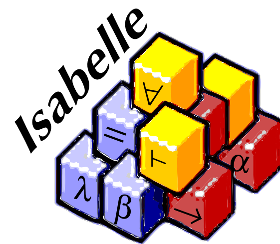
Executing Higher Order Logic

Stefan Berghofer and Tobias Nipkow

Institut für Informatik
TU München



Motivation

- Executing formal specifications: popular research topic for some decades (ACL2, constructive type theory, . . .)
- Essential for validating complex specifications by running test cases ("Have I specified the right thing?")
- Rapid prototyping: generating code automatically
- Problem with HOL specifications:
  - can be highly non-executable
  - various approaches to their execution

Higher Order Logic specifications

... can contain elements of functional programming ...

$$\textbf{datatype}\ \mathsf{nat}\ =\ 0 \mid \mathsf{Suc}\ \mathsf{nat}$$

$$\begin{aligned}&\textbf{primrec}\\ &\mathsf{add}\ 0\ y\ =\ y\\ &\mathsf{add}\ (\mathsf{Suc}\ x)\ y\ =\ \mathsf{Suc}\ (\mathsf{add}\ x\ y)\end{aligned}$$

... and of logic programming

$$\begin{aligned}&\textbf{inductive}\\ &0 \in \mathsf{even}\\ &x \in \mathsf{even} \Longrightarrow \mathsf{Suc}\ (\mathsf{Suc}\ x) \in \mathsf{even}\end{aligned}$$

Possible target languages

Curry execution model: Narrowing

Mercury strong mode system, generates efficient C code

λ-**Prolog** higher order pattern unification

Haskell lazy evaluation, type classes

ML implementation language of Isabelle $\surd$

Design goals

- Not: reach or extend the limits of functional-logic programming
- But: design a lightweight and efficient execution mechanism for HOL that
  - requires only a functional programming language
  - is sufficient for typical applications like
    - execution of programming language semantics
    - abstract machines

An executable subset of HOL

- 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.
    $\wedge$, $\vee$ and $\neg$, `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, t_i executable terms, q_i either p or other executable inductive relation.
- Executable recursive functions, i.e. sets of rewrite rules, lhs contains only constructor patterns with distinct variables, rhs executable term.

Execution

- Execution: finding solutions to queries
- Solution σ: a mapping of variables to closed solution terms
- t is called a solution term iff
  - $t :: \tau \to \tau'$, or
  - $t = c\ t_1\ \ldots\ t_n$, where $n \geq 0$, t_i are solution terms and $c \in \mathcal{C}$

Different kinds of queries

Functional queries $t = X$

t closed executable term, X variable

Result: at most one solution, e.g. solve(add 0 (Suc 0) $= X$) $= \{[X \mapsto$ Suc $0]\}$

Relational queries $(t_1, \ldots, t_n) \in r$

r executable inductively defined relation, t_i closed executable term or variable

Result of query Q: set of solutions solve(Q) $= \{\sigma_1, \sigma_2, \ldots\}$

May be empty, e.g. solve($Suc\ 0 \in$ even) $= \{\}$,

or infinite, e.g. solve($X \in$ even) $= \{[X \mapsto 0], [X \mapsto$ Suc (Suc 0)$], \ldots\}$.

Mode analysis

- Translation of logic into functional program requires analysis of dataflow
- A mode is a specific direction of dataflow
- A mode is a set of indices, denoting positions of input arguments
- Standard tool for analysis / optimization of logic programs

Example

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

Mode $\{1,2\}$ Given two lists $xs = [1,\ 2]$ and $ys = [3,\ 4]$, returns $zs = [1,\ 2,\ 3,\ 4]$

Mode $\{3\}$ Given the list $zs = [1,\ 2,\ 3,\ 4]$, returns a sequence of pairs of lists, i.e.
$xs = [1,\ 2,\ 3,\ 4]$ and $ys = []$, or
$xs = [1,\ 2,\ 3]$ and $ys = [4]$, or
$xs = [1,\ 2]$ and $ys = [3,\ 4]$, etc.

Mode analysis — some notation

Given a set of predicates P, a relation modes is called a mode assignment iff

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

Set of modes assigned to predicate p:

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

args_of M returns tuple of input arguments specified by mode M, e.g.

$$\textsf{args_of}\ \{1,\ 2\}\ (\textsf{Cons}\ x\ xs,\ ys,\ \textsf{Cons}\ x\ zs) = (\textsf{Cons}\ x\ xs,\ ys)$$

vars_of returns all variables occurring in a tuple, e.g.

$$\textsf{vars_of}\ (\textsf{Cons}\ x\ xs,\ ys) = \{x,\ xs,\ ys\}$$

known_args returns indices of all arguments, whose value is fully known, provided the values of the variables given are known, e.g.

$$\textsf{known_args}\ \{x,\ xs,\ ys\}\ (\textsf{Cons}\ x\ xs,\ ys,\ \textsf{Cons}\ x\ zs) = \{1,\ 2\}$$

Consistency of modes

A mode M is called consistent w.r.t. a mode assignment $modes$ and a clause

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

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

$$(1) \quad v_0 = \text{vars_of } (\text{args_of } M\ (t_1,\ \ldots,\ t_k))$$

$$(2) \quad \forall 1 \le i \le m.\ \exists M' \in modes\ q_{\pi(i)}.\ M' \subseteq \text{known_args } v_{i-1}\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})$$

$$(3) \quad \forall 1 \le i \le m.\ v_i = v_{i-1} \cup \text{vars_of } (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})$$

$$(4) \quad \text{vars_of } (t_1,\ \ldots,\ t_k) \subseteq v_m$$

Mode inference

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

Greatest set of allowable modes:

$$\bigcap_i\ \Gamma^i\ \{(p,\ M) \mid p \in P \wedge M \subseteq \{1,\ \ldots,\ \text{arity } p\}\}$$

Translation of inductive relations — basic concepts

- Input: tuple of arguments
  Output: (possibly infinite) sequence of result tuples
- ML translation uses lazy lists:

```
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

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

  write `s1 ++ s2` instead of `Seq.append (s1, s2)`
- Generate separate function for each mode, e.g.

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

  for modes $\{1, 2\}$ and $\{3\}$

Translation of inductive relations — an example

```
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);
```

The general translation scheme

Assume p has clause

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

Translation of p

```
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)
  ++
  ...;
```

Alternative: Haskell translation

Compact notation using list comprehension:

```
append_1_2 inp =
  [ys | ([], ys) <- [inp]] ++
  [x : zs | (x : xs, ys) <- [inp], zs <- append_1_2 (xs, ys)]

append_3 inp =
  [([], ys) | ys <- [inp]] ++
  [(x : xs, ys)  | x : zs <- [inp], (xs, ys) <- append_3 zs]
```

General translation scheme:

p `inp =`
  `[`$opat_0$ `|` $ipat_0$ `<- [inp],` $opat_1$ `<-` q_1 $ipat_1$`,` $\ldots$`,` $opat_m$ `<-` $ipat_m$`]`
  `++`
  $\ldots$;

Mixing relations and functions

β-reduction for λ-terms in de Bruijn notation (taken from Nipkow, JAR 2001):

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) $(k + 1)$)

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

The translation

```
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);
```

Mixing relations and functions — additional mode constraints

Problem:

- non-constructor functions may not be inverted
- equality of functions is undecidable

Therefore: in an input position in the clause head or in an output position in the clause body only constructor terms may occur.

$$(1') \quad v_0 = \mathsf{vars_of}\ (\mathsf{args_of}\ M\ (t_1,\ \ldots,\ t_k)) \wedge \\ \mathsf{funs_of}\ (\mathsf{args_of}\ M\ (t_1,\ \ldots,\ t_k)) \subseteq \mathcal{C}$$

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

Related work

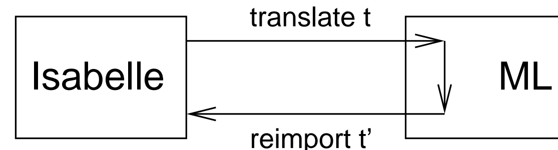
- Other approaches to executing HOL: Rajan (TPHOLs92, covers datatypes and recursive functions, target language: ML), Andrews (TPHOLs97, covers also specifications similar to HOL's inductive definitions, target language λ-Prolog)
- Elf / Twelf
- Aagaard et al, Lifted-fl (TPHOLs99): functional language with theorem prover: execution of fl functions / reasoning about fl functions
- PVS (version 2.3): ground evaluation, translation to Common Lisp
- Coq: based on type theory CIC, uniform treatment of proofs and programs, universes Set and Prop for marking computational content. Can generate code from recursive functions. To get a program from relation $P\ x\ y$, prove $\forall x.\ \exists y.\ P\ x\ y$ constructively and extract program from proof. What to do if P is undecidable?
- Centaur system (INRIA): environment for specifying programming languages, Prolog-style language Typol for specifying transition rules of natural semantics

Conclusion

- A surprisingly large part of HOL is executable
- Translation scheme already tested on larger examples: specification of μJava semantics (inductive definition), JVM specification (recursive function)

Further work

- Tighter integration with Isabelle



- Automatic and provably correct replacement of non-executable elements of specifications (e.g. sets) by executable ones (e.g. lists)
- More expressive mode system supporting higher order modes, for inductive relations such as

  $(x,\ x) \in \text{trancl}\ r$

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

  Mode of trancl r depends on mode of r