Executing Higher Order Logic

Stefan Berghofer* and Tobias Nipkow

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

Abstract. We report on the design of a prototyping component for the theorem prover Isabelle/HOL. Specifications consisting of datatypes, recursive functions and inductive definitions are compiled into a functional program. Functions and inductively defined relations can be mixed. Inductive definitions must be such that they can be executed in Prolog style but requiring only matching rather than unification. This restriction is enforced by a mode analysis. Tail recursive partial functions can be defined and executed with the help of a `while` combinator.

1 Introduction

Executing formal specifications has been a popular research topic for some decades, covering every known specification formalism. Executability is essential for validating complex specifications by running test cases and for generating code automatically ("rapid prototyping"). In the theorem proving community executability is no less of an issue. Two prominent examples are the Boyer-Moore system and its successor ACL2 [11] or constructive type theory, both of which contain a functional programming language. In contrast, HOL specifications can be highly non-executable, and various approaches to their execution have been reported in the literature (see §5 for references). The aim of our paper is to give a precise definition of an executable subset of HOL and to describe its compilation into a functional programming language.

The basic idea is straightforward: datatypes and recursive functions compile directly into their programming language equivalents, and inductive definitions are executed like Prolog programs. Things become interesting when functions and relations are mixed.

We are the first to acknowledge that very few of the ideas in this paper are genuinely original. Instead we flatter ourselves by believing we have achieved a new blend of HOL and functional-logic programming that may serve as the basis for many future approaches to executing HOL. In particular we have precisely identified a subset of HOL definitions that allow efficient execution (§2), outlined a compilation schema for inductive definitions (§3), and devised a method for the

* Supported by DFG Graduiertenkolleg *Logic in Computer Science*, and IST project 29001 *TYPES*

definition and execution of tail recursive functions without tears (§4). Our aim has not been to reach or extend the limits of functional-logic programming but to design a lightweight and efficient execution mechanism for HOL specifications that requires only a functional programming language and is sufficient for typical applications like execution of programming language semantics or abstract machines.

2 An executable subset of Isabelle/HOL

As promised in the introduction, we now give a more precise definition of the executable subset of the specification language *Isabelle/HOL*, which is based on Church's simple theory of types. The main ingredients of HOL specifications are:

inductive datatypes can be defined by specifying their *constructors*, e.g.

> **datatype** nat = 0 | Suc nat

recursive functions can be defined by specifying several characteristic equations, e.g.

> **primrec**
> add 0 y = y
> add (Suc x) y = Suc (add x y)

> All functions in HOL must be terminating. Supported recursion schemes are *primitive recursion* (**primrec**) and *well-founded recursion* (**recdef**) [20].

inductive relations (or predicates) can be defined by specifying a set of *introduction rules*, e.g.

> **inductive**
> $0 \in$ even
> $x \in$ even $\Longrightarrow$ Suc (Suc x) $\in$ even

> Introduction rules are essentially *Horn Clauses*, which are also used in logic programming languages such as Prolog.

Recursive functions and inductive definitions may also be intermixed: For example, an inductive predicate may refer to a recursive function and vice versa.

Executable elements of HOL specifications We now inductively define the elements an *executable* HOL specification may consist of:

- **Executable terms** contain only executable constants
- **Executable constants** can be one of the following
  - executable inductive relations
  - executable recursive functions
  - constructors, recursion and case combinators of executable datatypes
  - operators on executable primitive types such as bool, i.e. the usual propositional operators $\wedge$, $\vee$ and $\neg$, as well as `if _ then _ else _`.

- **Executable datatypes**, where each constructor argument type is again an executable datatype or an executable primitive type such as bool or $\rightarrow$.
- **Executable inductive relations**, whose introduction rules have the form

$$(u_1^1,\ \ldots,\ u_{n_1}^1) \in q_1 \Longrightarrow \ldots \Longrightarrow (u_1^m,\ \ldots,\ u_{n_m}^m) \in q_m \Longrightarrow (t_1,\ \ldots,\ t_k) \in p$$

  where u_j^i and t_i are executable terms and q_i is either p or some other executable inductive relation. In addition, also arbitrary executable terms not of the form $(\ldots) \in p_i$, so-called *side conditions*, which may not contain p, are allowed as premises of introduction rules.
- **Executable recursive functions**, i.e. sets of rewrite rules, whose left-hand side contains only constructor patterns with distinct variables, and the right-hand side is an executable term.

In the sequel, we write $\mathcal{C}$ to denote the set of datatype constructors. Note that in the above definition, we view $t \in p$, where p is an inductive relation, as synonymous with $p(t)$. Thus, the term $t \in p$ is executable, provided that t and p are, whereas a term of the form $t \in u$ is not executable in general.

The non-executable elements of HOL are, among others, arbitrary universal and existential quantification, equality of objects having higher-order types, Hilbert's selection operator ε, arbitrary type definitions (other than datatypes) or inductive definitions whose introduction rules contain quantifiers, like

$$(\forall y.\ (y,\ x) \in r \Longrightarrow y \in \mathsf{acc}\ r) \Longrightarrow x \in \mathsf{acc}\ r$$

Execution What exactly do we mean by *execution* of specifications? Essentially, execution means finding solutions to queries. A *solution* σ is a mapping of variables to *closed solution terms*. A term t is called a *solution term* iff

- t is of function type, or
- $t = c\ t_1\ \ldots\ t_n$, where the t_i are solution terms and $c \in \mathcal{C}$.

Let solve be a function that returns for each query a set of solutions. We distinguish two kinds of queries:

Functional queries have the form $t = X$, where t is a closed executable term and X is a variable. Queries of this kind should return at most one solution, e.g. $\mathsf{solve}(\mathsf{add}\ 0\ (\mathsf{Suc}\ 0) = X) = \{[X \mapsto \mathsf{Suc}\ 0]\}$

Relational queries have the form $(t_1, \ldots, t_n) \in r$, where r is an executable inductively defined relation and t_i is either a closed executable term or a variable. A query Q of this kind returns a set of solutions $\mathsf{solve}(Q)$. Note that the set returned by solve may also be empty, e.g. $\mathsf{solve}(\mathsf{Suc}\ 0 \in \mathsf{even}) = \{\}$, or infinite, e.g. $\mathsf{solve}(X \in \mathsf{even}) = \{[X \mapsto 0], [X \mapsto \mathsf{Suc}\ (\mathsf{Suc}\ 0)], \ldots\}$.

It is important to note that all relational queries have to be *well-moded* in order to be executable. We will make this notion more precise in §3.1.

The restriction to ground terms in queries can be relaxed at the expense of the complexity and efficiency of the solver. We consider this an optional extension not necessary for our primary application areas (see §6).

Correctness and completeness Function solve is called *sound* w.r.t. an executable HOL specification *Spec* iff

$$\sigma \in \mathsf{solve}(Q) \Longrightarrow \mathit{Spec} \vdash \sigma(Q)$$

Here, $\vdash$ denotes derivability using introduction and elimination rules for $\forall$ and $\Longrightarrow$ as well as the substitution rule. The former corresponds to the execution of logic programs, while the latter corresponds to functional execution.

Completeness is more subtle. We omit its definition as we will not be able to guarantee completeness anyway.

3 Compiling functional logic specifications

Functional-logic programming languages such as Curry [9] should be ideal target languages for code generation from HOL specifications. But although such languages contain many of the required concepts and there is an impressive amount of research in this area, the implementations which are currently available are not always satisfactory. We therefore decided to choose ML, the implementation language of Isabelle, as a target language. Datatypes and recursive functions can be translated to ML in a rather straightforward way, with only minor syntactic modifications. Therefore, this section concentrates on the more interestinging task of translating inductive relations to ML.[1] The translation is based on assigning *modes* to relations, a well-known standard technique for the analysis and optimization of logic programs [12].

3.1 Mode analysis

In order to translate a predicate into a function, the direction of *dataflow* has to be analyzed, i.e. it has to be determined which arguments are *input* and which are *output*. Note that for a predicate there may be more than one possible direction of dataflow. For example, the predicate

$$(\mathsf{Nil},\ ys,\ ys) \in \mathsf{append}$$
$$(xs,\ ys,\ zs) \in \mathsf{append} \Longrightarrow (\mathsf{Cons}\ x\ xs,\ ys,\ \mathsf{Cons}\ x\ zs) \in \mathsf{append}$$

may be given two lists $xs = [1,\ 2]$ and $ys = [3,\ 4]$ as input, the output being the list $zs = [1,\ 2,\ 3,\ 4]$. We may as well give a list $zs = [1,\ 2,\ 3,\ 4]$ as an input, the output being a sequence of pairs of lists xs and ys, where zs is the result of appending xs and ys, namely $xs = [1,\ 2,\ 3,\ 4]$ and $ys = []$, or $xs = [1,\ 2,\ 3]$ and $ys = [4]$, or $xs = [1,\ 2]$ and $ys = [3,\ 4]$, etc.

Mode assignment A specific direction of dataflow is called a *mode*. We describe a mode of a predicate by a set of indices, which denote the positions of the input arguments. In the above example, the two modes described were $\{1,\ 2\}$ and $\{3\}$. Given a set of predicates P, a relation *modes* is called a *mode assignment* if

$$\mathit{modes} \subseteq \{(p,\ M) \mid p \in P \wedge M \subseteq \{1,\ \ldots,\ \mathsf{arity}\ p\}\}$$

[1] Translation into Haskell looks simpler because of lazy lists and list comprehension, but has essentially the same intellectual and computational complexity.

The set

$$\mathit{modes}\ p = \{M \mid (p,\ M) \in \mathit{modes}\} \qquad \subseteq \mathcal{P}(\{1,\ \ldots,\ \mathsf{arity}\ p\})$$

is the set of modes assigned to predicate p.

Consistency of modes A mode M is called *consistent* with respect to a mode assignment *modes* and a clause

$$(u_1^1,\ \ldots,\ u_{n_1}^1) \in q_1 \Longrightarrow \ldots \Longrightarrow (u_1^m,\ \ldots,\ u_{n_m}^m) \in q_m \Longrightarrow (t_1,\ \ldots,\ t_k) \in p$$

if there exists a permutation π and sets of variable names v_0, ..., v_m such that

(1) $v_0 = \mathsf{vars\_of}\ (\mathsf{args\_of}\ M\ (t_1,\ \ldots,\ t_k))$

(2) $\forall 1 \leq i \leq m.\ \exists M' \in \mathit{modes}\ q_{\pi(i)}.\ M' \subseteq \mathsf{known\_args}\ v_{i-1}\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})$

(3) $\forall 1 \leq i \leq m.\ v_i = v_{i-1} \cup \mathsf{vars\_of}\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})$

(4) $\mathsf{vars\_of}\ (t_1,\ \ldots,\ t_k) \subseteq v_m$

The permutation π denotes a suitable execution order for the predicates q_1, ..., q_m in the body of p, where v_i is the set of variables whose value is known after the ith execution step. Condition (1) means that initially, when invoking mode M of predicate p, the values of all variables occurring in the input arguments of the clause head are known. Condition (2) means that in order to invoke a mode M' of a predicate $q_{\pi(i)}$, all of the predicate's input arguments which are specified by M' must be known. According to condition (3), the values of all arguments of $q_{\pi(i)}$ are known after its execution. Finally, condition (4) states that the values of all variables occurring in the clause head of p must be known. Here, function $\mathsf{args\_of}\ M$ returns the tuple of input arguments specified by mode M, e.g.

$$\mathsf{args\_of}\ \{1,\ 2\}\ (\mathsf{Cons}\ x\ xs,\ ys,\ \mathsf{Cons}\ x\ zs) = (\mathsf{Cons}\ x\ xs,\ ys)$$

Function $\mathsf{vars\_of}$ returns all variables occurring in a tuple, e.g.

$$\mathsf{vars\_of}\ (\mathsf{Cons}\ x\ xs,\ ys) = \{x,\ xs,\ ys\}$$

Given some set of variables and an argument tuple, $\mathsf{known\_args}$ returns the indices of all arguments, whose value is fully known, provided the values of the variables given are known, e.g.

$$\mathsf{known\_args}\ \{x,\ xs,\ ys\}\ (\mathsf{Cons}\ x\ xs,\ ys,\ \mathsf{Cons}\ x\ zs) = \{1,\ 2\}$$

Mode inference We write

$$\mathsf{consistent}\ (p,\ M)\ \mathit{modes}$$

if mode M of predicate p is consistent with respect to all clauses of p, under the mode assignment *modes*. Let

$$\Gamma(\mathit{modes}) = \{(p,\ M) \mid (p,\ M) \in \mathit{modes} \wedge \mathsf{consistent}\ (p,\ M)\ \mathit{modes}\}$$

Then the greatest set of allowable modes for a set of predicates P is the greatest fixpoint of Γ. According to Kleene's fixpoint theorem, since Γ is monotone and its

domain is finite, this fixpoint can be obtained by finite iteration: we successively apply Γ, starting from the greatest mode assignment

$$\{(p,\ M) \mid p \in P \wedge M \subseteq \{1,\ \ldots,\ \mathsf{arity}\ p\}\}$$

until a fixpoint is reached.

Example For append, the allowed modes are inferred as follows:

$\{\}$ is illegal, because it is impossible to compute the value of ys in the first clause
$\{1\}$ is illegal for the same reason
$\{2\}$ is illegal, because it is impossible to compute the value of x in the second clause
$\{3\}$ is legal, because
- in the first clause, we can compute the first and second argument (Nil, ys) from the third argument ys
- in the second clause, we can compute x and zs from the third argument. By recursively calling append with mode $\{3\}$, we can compute the value of xs and ys. Thus, we also know the value of the first and second argument (Cons x xs, ys).

$\{1,\ 2\}$ is legal, because
- in the first clause, we can compute the third argument ys from the first and second argument
- in the second clause, we can compute x, xs and ys from the first and second argument. By recursively calling append with mode $\{1,\ 2\}$, we can compute the value of zs. Thus, we also have the value of the third argument Cons x zs

$\{1,\ 3\}$, $\{2,\ 3\}$, $\{1,\ 2,\ 3\}$ are legal as well (see e.g. $\{3\}$)

Well-moded queries A query $(t_1, \ldots, t_n) \in p$ is called *well-moded* with respect to a mode assignment *modes* iff

$$\{i \mid t_i \text{ is not a variable}\} \in \mathit{modes}\ p$$

Mixing predicates and functions The above conditions for the consistency of modes are sufficient, if the only functions occurring in the clauses are *constructor functions*. If we allow arbitrary functions to occur in the clauses, we have to impose some additional restrictions on the positions of their occurrence. Since non-constructor functions may not be inverted, they cannot appear in an *input position* in the clause head or in an *output position* in the clause body. Thus, we rephrase conditions (1) and (2) to

$$(1') \quad v_0 = \mathsf{vars\_of}\ (\mathsf{args\_of}\ \{i \in M \mid \mathsf{funs\_of}\ t_i \subseteq \mathcal{C}\}\ (t_1,\ \ldots,\ t_k)) \wedge \forall i \in M.\ \mathsf{funs\_of}\ t_i \not\subseteq \mathcal{C} \longrightarrow \mathsf{eqtype}\ t_i$$

$$(2') \quad \forall 1 \leq i \leq m.\ \exists M' \in \mathit{modes}\ q_{\pi(i)}.\ M' \subseteq \mathsf{known\_args}\ v_{i-1}\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)}) \wedge \mathsf{funs\_of}\ (\mathsf{args\_of}\ (\{1, \ldots, \mathsf{arity}\ q_{\pi(i)}\} \backslash M')\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})) \subseteq \mathcal{C}$$

where $\mathcal{C}$ is the set of constructor functions and funs_of returns the set of all functions occurring in a tuple. The intuition behind $(1')$ is as follows: if some of

the input parameters specified by M contain non-constructor functions, we try mode analysis with a subset of M that does not contain the problematic input parameters. After successful execution, we compare the computed values of t_j, where $j \in M \wedge \mathsf{funs\_of}\ t_j \not\subseteq \mathcal{C}$, with the values provided as input arguments to the predicate. For this to work properly, the terms t_j need to have an *equality type*, i.e. not be of a function type or a datatype involving function types. Note that any M_2 with $M_1 \subseteq M_2$ will be a valid mode, provided M_1 is a valid mode and $\forall j \in M_2 \backslash M_1.\ \mathsf{funs\_of}\ t_j \not\subseteq \mathcal{C} \longrightarrow \mathsf{eqtype}\ t_j$. As condition $(2')$ suggests, we can get around the restriction on the occurrence of non-constructor functions in the clause body by choosing modes M' which are sufficiently large, i.e. have sufficiently many input parameters.

3.2 Translation scheme

In the following section, we will explain how to translate predicates given by a set of Horn Clauses into functional programs in the language ML. For each legal mode of a predicate, a separate function will be generated. Given a tuple of input arguments, a predicate may return a potentially infinite sequence of result tuples. Sequences are represented by the type `'a seq` which supports the following operations:

```
Seq.empty  : 'a seq
Seq.single : 'a -> 'a seq
Seq.append : 'a seq * 'a seq -> 'a seq
Seq.map    : ('a -> 'b) -> 'a seq -> 'b seq
Seq.flat   : 'a seq seq -> 'a seq
```

In the sequel, we will write `s1 ++ s2` instead of `Seq.append (s1, s2)`. In addition, we define the operator

```
fun s :-> f = Seq.flat (Seq.map f s);
```

which will be used to compose subsequent calls of predicates. Using these operators, the modes $\{1,\ 2\}$ and $\{3\}$ of predicate `append` can be translated into the ML functions

```
append_1_2 : 'a list * 'a list -> 'a list seq
append_3   : 'a list -> ('a list * 'a list) seq
```

which are defined as follows:

```
fun append_1_2 inp =
  Seq.single inp :->
    (fn (Nil, ys) => Seq.single (ys) | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (Cons (x, xs), ys) =>
      append_1_2 (xs, ys) :->
        (fn (zs) => Seq.single (Cons (x, zs)) | _ => Seq.empty)
    | _ => Seq.empty);
```

```
fun append_3 inp =
  Seq.single inp :->
    (fn (ys) => Seq.single (Nil, ys) | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (Cons (x, zs)) =>
      append_3 (zs) :->
        (fn (xs, ys) => Seq.single (Cons (x, xs), ys)
          | _ => Seq.empty)
      | _ => Seq.empty);
```

In the above translation, every operand of `++` corresponds to one clause of the predicate. Initially, the input is converted into a one-element sequence using `Seq.single`, to which successively all predicates in the body of the clause are applied using `:->`. Therefore, the operator `:->` can also be interpreted as a visualization of dataflow.

We will now describe the general translation scheme. Assume the predicate to be translated has the clause

$$(ipat_1,\ opat_1) \in q_1 \Longrightarrow \ldots \Longrightarrow (ipat_m,\ opat_m) \in q_m \Longrightarrow (ipat_0,\ opat_0) \in p$$

To simplify notation, we assume without loss of generality that the predicates in the body of p are already sorted with respect to the permutation π calculated during mode analysis and that the arguments of the predicates are already partitioned into input arguments $ipat_i$ and output arguments $opat_i$. Then, p is translated into the function

```
fun p inp =
  Seq.single inp :->
    (fn ipat_0 => q_1 ipat_1 :->
        (fn opat_1 => q_2 ipat_2 :->
            ...
                (fn opat_m => Seq.single opat_0
                  | _ => Seq.empty)
              ⋮
          | _ => Seq.empty)
      | _ => Seq.empty)
  ++
  ...;
```

where the ... after the operator `++` correspond to the translation of the remaining clauses of p. A characteristic feature of this translation is the usage of ML's built-in pattern matching mechanism instead of unification and logical variables. Before calling a predicate q_i in the body of the clause, the output pattern $opat_{i-1}$ of the preceeding predicate is checked. Before calling the first predicate q_1, the input pattern $ipat_0$ in the head of the clause is checked.

Example: some λ-calculus theory formalized As an example of a program making use of both functional and logical features, we now consider a specification of β-reduction for λ-terms in de Bruijn notation, which is taken from [13].

First, the datatype `term` of λ-terms is defined, together with a function lift for incrementing indices in a term as well as a function subst for substituting a term for a variable with a given index:

datatype term = Var nat | App term term | Abs term

primrec
lift (Var i) k = (if $i < k$ then Var i else Var $(i + 1)$)
lift (App s t) k = App (lift s k) (lift t k)
lift (Abs s) k = Abs (lift s $(k + 1)$)

primrec
subst (Var i) s k = (if $k < i$ then Var $(i - 1)$ else if $i = k$ then s else Var i)
subst (App t u) s k = App (subst t s k) (subst u s k)
subst (Abs t) s k = Abs (subst t (lift s 0) $(k + 1)$)

This is a purely functional specification, whose translation to ML is straightforward. It is therefore not shown here. Using subst, one can now define beta reduction $\rightarrow_\beta$ inductively:

inductive
App (Abs s) t $\rightarrow_\beta$ subst s t 0
$s \rightarrow_\beta t \Longrightarrow$ App s u $\rightarrow_\beta$ App t u
$s \rightarrow_\beta t \Longrightarrow$ App u s $\rightarrow_\beta$ App u t
$s \rightarrow_\beta t \Longrightarrow$ Abs s $\rightarrow_\beta$ Abs t

Note that $t \rightarrow_\beta u$ just abbreviates $(t,\ u) \in \rightarrow_\beta$. This specification of β-reduction is essentially a functional logic program. Using the translation scheme described above, the HOL specification of $\rightarrow_\beta$ can be translated to the following ML program for mode $\{1\}$:

```
fun beta_1 inp =
  Seq.single inp :->
    (fn (App (Abs s, t)) =>
      Seq.single (subst s t 0) | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (App (s, u)) =>
      beta_1 (s) :->
        (fn (t) => Seq.single (App (t, u)) | _ => Seq.empty)
      | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (App (u, s)) =>
      beta_1 (s) :->
        (fn (t) => Seq.single (App (u, t)) | _ => Seq.empty)
      | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (Abs s) =>
      beta_1 (s) :->
        (fn (t) => Seq.single (Abs t) | _ => Seq.empty)
      | _ => Seq.empty);
```

Note that the recursive function `subst` can easily be called from within the logic program `beta_1`.

Running the translated program We will now try out the compiled predicate on a small example: the sequence

```
val test = beta_1 (Abs (App (Abs (App (Var 0, Var 0)),
   App (Abs (App (Var 0, Var 0)), Var 0))));
```

contains the possible reducts of the term $\lambda x.\ (\lambda y.\ y\ y)\ ((\lambda z.\ z\ z)\ x)$. The first element of this sequence is

```
> Seq.hd test;
val it = Abs (App (App (Abs (App (Var 0, Var 0)), Var 0),
   App (Abs (App (Var 0, Var 0)), Var 0)))
```

There is yet another solution for our query, namely

```
> Seq.hd (Seq.tl test);
val it = Abs (App (Abs (App (Var 0, Var 0)), App (Var 0, Var 0)))
```

3.3 Extending the mode system

The mode system introduced in §3.1 is not always sufficient: For example, it does not cover inductive relations such as

inductive
$(x,\ x) \in \mathsf{trancl}\ r$
$(x,\ y) \in r \Longrightarrow (y,\ z) \in \mathsf{trancl}\ r \Longrightarrow (x,\ z) \in \mathsf{trancl}\ r$

which take other inductive relations as arguments. This case can be covered by introducing so-called *higher-order modes*: a mode of a higher-order relation p taking n relations $r_1, \ldots, r_l$ as arguments and returning a relation as result is an $n+1$ tuple, where the first n components of the tuple correspond to the modes of the argument relations, and the last component corresponds to the mode of the resulting relation, i.e.

$$\mathit{modes}\ p \subseteq \mathcal{P}(\{1, \ldots, \mathsf{arity}\ r_1\}) \times \cdots \times \mathcal{P}(\{1, \ldots, \mathsf{arity}\ r_l\}) \times \mathcal{P}(\{1, \ldots, \mathsf{arity}\ p\})$$

For example, trancl has modes $\{(\{1\}, \{1\}), (\{2\}, \{2\}), (\{1, 2\}, \{1, 2\})\}$, i.e. if r has mode $\{1\}$ then trancl r has mode $\{1\}$ as well. A higher-order relation may have clauses of the form

$$(u_1^1, \ldots, u_{n_1}^1) \in Q_1 \Longrightarrow \ldots \Longrightarrow (u_1^m, \ldots, u_{n_m}^m) \in Q_m \Longrightarrow (t_1, \ldots, t_k) \in p\ r_1 \ldots r_l$$

where $Q_{i'} = r_i \mid q_j\ Q'_{\varrho_1}\ \ldots\ Q'_{\varrho_{l'}}$

To describe the consistency of a higher order mode $(M_1,\ \ldots,\ M_l,\ M)$ with respect to a mode assignment *modes* and the above clause, we rephrase condition (2) of the definition of consistency given in §3.1 to

$$(2')\quad \forall 1 \leq i \leq m.\ \exists M' \in \mathit{modes}'\ Q_{\pi(i)}.\ M' \subseteq \mathsf{known\_args}\ v_{i-1}\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})$$

where
$\mathit{modes}'\ r_i = \{M_i\}$
$\mathit{modes}'\ (q_j\ Q'_{\varrho_1}\ \ldots\ Q'_{\varrho_{l'}}) = \{M' \mid \exists M'_1 \in \mathit{modes}'\ Q'_{\varrho_1} \ldots M'_{l'} \in \mathit{modes}'\ Q'_{\varrho_{l'}}.$
$\qquad (M'_1,\ \ldots,\ M'_{l'},\ M') \in \mathit{modes}\ q_j\}$

Mode $\{1\}$ of trancl could be translated as follows

```
fun trancl_1 r inp =
  Seq.single inp ++ r inp :-> trancl_1 r;
```

Interestingly, the translation of mode $\{2\}$ looks exactly the same.

3.4 Some notes on correctness and completeness

We claim that our translation scheme decribed above is correct in the sense of §2, although a formal proof is beyond the scope of this paper. As far as completeness is concerned, there are two problems that have to be taken into consideration. One problem is due to ML's *eager evaluation* strategy. For example,

defs
$g \equiv \lambda x\ y.\ x$
$f_1 \equiv \lambda x.\ g\ x\ (\mathsf{hd}\ [])$
recdef
$f_2\ 0 = 0$
$f_2\ (\mathsf{Suc}\ x) = g\ (f_2\ x)\ (f_2\ (\mathsf{Suc}\ x))$

is an admissible HOL specification, but, if compiled naively, f_1 raises an exception, because the argument [] cannot be handled by hd, and f_2 loops. To avoid this, the definition of g could be expanded, or the critical arguments could be wrapped into dummy functions, to delay evaluation. Paulin-Mohring and Werner [16] discuss this problem in detail and also propose alternative target languages with *lazy evaluation*.

Another source of possible nontermination is the Prolog-style *depth first* execution strategy of the translated inductive relations. Moreover, some inferred modes or permutations of predicates in the body of a clause may turn out to be non-terminating. Termination of logic programs is an interesting problem in its own right, which we do not attempt to solve here.

4 Partial functions

So far we have (implicitly) assumed that HOL is a suitable logic for defining recursive functions, i.e. that it is possible to define functions by recursion equations as in functional programming languages. This is indeed the case for primitive recursion and still works well for well-founded recursion. However, all HOL functions must be total. Hence we cannot hope to define truly partial functions. The best we can do are functions that are *underdefined*: for certain arguments we only know that a result exists, but we don't know what it is. When defining functions that are normally considered partial, underdefinedness turns out to be a very reasonable alternative. We will now discuss two issues: how to define such underdefined functions in the first place, and how to obtain recursion equations that allow efficient execution.

4.1 Guarded recursion

Given a partial function f that should satisfy the recursion equation $f(x) = t$ over its domain $dom(f)$, we turn this into the guarded recursion

```
f x = (if x ∈ dom f then t else arbitrary)
```

where *arbitrary* is a constant of type *'a* which has no definition, i.e. its value is completely underspecified. As a simple example we define division on *nat*:

```
consts divi :: "nat × nat → nat"
recdef divi "measure(λ(m,n). m)"
  "divi(m,n) = (if n = 0 then arbitrary else
                if m < n then 0 else divi(m-n,n)+1)"
```

The keyword **consts** declares constants and **recdef** defines a function by well-founded recursion — the term *measure (λ(m, n). m)* is the well-founded relation [20]. For the sake of the example, we equate *divi (m, 0)* with *arbitrary* rather than some specific number, for example 0.

As a more substantial example we consider the problem of searching a graph. For simplicity our graph is given by a function (*f*) of type *'a → 'a* which maps each node to its successor, and the task is to find the end of a chain, i.e. a node pointing to itself. Here is a first attempt:

```
find (f, x) = (if f x = x then x else find (f, f x))
```

This may be viewed as a fixed point finder or as one half of the well known *Union-Find* algorithm. The snag is that it may not terminate if *f* has non-trivial cycles. Phrased differently, the relation

```
constdefs step1 :: "('a → 'a) → ('a × 'a)set"
  "step1 f ≡ {(y,x). y = f x ∧ y ≠ x}"
```

must be well-founded (in Isabelle/HOL: *wf*). Thus we define

```
consts find :: "('a → 'a) × 'a → 'a"
recdef find
  "find(f,x) = (if wf(step1 f)
                then if f x = x then x else find(f, f x)
                else arbitrary)"
```

The recursion equation should be clear enough: it is our aborted first attempt augmented with a check that there are no non-trivial cycles. We omit to show the accompanying termination relation, which is not germane to the subject of our paper.

Although the above definition of *find* is quite satisfactory from a theorem proving point of view, it is a disaster w.r.t. executability: the test *wf (step1 f)* is undecidable in general. This is the key problem with guarded recursion: unless the domain of the function is (efficiently) decidable, this scheme does not yield (efficiently) executable functions.

4.2 The *while* combinator

Fortunately, tail recursive functions admit natural HOL definitions which can be executed efficiently. This insight was communicated to us by Wolfgang Goerigk [8]. To understand why, consider the following two "definitions":

$f(x) = f(x+1)$ is perfectly harmless, as it merely asserts that all values of f are the same, but leaving the precise value open.

$f(x) = f(x) + 1$ must not be admitted because there is no total function that satisfies this equation and it implies $0 = 1$.

The key property of tail recursive function definitions is that they have total models. Instead of dealing with arbitrary tail recursive function definitions we introduce a *while* combinator. This is merely a notational variant (and simplification) of tail recursion. The definition of *while* follows the guarded recursion schema, but we omit to show the termination relation:

```
consts while :: "('a → bool) × ('a → 'a) × 'a → 'a"
recdef while
  "while(b,c,s) = (if ∃f. f 0 = s ∧ (∀i. b(f i) ∧ c(f i) = f(i+1))
                   then arbitrary
                   else if b s then while(b,c,c s) else s)"
```

The guard checks if the loop terminates or not. If it does, *while(b,c,s)* mimicks the imperative program

```
x := s; while b(x) do x := c(x); return x
```

It appears we have not made much progress because the definition of *while* is certainly not executable. However, it is possible to derive the following unguarded recursion equation:

```
theorem while_rec: "while(b,c,s) = (if b s then while(b,c,c s) else s)"
```

The proof is easy. If the loop terminates, the left-hand side reduces by definition to the right-hand side. If it does not terminate, the left-hand side is *arbitrary*, as is the right-hand side: nontermination implies that *b s* must hold, in which case the right-hand side reduces to *while (b, c, c s)*, which diverges as well, and thus to *arbitrary*.

What has happened here is well-known in the program transformation literature: starting from a total function f it is easy to derive a recursion equation for f which, if interpreted as a function definition, yields a partial version of the original f. As an extreme example, one can always prove $f(x) = f(x)$, which is as partial as can be.

This phenomenon is usually considered a problem, but for us it is the solution: in HOL we are forced to define a total version of *while*, but can recover the desired partial one by proof. And instead of generating code for *while* from its definition, we generate the code from theorem *while_rec*, which any decent compiler will then translate into a loop. In fact, the code generator is always

driven by theorems. Thus the generated function definitions are necessarily partially correct, provided the code generator is correct. Basing everything upon theorems is possible because Isabelle/HOL follows the definitional approach: recursive functions are not axiomatized but the recursion equations are *proved* from a suitable non-recursive definition — the latter process is hidden from the user [20]. Thus the code generator takes an arbitrary list of theorems, checks they constitute a well-formed function definition (merely to avoid static errors later on) and translates them into ML. It makes no difference whether these theorems correspond to the initial definition of the function or are derived from it, as is the case for `while_rec`.

As an application of `while` we define the above function `find` without tears:

```
constdefs find2 :: "('a → 'a) → 'a → 'a"
  "find2 f x ≡
   fst(while (λ(x,x'). x' ≠ x, λ(x,x'). (x',f x'), (x,f x)))"
```

The loop operates on two "local variables" `x` and `x'` containing the "current" and the "next" value of function `f`. They are initalized with the global `x` and `f x`. At the end `fst` selects the local `x`.

Although the definition of `find2` was easy, there is no free lunch: when proving properties of functions defined by `while`, termination rears its ugly head again. Such proofs are best conducted by means of the derived `while_rule`, the well known proof rule for total correctness of loops expressed with `while`:

```
⟦P s; ⋀s. ⟦P s; b s⟧ ⟹ P (c s);
   ⋀s. ⟦P s; ¬ b s⟧ ⟹ Q s; wf r;
   ⋀s. ⟦P s; b s⟧ ⟹ (c s, s) ∈ r⟧
⟹ Q (while (b, c, s))
```

`P` needs to be true of the initial state `s` and invariant under `c` (premises 1 and 2). The post-condition `Q` must become true when leaving the loop (premise 3). And each loop iteration must descend along a well-founded relation `r` (premises 4 and 5). In order to show that `find2` does indeed find a fixed point

```
theorem "wf(step1 f) ⟹ f(find2 f x) = find2 f x"
```

we prove the following lemma with the help of `while_rule`:

```
lemma lem: "⟦ wf(step1 f); x' = f x ⟧ ⟹
   ∃y. while (λ(x,x'). x' ≠ x, λ(x,x'). (x',f x'), (x,x')) = (y,y) ∧
       f y = y"
```

The proof is almost automatic after supplying the invariant `x' = f x` and the termination relation.

As we have seen, the `while` combinator has the advantage of enabling us to define functions without having to worry about termination. However, this merely delays the evil hour, which comes as soon as one wants to prove properties of a function thus defined. On top of that, tail recursive functions tend to be more complicated to reason about. Therefore `while` should only be used if executability is an issue or the function in question is naturally tail recursive.

5 Related work

Previous work on executing HOL specifications There has already been some work on generating executable programs from specifications written in HOL. One of the first papers on this topic is by Rajan [19] who translates HOL datatypes and recursive functions to ML. However, inductive definitions are not covered in this paper. Andrews [2] has chosen λProlog as a target language. His translator for a higher order specification language called S can also handle specifications of transition rules of programming languages such as CCS, although these are given in a somewhat different way than the inductive definitions of Isabelle/HOL. In contrast to our approach, all functions have to be translated into predicates in order to be executable by λProlog. On the other hand it is possible to execute a wider range of specifications and queries, as λProlog allows embedded universal quantifiers and implications and supports higher-order unification, with all the performance penalties this entails. Similar comments apply to Elf [18], a type theoretic higher-order logic programming language.

Other theorem provers Aagaard et al [1] introduce a functional language called fl, together with a suitable theorem prover. Thanks to a "lifting" mechanism, their system supports both *execution* of fl functions as well as *reasoning* about fl functions in a seamless way.

Coq [4] is a type-theoretic proof assistant based on the *Calculus of Inductive Constructions*. Type theory allows for the uniform treatment of both proofs and programs within the same framework. In contrast to HOL, where computable and non-computable objects can be arbitrarily mixed, Coq strictly distinguishes between types that have a *computational content* and types that don't. This is done by introducing two *universes* called Set and Prop, the first of which contains types that have a *computational content*. Coq can directly generate code from definitions of recursive datatypes and functions such as nat and add from §2. To obtain a program from an inductive predicate such as append, an approach substantially different from ours described in §3 is used: one builds a constructive proof of

$$\forall(xs :: \alpha\ \mathsf{list})\ (ys :: \alpha\ \mathsf{list}).\ \exists(zs :: \alpha\ \mathsf{list}).\ \mathsf{append}\ xs\ ys\ zs$$

from which a functional program of type $\alpha\ \mathsf{list} \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$ can be *extracted* by erasing the parts of the proof which are in Prop. Note that this method relies on the inductive predicate to be decidable. Paulin-Mohring and Werner [16] describe how to obtain efficient functional programs from Coq specifications.

The latest version of the theorem prover *PVS* [15] includes a procedure for evaluating ground terms. The PVS ground evaluator essentially consists of a translator from an executable subset of PVS into Common Lisp. The unexecutable fragments are uninterpreted functions, non-bounded quantification and higher-order equalities.

The *Centaur* system [10] is an environment for specifying programming languages. One of its components is a Prolog-style language called *Typol* [6], in which transition rules of natural semantics can be specified. Attali et al [3] have

used Typol to specify a formal, executable semantics of a large subset of the programming language *Java*. Originally, Typol specifications were compiled to Prolog in order to execute them. Recently, Dubois and Gayraud [7] have proposed a translation of Typol specifications to ML. The consistency conditions for modes described in §3.1 are inspired by this paper.

6 Conclusion

We conclude the paper with a survey of the intended applications. Our primary aim has been to validate the rather complex specifications arising in our Java modelling efforts [14]. Such language and machine specifications are an important application area for theorem provers. They have the pleasant property that their execution requires no logical variables: results are not synthesized by unification but computed by evaluation. Hence these specifications meet the requirements imposed by our mode system. No major changes were required to make our Java semantics, an inductive definition, executable. Note that this was possible only because our Java model is based on a first-order abstract syntax. Higher-order abstract syntax requires a richer language of inductive definitions than what we can currently compile.

Of course prototypes can be used to debug all kinds of specifications. One further promising application area that we intend to study is that of cryptographic protocols. Paulson [17] has shown how to specify protocols as inductive definitions and how to verify them. Basin [5] has shown that functional languages are well-suited for prototyping such protocols and for finding bugs. Thus it is highly desirable to have one integrated environment for specification, debugging, and verification, which our prototyping facility could turn Isabelle into.

Finally we intend to pursue *reflection*, i.e. re-importing the result of an external execution into the theorem prover. A simple example would be the efficient evaluation of arithmetic or other expressions.

References

[1] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference (TPHOLs'99)*, volume 1690 of *Lect. Notes in Comp. Sci.*, pages 323–340. Springer-Verlag, 1999.

[2] J. H. Andrews. Executing formal specifications by translation to higher order logic programming. In E. L. Gunter and A. Felty, editors, *10th International Conference on Theorem Proving in Higher Order Logics*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 17–32. Springer-Verlag, 1997.

[3] I. Attali, D. Caromel, and M. Russo. A formal and executable semantics for Java. In *Proceedings of Formal Underpinnings of Java, an OOPSLA'98 Workshop, Vancouver, Canada*, 1998. Technical report, Princeton University.

[4] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq proof assistant reference manual – version 6.3.1. Technical report, INRIA, 1999.

[5] D. Basin. Lazy infinite-state analysis of security protocols. In *Secure Networking — CQRE [Secure] '99*, volume 1740 of *Lect. Notes in Comp. Sci.*, pages 30–42. Springer-Verlag, 1999.

[6] T. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, 1988.

[7] C. Dubois and R. Gayraud. Compilation de la sémantique naturelle vers ML. In *Proceedings of journées francophones des langages applicatifs (JFLA99)*, 1999. Available via http://pauillac.inria.fr/~weis/jfla99/ps/dubois.ps.

[8] W. Goerigk, July 2000. Personal communication.

[9] M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[10] I. Jacobs and L. Rideau-Gallot. A Centaur tutorial. Technical Report 140, INRIA Sophia-Antipolis, July 1992.

[11] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[12] C. S. Mellish. The automatic generation of mode declarations for Prolog programs. Technical Report 163, Department of Artificial Intelligence, University of Edinburgh, August 1981.

[13] T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26, 2001.

[14] D. v. Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 119–156. Springer-Verlag, 1999.

[15] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS System Guide version 2.3. Technical report, SRI International Computer Science Laboratory, Menlo Park CA, September 1999.

[16] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[17] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.

[18] F. Pfenning. Logic programming in the LF Logical Framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 66–78. Cambridge University Press, 1991.

[19] P. S. Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher order Logic Theorem Proving and its Applications*, Leuven, Belgium, September 1992. Elsevier.

[20] K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Institut für Informatik, TU München, 1999.