ∈ vt cs; τ [cs> e⟧ ⟹ (e # τ) ∈ vt cs

consts derivable :: 'a ⇒ 'b ⇒ bool (- ⊢ - [64, 64] 50)

defs (overloaded)
  fnt-valid-def:  cs ⊢ τ ≡ τ ∈ vt cs
  inf-valid-def:  cs ⊢ σ ≡ ∀i. ⟦σ⟧_i [cs> σ_i
```

Fig. 1. The definitions of concurrent system

from lemma *ve-prefix*: $(cs \vdash \sigma) = (\forall\, i.\ cs \vdash [\![\sigma]\!]_i)$ that an infinite execution σ is valid under cs iff. all of its prefixes are valid.

3 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 Figure 2 to make sure that the proof rules derived in this work are indeed for liveness properties.

LTL formulae are written as φ, ψ, κ etc. The type of LTL formulae is defined as *'a tlf*. The expression $(\sigma,\ i) \models \varphi$ means that LTL formula φ is valid at moment i of the infinite execution σ. The operator $\models$ is overloaded, so that $\sigma \models \varphi$ can be defined as the abbreviation of $(\sigma,\ 0) \models \varphi$.

The *always* operator $\Box$, *eventual* operator $\Diamond$, *next* operator $\odot$, *until* operator $\rhd$ are defined literally.

An operator $\langle\text{-}\rangle$ is defined to lift a predicate on finite executions up to a LTL formula. The temporal operator $\hookrightarrow$ is the lift of logical implication $\longrightarrow$ up to LTL level. For an event e, the term $(\!|e|\!)$ is a predicate on finite executions stating that the last happened event is e. Therefore, the expression $\langle(\!|e|\!)\rangle$ is an LTL formula saying that event e happens at the current moment.

```
types ′a tlf = (nat ⇒ ′a) ⇒ nat ⇒ bool

consts valid-under :: ′a ⇒ ′b ⇒ bool (- ⊨ -  [64, 64] 50)
defs (overloaded)  pr ⊨ φ ≡ let (σ, i) = pr in φ σ i
defs (overloaded)  σ ⊨ φ ≡ (σ::nat ⇒ ′a, (0::nat)) ⊨ φ

  □φ ≡ λ σ i. ∀ j. i ≤ j ⟶ (σ, j) ⊨ φ
  ◇φ ≡ λ σ i. ∃ j. i ≤ j ∧ (σ, j) ⊨ φ
  ⊙φ ≡ λ σ i. (σ, (Suc i)) ⊨ φ
  φ ▷ ψ ≡ λ σ i. ∃j. i ≤ j ∧ (σ, j) ⊨ ψ ∧ (∀k. i ≤ k ∧ k < j ⟶ (σ, k) ⊨ φ)

constdefs lift-pred :: (′a list ⇒ bool) ⇒ ′a tlf   (⟨-⟩ [65] 65)
  ⟨P⟩ ≡ λ σ i. P ⟦σ⟧ᵢ

constdefs lift-imply :: ′a tlf ⇒ ′a tlf ⇒ ′a tlf (-↪- [65, 65] 65)
  φ ↪ ψ ≡ λ σ i. φ σ i ⟶ ψ σ i

constdefs last-is :: ′a ⇒ ′a list ⇒ bool
  last-is e τ ≡ (case τ of Nil ⇒ False | (e1#t) ⇒ e1 = e)

syntax -is-last :: ′a ⇒ (′a list ⇒ bool) (⦇-⦈ [64] 1000)
translations ⦇e⦈ ⇌ last-is e
```

Fig. 2. A shallow embedding of LTL

4 The notion of parametric fairness

In this section, *parametric fairness* is introduced as a new notion of fairness which suits the setting of HOL. *Parametric fairness PF* is defined in Figure 3 at the end of a spectrum of fairness notions, among which are the standard ones such as *weak fairness WF* and *strong fairness SF*. The spectrum is arranged to manifest structural resemblances and differences, so that *parametric fairness PF* can be understood as a natural development from standard fairness notions, with the help of some motivating examples.

Fairness notions are expressed as *?F cs* σ, where *cs* is a concurrent system and σ is an infinite execution. Each *?F* is obtained from a corresponding pre-version $?F_\alpha$ with extra parameters hidden by universal quantification. For example, *WF* is obtained from WF_α by quantifying on *e*, *SF* from SF_α by quantifying on *e*, *USF* from USF_α by quantifying on *P* and *e*, etc. Among the $?F_\alpha$s, EF_α has the most general form:

$$\sigma \models \Box\Diamond(\lambda\ \sigma\ i.\ P\ [\![\sigma]\!]_i \wedge [\![\sigma]\!]_i\ [cs{>}\ E\ [\![\sigma]\!]_i) \longrightarrow \sigma \models \Box\Diamond(\lambda\ \sigma\ i.\ P\ [\![\sigma]\!]_i \wedge \sigma_i = E\ [\![\sigma]\!]_i)$$

where $[\![\sigma]\!]_i$ is the current system state. In liveness proofs, progress is made by the happening of helpful events under corresponding help states. The *P* in EF_α is used to specify helpful states, and *E* used to specify helpful events. The meaning of WF_α is that if helpful *E*-events are eligible to happen under helpful *P*-states infinitely often, then *E*-events should happen infinitely often under *P*-states.

Other $?F_\alpha$s can be seen as a specialization of EF_α. For example, USF_α is obtained from EF_α by replacing *E* with $\lambda\ \tau.\ e$, where *e* is the parameter. SF_α is obtained by replacing *P* with $\lambda\ \tau.\ True$, *E* with $\lambda\ \tau.\ e$. Therefore, SF_α has no specific requirement on helpful states. The consequence of this is that helpful

```
constdefs WFα :: ('a list × 'a) set ⇒ 'a ⇒ (nat ⇒ 'a) ⇒ bool
 WFα cs e σ ≡ σ ⊨ □(λ σ i. ⟦σ⟧ᵢ [cs> e) ⟶ σ ⊨ □◇(λ σ i. σᵢ = e)

 WF cs σ ≡ ∀ e. WFα cs e σ

 SFα cs e σ ≡ σ ⊨ □◇(λ σ i. ⟦σ⟧ᵢ [cs> e) ⟶ σ ⊨ □◇(λ σ i. σᵢ = e)

 SF cs σ ≡ ∀ e. SFα cs e σ

 USFα :: ('a list × 'a) set ⇒ ('a list ⇒ bool) ⇒ 'a ⇒ (nat ⇒ 'a) ⇒ bool
 USFα cs P e σ ≡
   σ ⊨ □◇(λ σ i. P ⟦σ⟧ᵢ ∧ ⟦σ⟧ᵢ [cs> e) ⟶ σ ⊨ □◇(λ σ i. P ⟦σ⟧ᵢ ∧ σᵢ = e)

 USF cs σ ≡ ∀ P e. USFα cs P e σ

 EFα :: ('a list × 'a) set ⇒ ('a list ⇒ bool) ⇒ ('a list ⇒ 'a) ⇒ (nat ⇒ 'a) ⇒ bool
 EFα cs P E σ ≡
   σ ⊨ □◇(λ σ i. P ⟦σ⟧ᵢ ∧ ⟦σ⟧ᵢ [cs> E ⟦σ⟧ᵢ) ⟶ σ ⊨ □◇(λ σ i. P ⟦σ⟧ᵢ ∧ σᵢ = E ⟦σ⟧ᵢ)

 EF cs σ ≡ ∀ P E. EFα cs P E σ

types 'a pe = ('a list ⇒ bool) × ('a list ⇒ 'a)

constdefs PF :: ('a list × 'a) set ⇒ 'a pe list ⇒ (nat ⇒ 'a) ⇒ bool
 PF cs pel σ ≡ list-all (λ (P, E). EFα  cs P E σ) pel
```

Fig. 3. Different notions of fairness

events do not have to happen under helpful state, even if they are enabled infinitely often. WF_α is obtained from SF_α by replacing the first $\Box\Diamond$ with $\Box$. Based on such an observation, a strength hierarchy for the fairness notions in Figure 3 is established by the following lemmas.

lemma *ef-usf*: *EF cs* $\sigma \Longrightarrow$ *USF cs* σ
lemma *usf-sf*: *USF cs* $\sigma \Longrightarrow$ *SF cs* σ
lemma *sf-wf*: *SF cs* $\sigma \Longrightarrow$ *WF cs* σ
lemma *ef-pf*: *EF cs* $\sigma \Longrightarrow$ *PF cs pel* σ

We propose using fairness notion stronger than the standard *WF* and *SF*, because both standard notions fail to specify that helpful events must happen under helpful states. Liveness reasoning under *WF* or *SF* fairness assumptions rely on the underlying concurrent system to ensure that helpful events can *only* happen under the corresponding helpful states. However, this is not always the case. The following two examples show that stronger fairness notion is needed for the desired liveness properties to be derived. The state transition diagram and formal definition of the first example system cs_1 are shown in Figure 4 and Figure 5.

In Figure 5, transitions $e_0, \ldots, e_4$ are treated as constructors of type *Evt.* Concurrent system cs_1 itself has type (*Evt list* × *Evt*) *set* and is defined inductively. In this paper, we identify state number with function value. A state is called 'state s', if the value of function F on that state equals s.

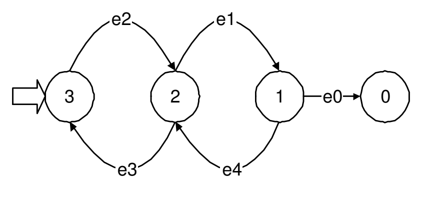


Fig. 4. The diagram of cs_1

```
datatype Evt = e₀ | e₁ | e₂ | e₃ | e₄

consts F :: Evt list ⇒ nat
recdef F measure size
  F [] = 3
  F (e₀ # τ) = (if (F τ = 1) then 0 else F τ)
  F (e₁ # τ) = (if (F τ = 2) then 1 else F τ)
  F (e₂ # τ) = (if (F τ = 3) then 2 else F τ)
  F (e₃ # τ) = (if (F τ = 2) then 3 else F τ)
  F (e₄ # τ) = (if (F τ = 1) then 2 else F τ)

consts cs₁ :: (Evt list × Evt) set
inductive cs₁ intros
  r₀ : F τ = 1 ⟹ (τ, e₀) ∈ cs₁
  r₁ : F τ = 2 ⟹ (τ, e₁) ∈ cs₁
  r₂ : F τ = 3 ⟹ (τ, e₂) ∈ cs₁
  r₃ : F τ = 2 ⟹ (τ, e₃) ∈ cs₁
  r₄ : F τ = 1 ⟹ (τ, e₄) ∈ cs₁
```

Fig. 5. The definition of cs_1

Suppose we want to prove the liveness property that once the system gets into state *2*, it will eventually get into system *0*. The property is formalized as:

$$[\![cs_1 \vdash \sigma;\ ?F\ cs\ \sigma]\!] \Longrightarrow \sigma \models \Box(\langle \lambda\ \tau.\ F\ \tau = 2\rangle \hookrightarrow \Diamond\ \langle \lambda\ \tau.\ F\ \tau = 0\rangle) \qquad (1)$$

where, the *?F* is some fairness notion. The *?F* can not be *WF*. Consider the execution $(e_2.\ e_1.\ e_4.\ e_3)^\omega$ as a counter example. The execution satisfies *WF* cs_1, but the conclusion of (1) does not hold on this execution, because $F\ \tau = 2$ happens for infinitely many times while $F\ \tau = 0$ never happens.

The *?F* should at least be *SF* for (1) to be true. The execution $(e_2.\ e_1.\ e_4.\ e_3)^\omega$ is no longer a counter example, because it does not satisfy *SF* cs_1. Consider the instance $SF_\alpha\ cs_1\ e_0$ of *SF* cs_1, event e_0 is enabled for infinitely many times in execution $(e_2.\ e_1.\ e_4.\ e_3)^\omega$, but never happens. The validity of statement (1) can be established by the following argument:

Argument 1 *Assume the conclusion of (1),* $\sigma \models \Box(\langle \lambda\ \tau.\ F\ \tau = 2\rangle \hookrightarrow \Diamond\ \langle \lambda\ \tau.\ F\ \tau = 0\rangle)$*, does not hold, we are going to derive absurdity. If this is the case, the execution* σ *must get into states 1, 2 and 3 for infinitely many times after reaching state 2 and never enters state 0. Therefore, there must be one state s among 1, 2 and 3, which is entered infinitely. Suppose this s is 3, then event* e_2 *is enabled for infinitely many times, and according to the premise instance* $SF_\alpha\ cs_1\ e_2\ \sigma$ *(an instance of the premise SF* $cs_1\ \sigma$ *in (1)), event* e_2 *must happen for infinitely many times. Since the happening of* e_2 *can only lead from state 3 to state state 2, it can be derived that state 2 is entered infinitely often. In a similar way, it can be derived that state 1 is entered infinitely often, because of the happening of event* e_1*. And finally, it can be derived that state 0 is entered infinitely often because of the happenings of event* e_o*. Absurdity is derived. The case of* $s = 2$ *and* $s = 1$ *can be treated similarly.*

It is obvious from this argument that the fairness notion *SF* is instantiated at helpful events, to make them happen infinitely often. However, because *SF* does not insist that the happenings of helpful event are under help states, in Argument 1, we have to resort to the definition of cs_1 to show that helpful events can *only* happen under their corresponding helpful states. For some concurrent systems, such kind of arguments are not possible. As shown by the example system cs_2 in Figure 6, where the helpful event e_0 can happen under both state *1* and state *2*, with only state *1* being the helpful state.

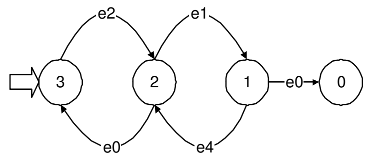


Fig. 6. The diagram of cs_2

```
consts F₂ :: Evt list ⇒ nat
recdef F₂ measure size
  F₂ [] = 3
  F₂ (e₀ # τ) = (if (F₂ τ = 1) then 0 else
                 if (F₂ τ = 2) then 3 else F₂ τ)
  F₂ (e₁ # τ) = (if (F₂ τ = 2) then 1 else F₂ τ)
  F₂ (e₂ # τ) = (if (F₂ τ = 3) then 2 else F₂ τ)
  F₂ (e₄ # τ) = (if (F₂ τ = 1) then 2 else F₂ τ)

consts cs₂ :: (Evt list × Evt) set
inductive cs₂ intros
  r₀ : F₂ τ = 1 ⟹ (τ, e₀) ∈ cs₂
  r₁ : F₂ τ = 2 ⟹ (τ, e₁) ∈ cs₂
  r₂ : F₂ τ = 3 ⟹ (τ, e₂) ∈ cs₂
  r₃ : F₂ τ = 2 ⟹ (τ, e₀) ∈ cs₂
  r₄ : F₂ τ = 1 ⟹ (τ, e₄) ∈ cs₂
```

Fig. 7. The definition of cs_2

The formal definition of cs_2 is given in Figure 7. The counterpart of statement (1) in the case of cs_2 is:

$$[\![cs_2 \vdash \sigma;\ ?F\ cs_2\ \sigma]\!] \Longrightarrow \sigma \models \Box(\langle \lambda\ \tau.\ F_2\ \tau = 2 \rangle \hookrightarrow \Diamond\ \langle \lambda\ \tau.\ F_2\ \tau = 0 \rangle) \tag{2}$$

The effort to repeat Argument 1 based on *SF* assumption failed. Consider the counter example $(e_2.\ e_1.\ e_4.\ e_0)^{\omega}$, which satisfies *SF* cs_2. However, execution $(e_2.\ e_1.\ e_4.\ e_0)^{\omega}$ never enters state *0*, therefore, the conclusion of (2) does not hold. The deficiency of *SF* allows $(e_2.\ e_1.\ e_4.\ e_0)^{\omega}$ to avoid the happening of the helpful event e_0 under the corresponding helpful state *1*. The fairness notion *USF* is proposed to solve this problem. The definition of *USF* is a literal translation of the α*-fairness*, or *extreme fairness*, as defined in [17,2,16][1]. The validity argument for (2) under *USF* assumption is as the following:

[1] In [17,2,16], α*-fairness* and *extreme fairness* are different. This difference disappear in this paper because in the definition of concurrent system, we identify current system state with its history.

Argument 2 *The structure of this argument is quite similar to Argument 1. Suppose* $s = 3$*, so the helpful event* e_2 *is enabled infinitely often under helpful state 3. From this and the premise instance* $USF_\alpha\ cs\ (\lambda\tau.\ F_2\ \tau = 3)\ e_2$*, it can be derived that helpful event* e_2 *must happen infinitely often under helpful state 3. Therefore state 2 is entered infinitely often. From this and the premise instance* $USF_\alpha\ cs\ (\lambda\tau.\ F_2\ \tau = 2)\ e_1$*, it can be derived that state 1 is entered infinitely often. From this and the premise instance* $USF_\alpha\ cs\ (\lambda\tau.\ F_2\ \tau = 1)\ e_0$*, it can be derived that state 0 is entered infinitely often. Absurdity is derived. The case of* $s = 2$ *and* $s = 1$ *can be treated similarly.*

In addition to more derivable liveness properties, another benefit of using stronger fairness notions is that they usually produces simpler liveness proofs. For example, statement (1) can be proved under either of the assumptions *SF* and *USF*. However, the proof under *USF* assumption is simpler, because it is not necessary to derive that helpful event e_o *only* happens under helpful state state *1*, e_1 *only* under state *2*, etc. For complex systems, this could be a great overhead.

Although *USF* is already strong enough to prove statement (2), we prefer to use the slightly stronger fairness notions *EF*, because *EF* provides more flexibility in liveness proof. The use of fairness constraints in liveness arguments is to instance them at helpful events and helpful states, as shown in Argument 1 and Argument 2. The function E in EF_α has the same usage as the e in SF_α and USF_α, it represents a strategy to decide the helpful event based on the current system state $[\![\sigma]\!]_i$. The use of E instead of e is more flexible, because it deals with helpful events collectively.

However, the universal quantification on P and E in the definition of *EF* is too harsh. Since the P and E in *EF*'s definition are higher-order, therefore, the universal quantification over them makes it possible for *EF* to be instantiated to *any* higher-order constructions. The proof in Figure 8 shows that most infinite execution σ do not satisfy *EF*, if the P and E in *EF* are instantiated to the *?P* and *?E*, which are constructed from σ itself.

Parametric fairness *PF* is proposed to solve the problem of *EF*. Instead of quantifying the P and E in EF_α over uncountable domains, *PF* quantifies P and E over *pel*, an explicitly given list of (P, E)-pairs, each of which specifies a collection of helpful events together its corresponding helpful states. This is not really a restriction in practice, because the use of *EF* in liveness proof always is to instantiate the P and E in EF_α to specific helpful events and states, the definition of *PF* is merely to make such instantiations explicit at the level of fairness constraint.

The definition of *PF* indeed solves the problem of *EF*. Although one can still put the (*?P*, *?E*) constructed from σ into *pel* to exclude σ from the set of *PF*-fair executions, he can do this only for finitely many infinite executions, because the length of *pel* is finite. Since there are uncountably many infinite executions, the ones excluded by any specific *PF*-constraints are at most countably many, most infinite executions are still *PF*-fair.

One burden of using *PF* is that we have to put in advance the helpful (P, E)-pairs likely to be used in liveness proof into *pel*, so that the EF_α constraints on these (P, E)-pairs can be extracted from the *PF* premises when needed.

```
lemma infeasiblity-of-EF:
  — $\sigma_i$ is execution $\sigma$'s choice of the event to happen at step $i$. The following
    assumption says that along the way $\sigma$ executes, it is infinitely often the case
    that there are more than one event eligible to happen next. Such kind of
    executions are very common, as in the cases of $cs_1$ and $cs_2$ above.
  assumes aev: $\sigma \models \Box\Diamond(\lambda\ \sigma\ \mathrm{i}.\ \{\mathrm{e}.\ \llbracket\sigma\rrbracket_i\ [\mathrm{cs}\!>\ \mathrm{e} \wedge \mathrm{e} \neq \sigma_i\} \neq \{\})$
  — We are going to show that $\sigma$ does not satisfy $EF$.
  shows $\neg$ EF cs $\sigma$
proof
  assume ef: EF cs $\sigma$ — Assume $\sigma$ satisfies $EF$.
  show False — We are going to show absurdity.
  proof −
    — In this proof, the following $?S$, $?P$ and $?E$ are intended to be applied to $\llbracket\sigma\rrbracket_i$.
    let $?\mathrm{S} = \lambda\ \tau.\ \{\mathrm{e}.\ \llbracket\sigma\rrbracket_{|\tau|}\ [\mathrm{cs}\!>\ \mathrm{e} \wedge \mathrm{e} \neq \sigma_{|\tau|}\}$
      — Since $|\tau| \mapsto |\llbracket\sigma\rrbracket_i| \mapsto i$, $?S\ \llbracket\sigma\rrbracket_i$ becomes $\{e.\ \llbracket\sigma\rrbracket_i\ [cs\!>\ e \wedge e \neq \sigma_i\}$,
        which is the set of events eligible to happen at step $i$, not including $\sigma_i$.
    let $?\mathrm{P} = \lambda\ \tau.\ ?\mathrm{S}\ \tau \neq \{\}$ — $?P\ \llbracket\sigma\rrbracket_i$ requires that $?S\ \llbracket\sigma\rrbracket_i$ is not empty.
    let $?\mathrm{E} = \lambda\ \tau.\ (\varepsilon\ \mathrm{e}.\ \mathrm{e} \in ?\mathrm{S}\ \tau)$ — $?E\ \llbracket\sigma\rrbracket_i$ chooses one element from $?S\ \llbracket\sigma\rrbracket_i$.
    from ef have h: — Expand $ef$ into rule format, and bind the result to $h$.
      $\bigwedge$ P E. $\sigma \models \Box\Diamond(\lambda\ \sigma\ \mathrm{i}.\ \mathrm{P}\ \llbracket\sigma\rrbracket_i \wedge\ \llbracket\sigma\rrbracket_i\ [\mathrm{cs}\!>\ \mathrm{E}\ \llbracket\sigma\rrbracket_i) \Longrightarrow$
              $\sigma \models \Box\Diamond(\lambda\ \sigma\ \mathrm{i}.\ \mathrm{P}\ \llbracket\sigma\rrbracket_i \wedge \sigma_i = \mathrm{E}\ \llbracket\sigma\rrbracket_i)$ by (simp add:EF-def EF$_\alpha$-def)
    — An instance of $h$'s premise, with $P$ instantiated to $?P$, $E$ instantiated to $?E$,
      can be proved as follows:
    have $\sigma \models \Box\Diamond(\lambda\ \sigma\ \mathrm{i}.\ ?\mathrm{P}\ \llbracket\sigma\rrbracket_i \wedge\ \llbracket\sigma\rrbracket_i\ [\mathrm{cs}\!>\ ?\mathrm{E}\ \llbracket\sigma\rrbracket_i)$
      — This is drived from assumption $aev$ by resorting to the meaning of $?P$ and
        $?E$.
    — By applying $h$ to this instance, it can be derived that:
    from h [OF this] have $(\sigma \models \Box\Diamond(\lambda\ \sigma\ \mathrm{i}.\ ?\mathrm{P}\ \llbracket\sigma\rrbracket_i \wedge \sigma_i = ?\mathrm{E}\ \llbracket\sigma\rrbracket_i))$ by simp
    — However, we can derive the opposite of this:
    moreover have $\neg\ (\sigma \models \Box\Diamond(\lambda\ \sigma\ \mathrm{i}.\ ?\mathrm{P}\ \llbracket\sigma\rrbracket_i \wedge \sigma_i = ?\mathrm{E}\ \llbracket\sigma\rrbracket_i))$
      — It is proved by resorting to the very meaning of $?P$ and $?E$. Notice that
        $?E$ is designed deliberately to avoid returing $\sigma_i$, when applied to $\llbracket\sigma\rrbracket_i$.
    ultimately show ?thesis by simp — Absurdity is finally derived.
  qed
qed
```

Fig. 8. The infeasibility of *EF*

Another burden of using *PF* is that each liveness proof may need a different *pel* list. Suppose a list of liveness properties $LP_i\ \sigma\ (i \in \{1..n\})$ have been proved under their corresponding *PF* assumptions $PF\ cs\ pel_i\ \sigma\ (i \in \{1..n\})$, obtaining the following list of lemmas: $PF\ cs\ pel_i\ \sigma \Longrightarrow LP_i\ \sigma\ (i \in \{1..n\})$. We have to combine the results to obtain the conjunction of $LP_i\ \sigma\ (i \in \{1..n\})$ in the form:

$$PF\ cs\ (@_{i \in \{1..n\}}\ pel_i)\ \sigma \Longrightarrow \wedge_{i\ \in\ \{1..n\}}\ LP_i\ \sigma$$

through the use of the following lemma:

$$\textit{PF-app-from} : PF\ cs\ l1\ \ \sigma \wedge PF\ cs\ l2\ \sigma \Longrightarrow PF\ cs\ (l1\ \ @\ \ l2)\ \sigma$$

In [2], Baier defined a class of fairness notions which subsumes standard fairness notions *WF* and *SF*. Baier uses *probabilistic transition system* as the underlying execution model of concurrent systems, where the decision of which event to happen next is made by coin tossing. It is shown in Theorem 1 of [2] that the probability of an execution belonging to Baier's general fairness class is *1*. Baier's result means that properties derived under this fairness class are true in probabilistic sense. The *PF* also falls into Baier's fairness class, because the proof of Theorem 1 in [2] only relies on the fact that the set of liveness labels *L* is countable, and the *pel* in *PF* can be regarded as such a *L*. However, a formal treatment of this argument requires the formalization of measurable set and probability theory in Isabelle, which is certainly beyond the scope of this paper. If we believe that the execution of concurrent system is controlled by coin tossing, then *PF* is a better choice than *WF* and *SF*, because it renders more results and simpler proofs.

5 Liveness rules

In this section, proof rules for both response and reactivity properties are derived. According to Manna[8], response properties, written in our embedding of LTL, are of the form $\sigma \models \Box(\langle P\rangle \hookrightarrow \Diamond\langle Q\rangle)$, where $\langle P\rangle$ and $\langle Q\rangle$ are *past formulae* (in [8]'s term) obtained by lifting predicates on finite traces. The conclusions of statements (1) and (2) are of this form. The form of reactivity properties are of the form $\sigma \models (\Box\Diamond\langle P\rangle) \hookrightarrow (\Box\Diamond\langle Q\rangle)$, the informal meaning of which is: *if execution σ gets into $\langle P\rangle$-states infinitely often, σ will get into $\langle Q\rangle$-states infinitely often as well.*

The proof rule for response property is the theorem *resp-rule*:

$$[\![RESP\ cs\ F\ E\ N\ P\ Q;\ cs \vdash \sigma;\ PF\ cs\ \{\!|F,\ E,\ N|\!\}\ \sigma]\!] \Longrightarrow \sigma \models \Box\langle P\rangle \hookrightarrow \Diamond\langle Q\rangle$$

and the proof rule for reactivity property is the theorem *react-rule*:

$$[\![REACT\ cs\ F\ E\ N\ P\ Q;\ cs \vdash \sigma;\ PF\ cs\ \{\!|F,\ E,\ N|\!\}\ \sigma]\!] \Longrightarrow \sigma \models \Box\Diamond\langle P\rangle \hookrightarrow \Box\Diamond\langle Q\rangle$$

The symbols used in these two theorems are given in Figure 9.

Let's explain the *resp-rule* first. The proof of *resp-rule* is essentially a generalization of Argument 2 to cope with concurrent systems with more liberally shaped

```
  ⦃F, E, 0⦄  = []
  ⦃F, E, (Suc n)⦄ = (λ τ. F τ = Suc n, E) # ⦃F, E, n⦄

syntax -drop :: 'a list ⇒ nat ⇒ 'a list (⌈-⌉_- [64, 64] 1000)
translations ⌈l⌉_n ⇌ drop n l

constdefs exp-nq ::
  ('a list ⇒ bool) ⇒ ('a list ⇒ bool) ⇒ ('a list ⇒ bool) (⌈-⟼¬-∗⌉ [65, 65] 1000)
  ⌈P ⟼¬Q ∗⌉ ≡ λ τ. (∃i ≤ |τ|. P ⌈τ⌉_i ∧ (∀k. 0 < k ∧ k ≤ i ⟶ ¬ Q ⌈τ⌉_k))

locale RESP =
  fixes cs :: ('a list × 'a) set
  and F :: 'a list ⇒ nat
  and E :: 'a list ⇒ 'a
  and N :: nat
  and P :: 'a list ⇒ bool
  and Q :: 'a list ⇒ bool
  assumes mid: ⟦cs ⊢ τ; ⌈P ⟼¬Q ∗⌉ τ; ¬ Q  τ⟧ ⟹  0 < F τ ∧ F τ < N
  and fd: ⟦cs ⊢ τ; 0 < F τ⟧ ⟹ τ [cs> E τ ∧ F (E τ # τ) < F τ

locale REACT =
  fixes cs :: ('a list × 'a) set
  and F :: 'a list ⇒ nat
  and E :: 'a list ⇒ 'a
  and N :: nat
  and P :: 'a list ⇒ bool
  and Q :: 'a list ⇒ bool
  assumes init: ⟦cs ⊢ τ; P τ⟧ ⟹ F τ < N
  assumes mid: ⟦cs ⊢ τ; F τ < N; ¬ Q τ⟧ ⟹ τ [cs> E τ ∧ F (E τ # τ) < F τ
```

Fig. 9. Primises for response and reactivity rules

state transition diagrams. In Argument 2, it is first shown that there is a state s (the state 3), which occurs infinitely often. There is an event path leading from state s into the desired state (the path $[e_0,\ e_1,\ e_2]$), each state along the path (state 3, 2 and 1) is a helpful state, with the outbound event along the path as the corresponding helpful event (event e_2 for state 3, event e_1 for state 2, etc.). Since the fairness assumption ($USF_\alpha\ cs\ (\lambda\ \tau.\ F_2\ \tau = 3)\ e_2\ \sigma$, $USF_\alpha\ cs\ (\lambda\ \tau.\ F_2\ \tau = 2)\ e_1\ \sigma$, $USF_\alpha\ cs\ (\lambda\ \tau.\ F_2\ \tau = 1)\ e_0\ \sigma$) convers all helpful states and helpful events along the path, infinite occurrence of state s will naturally leads to infinite occurrences of the desired state (state 0).

In *resp-rule*, the requirements on the shape of state transition diagrams is expressed as the locale *RESP*. The *mid* assumption of *RESP* ensures the existence of the state s and the *fd* assumption ensures the existence of a path leading from s to the desired Q-states. The $PF\ cs\ \{\!|F,\ E,\ N|\!\}\ \sigma$ premise of *resp-rule* generates premise instances which covers all helpful states and helpful events along the path from state s to the desired Q-states.

The reason that assumption *mid* ensures the existence of state s is explained as follows: The set of all intermediate states after reaching P and before reaching Q are characterized by the premises $\lceil P \longmapsto \neg Q\ * \rceil\ \tau$ and $\neg\ Q\ \ \tau$ of *RESP*'s assumption *mid*. The meaning of $\lceil P \longmapsto \neg Q\ * \rceil\ \tau$ is that a P-state has been reached earlier in τ and from that P-moment till before the head of τ, a Q-state has never been reached (notice that τ is a list of events arranged in reverse order of happening). The conclusion of *mid* requires that the function F has a upper bound N and larger than zero when applied to intermediate states. If an execution σ violates *resp-rule*, it will remain in intermediate states forever, since assumption *mid* bounds the value of F, there must be a state s ($0 < s \wedge s < N$) which happens infinitely often.

The reason that assumption *fd* ensures the existence of a path leading from s to the desired Q-states is as follows: Since $0 < s \wedge s < N$, from this, it can be proved that the premise $0 < F\ \tau$ of *fd* holds. The application of *fd* derives that the happening of E-event at state s will lead the system into a state with lower F-value. If this lower F-value is non-zero, then *fd* rule can be applied again, until F-value reaches 0, when the execution σ finally reaches the desired Q-states.

The reason that $PF\ cs\ \{\!|F,\ E,\ N|\!\}\ \sigma$ generates premise instances which covers the path from state s to the desired Q-states can be understood by considering the case when *resp-rule* is used to prove statement (2). In this case, the P is instantiated to $(\lambda\ \tau.\ F_2 = 2)$ and Q to $(\lambda\ \tau.\ F_2 = 0)$. The F is instantiated to F_2, and the the E can be instantiated to the following function E_2:

$$\begin{array}{l}
\mathbf{constdefs}\ E_2 :: \mathit{Evt\ list} \Rightarrow \mathit{Evt} \\
\quad E_2\ \tau \equiv (\mathit{if}\ (F_2\ \tau = 3)\ \mathit{then}\ e_2\ \mathit{else} \\
\qquad\qquad \mathit{if}\ (F_2\ \tau = 2)\ \mathit{then}\ e_1\ \mathit{else} \\
\qquad\qquad \mathit{if}\ (F_2\ \tau = 1)\ \mathit{then}\ e_0\ \mathit{else}\ e_0)
\end{array}$$

If N is instantiated to 3, the expression $\{\!|F,\ E,\ N|\!\}$ will be instantiated to $\{\!|F_2,\ E_2,\ 3|\!\}$, which evaluates to

$$[(\lambda\ \tau.\ F_2\ \tau = 3,\ E),\ (\lambda\ \tau.\ F_2\ \tau = 2,\ E),\ (\lambda\ \tau.\ F_2\ \tau = 1,\ E)]$$

From this, it can be derived that:

$$PF\ cs_2\ \{\!|F_2,\ E_2,\ 3|\!\}\ \sigma = \begin{array}{l} USF_\alpha\ cs\ (\lambda\ \tau.\ F_2\ \tau = 3)\ e_2\ \sigma\ \wedge \\ USF_\alpha\ cs\ (\lambda\ \tau.\ F_2\ \tau = 2)\ e_1\ \sigma\ \wedge \\ USF_\alpha\ cs\ (\lambda\ \tau.\ F_2\ \tau = 1)\ e_0\ \sigma \end{array}$$

The right hand side of the above equation is just the premise instances used in Argument 2. Generally, if the value of N equals to the longest possible path from state s to the desired Q-states, $PF\ cs\ \{\!|F,\ E,\ N|\!\}\ \sigma$ will cover the whole path.

Now, let's explain the rule *react-rule*. The premise $PF\ cs\ \{\!|F,\ E,\ N|\!\}\ \sigma$ still has the same meaning as in *resp-rule*, and the proof of *react-rule* is quite similar to the proof of *resp-rule*. If the infinite occurrence of P-states, the existence of a state s and a path from state s to the desired Q-states can be derived, the conclusion of *react-rule* can be derived. The assumption *init* of $REACT$ ensures the existence of state s and the assumption *mid* ensures the existence of a path leading from state s to the desired Q-states.

6 Related works

Approaches for verification of concurrent systems can roughly be divided into theorem proving and model checking. The work in this paper belongs to the theorem proving category, which can deal with infinite state systems directly.

LTL is widely used in specification and verification of concurrent systems. Proof systems for LTL usually are based on weak fairness *WF* or strong fairness *SF* assumptions. As shown in this paper, these two fairness notions are not sufficient to derive the desired property in certain cases. The parametric fairness defined in this paper is an adaption of the α-fairness[17,2,16] to suit the setting of HOL. *PF* is properly stronger than *WF* and *SF*, and, according to Baier's work[2], *PF* still has a sensible semantics based on coin tossing. The use of *PF* can derive more liveness properties and usually the proofs are simpler than using *WF* and *SF*.

Loops in state transition diagram are great obstacles for the proof of liveness properties. To the best of our knowledge, the only LTL proof system, which could tackle the loop problem, is given by Manna and Puneli[8]. The system is proved to be complete. However, an attempt to use the rule F-RESP to prove statement (1) reveals a problem. The rule F-RESP is given in Figure 10.

F-RESP

$$\begin{array}{ll}
\text{F1.} & p \Rightarrow (q \vee \varphi) \\
\left.\begin{array}{ll}
\text{F2.} & \{\varphi_i \wedge (\delta = \alpha)\}\ \mathcal{T}\ \{q \vee \big(\varphi \wedge (\delta \prec \alpha)\big) \vee \big(\varphi_i \wedge (\delta \preceq \alpha)\big)\} \\
\text{F3.} & \{\varphi_i \wedge (\delta = \alpha)\}\ \tau_i\ \{q \vee \big(\varphi \wedge (\delta \prec \alpha)\big)\} \\
\multicolumn{2}{l}{\text{If } \tau_i \in \mathcal{J}\text{, then}} \\
\text{J4.} & \varphi_i \Rightarrow (q \vee En(\tau_i)) \\
\multicolumn{2}{l}{\text{If } \tau_i \in \mathcal{C}\text{, then}} \\
\text{C4.} & \mathcal{F} - \{\tau_i\} \vdash \varphi_i \Rightarrow \Diamond(q \vee En(\tau_i))
\end{array}\right\} & \text{for } i = 1, \ldots, m \\
\hline
 & p \Rightarrow \Diamond\, q
\end{array}$$

Fig. 10. The **F-RESP** rule from [8]

The δ in F-RESP is a measure function similar to F in *resp-rule*. The premises *F1* and *F2* require that the value of δ can never increase, no matter which transition is made in intermediate states. For concurrent systems with loops, like cs_1, we can not see a simple way to define such a δ. A close look at the completeness proof in [8], we find that the existence of such a δ depends on validity of the conclusion $p \Rightarrow \Diamond q$ itself. The rule F-RESP is a 'logical shift' which reduces the proof of $p \Rightarrow \Diamond q$ to the existence of δ, which is equally difficult. For this reason, practical LTL-based systems [9,3,7] only prove simple response properties using simplified versions of F-RESP, and none of them can prove statement (1). We haven't seen any rule which can prove reactivity properties in these systems neither. In these practical systems, complex liveness properties are usually proved using model checking, which is confined to finite state systems. Because the premises of *resp-rule* and *react-rule* are much more liberal than the rules presented in [8], they are powerful enough to deal with general liveness properties for infinite state systems with loops.

7 Conclusion

We have extended Paulson's inductive protocol verification approach to deal with general liveness properties. Different from model checking, this approach is applicable to infinite state systems. We justified the use of a new notion of fairness, parametric fairness, which is stronger than standard ones. The feasibility of our approach is shown by a companion draft paper[18], which proved the liveness of an elevator control system.

References

1. F. Andersen, U. Binau, K. Nyblad, K. D. Petersen, and J. S. Pettersson. The HOL-UNITY verification system. In *Proc. of the Sixth International Joint Conference CAAP/FASE: TAPSOFT'95-Theory and Practice of Software Development*, pages 795–796, Aarhus, Denmark, 1995.
2. C. Baier and M. Kwiatkowska. On the verification of qualitative properties of probabilistic processes under fairness constraints. *Information Processing Letters*, 66(2):71–79, 29 April 1998.
3. A. Browne, A. Kapur, B. Finkbeiner, E. Chang, H. B. Sipma, T. E. Uribe, and Z. Manna. STeP: The stanford temporal prover educational release - user's manual, February 10 1997.
4. M. Devillers, D. Griffioen, and O. Mü. Possibly Infinite Sequences in Theorem Provers: A Comparative Study. In Elsa Gunther, editor, *Theorem Proving in Higher Order Logics (TPHOL'97)*, number LNCS 1275. Springer-Verlag, 1997.
5. B. Heyd and P. Cregut. A modular coding of Unity in Coq. *Lecture Notes in Computer Science*, 1125:251–266, 1996.
6. M. Jaeger. Fairness, computable fairness and randomness. In *Proc. 2nd International Workshop on Probabilistic Methods in Verification*, 1999.

7. Z. Manna, A. Anuchitanukul, N. Bjorner, A. Browne, E. Chang, M. Colon, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: The stanford temporal prover. Technical Report CS-TR-94-1518, Stanford University, Department of Computer Science, June 1994.
8. Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comput. Sci.*, 83(1):91–130, 1991.
9. Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 726–765. Springer, Berlin, Heidelberg, 1994.
10. O. Muller. I/O automata and beyond: Temporal logic and abstraction in isabelle. In *Proc. 11th International Theorem Proving in Higher Order Logics Conference*, pages 331–348, 1998.
11. O. Müller and T. Nipkow. Traces of I/O-automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *Proceedings of the Seventh International Joint Conference on the Theory and Practice of Software Development (TAPSOFT'97)*, Lille, France, April 1997. Springer-Verlag LNCS 1214.
12. L. Paulson. Mechanizing UNITY in Isabelle. *ACM Trans. Comput. Logic*, 1(1):3–32, 2000.
13. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.
14. L. C. Paulson. Inductive analysis of the Internet protocol TLS. *ACM Transactions on Computer and System Security*, 2(3):332–351, 1999.
15. L. C. Paulson. Verifying the SET protocol: Overview. In *International Conference on Formal Aspects of Security (FASec), LNCS*, volume 1, 2002.
16. A. Pnueli. On the extremely fair treatment of probabilistic algorithms. In ACM, editor, *Proceedings of the 15th annual ACM Symposium on Theory of Computing, Boston, Massachusetts, April 25–27, 1983*, pages 278–290, New York, NY, USA, 1983. ACM Press.
17. A. Pnueli and L. D. Zuck. Probabilistic verification. *Information and Computation*, 103(1):1–29, March 1993.
18. H. Yang, X. Zhang, and Y. Wang. Liveness proof of an elevator control system. In *Draft paper to be published, available through E-mail request*, 2005.

Author Index