Reasoning About CBV Functional Programs in Isabelle/HOL

Randy Pollack and John Longley

Draft paper from
`http://homepages.inf.ed.ac.uk/rap/`

April 12, 2004 (1041)

Project Goals

- Reason about SML programs
  - First, the functional core . . .
  - . . . add exceptions . . .
  - . . . add references and I/O.
  - Modules?
- High level *program logic*
  - Understood by programmers in terms of *operational concepts*.
    - syntactic definability
    - observational equivalence
  - Justified by denotational models.

Overview of Approach ...

- SML embedded in HOL
  - SML types as HOL types.
  - HOL equality on SML types represents observational equivalence.
  - Some problems.
    - Representation not completely faithful to SML.
- Program logic in HOL
  - HOL constructions apply to SML types.
    - E.g. inductively defined relations.
  - Yields expressive logic for *specification* of programs.
  - Example: *sorting* is specified in terms HOL inductive relations *ordered* and *permutation* on SML lists.

. . . Overview of Approach

- Implemented in Isabelle/HOL
  - pragmatic experiments using Isabelle's tools
  - Isabelle typeclass of SML types
  - use Isabelle typechecking to typecheck programs

Draft paper on `http://homepages.inf.ed.ac.uk/rap/`.

Overview of Work Done ...

- Syntax of core language in Isabelle:
  - functions, unit, bool, products, sums.
- Axioms of the core language
  - Evaluation rules.
  - Semantic properties:
    - least fixpoint, SML function extensionality, induction principles, ...
- Extend core with inductive datatypes:
  - natural numbers
    - iterator, order, derived induction principles, Fibonacci
  - polymorphic lists
    - iterator, folding, append, reverse, map, length induction, some theorems from Paulson's SML textbook
    - polymorphic insertion sort

. . . Overview of Work Done

- Streams, a non-well-founded datatype.
  - equivalence of several formulations: initial algebra, final coalgebra, . . .
  - A program to remove all zeros from a stream returns a stream with no zeros.
    - This showed the need for one more core axiom!
- Work towards uniform treatment of positive recursive datatypes.
  - In progress.
- Starting to work on exceptions.

Syntax of the Programming Language: Some core types

```
classes
  SML < type      --{* a class of program types     *}
  SMLeq < SML     --{* a subclass of equality types *}
defaultsort SML


typedecl  UNIT
typedecl  BOOL
typedecl  ('a,'b) "->"       (infixr 80)  --{* functions *}
typedecl  ('a,'b) "**"       (infixr 90)  --{* products *}
arities
  UNIT :: SMLeq
  BOOL :: SMLeq
  "->" :: (SML,SML)SML
  "**" :: (SML,SML)SML
  "**" :: (SMLeq,SMLeq)SMLeq
```

Syntax of the Programming Language: functions and recursion

Usual HOAS encoding of functions:

```
consts  --{* functions and recursion *}
  lam   :: "('a => 'b) => 'a->'b"          (binder "fn " 30)
  APP   :: "['a->'b,'a] => 'b"             (infixl "$" 55)
  Fix   :: "(('a->'b) -> 'a->'b) -> 'a->'b"
```

Syntax of the Programming Language: bottom, unit and bool

```
consts   --{* bottom, unit and bool *}
  bot   :: "'a::SML"                  --{* polymorphic bottom *}
  UN    :: UNIT                                  ("<>")
  tt    :: BOOL
  ff    :: BOOL
  EQ    :: "('a::SMLeq) -> 'a -> BOOL"
  BCASE :: "['a,'a] => (BOOL ->'a)"       --{* non-strict *}

syntax  --{* some syntactic sugar *}
  IF    :: "[BOOL, 'a, 'a] => 'a"
  "[=]" :: "[('a::SMLeq), 'a] => BOOL"      (infixl 34)
translations
  "IF b x y" == "(BCASE x y) $ b"
  "x[=]y"    == "EQ $ x $ y"
```

Remarks on the Representation of Syntax

- We use Isabelle typechecking.
  - Convenient for both developers and users . . .
  - **but not completely faithful to SML.** E.g. no *let polymorphism*.
- **Too many expressions in SML types.**
  - Consider the following function which claims to swap `bot` and `<>`.

    ```
    UNIT_swap :: UNIT -> UNIT == fn x. (THE y. y ~= x)
    ```
  - Not definable in SML; `THE` should not be allowed under a lambda binding a variable of SML type.
  - A variation on HOL makes the embedding sound.
- Other minor problems.

Pragmatic experiment, not faithful representation of official SML.

Program Logic: Basic relations

- Equality of programs is intended to mean *observational equivalence*.
- *Definedness* means different from bottom

```
"dfd x == x ~= bot"
"udfd x == ~(dfd x)"
```

Axioms for evaluation: functions, UNIT, BOOL

```
      --{* strategy is CBV *}
Bot_ap[simp]:     "udfd (bot $ x)"
ap_Bot[simp]:     "udfd (f $ bot)"             --{* strict *}

      --{* function types *}
fn_dfd[simp]:     "dfd (lam F)"
beta_rule[simp]: "dfd x ==> (lam F) $ x = F x" --{* CBV *}
Fix_rule:         "Fix = (fn F x. F $ (Fix $ F) $ x)"

      --{* UNIT and BOOL *}
dfd_unit[simp]:  "dfd <>"
dfd_BCASE[simp]: "dfd (BCASE f g)"
if_true[simp]:    "IF tt x y = x"
if_false[simp]:  "IF ff x y = y"
eq_dfd[simp]:     "[| dfd x; dfd y |] ==> dfd (x [=] y)"
```

Axioms: Semantic properties

- SML extensionality is the *context lemma*:

```
fn_ext: "[| dfd f; dfd g; !!x. dfd x ==> f$x = g$x |]
                                              ==> f = g"
```

- Boolean-valued equality is reflected:

```
eq_reflection: "((x [=] y) = tt)  =  (dfd x & x = y)"
```

- Induction principles:

```
unit_Induct:     "[| P <>; dfd x |] ==> P x"
boolInduct:      "[| P tt; P ff; dfd x |] ==> P x"
```

- `Fix` computes the *least fixpoint*:

```
Fix_min: "[| dfd F; dfd (C (Fix $ F)) |] ==>
               EX (k::nat). dfd (C (mlIter k bot F))"
```

  - If `(Fix $ F)` is defined in some context, then some finite unfolding is defined.
  - We iterate over HOL naturals.

Inductive Datatypes: `NAT` and `'a LIST`

- Similar to `BOOL`
  - Declare the type.
  - Constants for constructors and case eliminator.
  - Axioms for definedness of constants and for evaluation.
  - Axiom for induction principle.
- Interesting relations can be defined using HOL inductive definition. E.g. order on `NAT`:

```
inductive NATLT intros
NATLT_Z: "dfd x ==> ZZ [<] SS $ x"
NATLT_S: "m [<] n ==> SS $ m [<]  SS $ n"
```

More convenient for specification and reasoning than a `BOOL`ean valued program.

`NAT` and `'a LIST`: Some examples

- A fast Fibonacci function satisfies the fibonacci recursion equations, so it is correct.
- Basic list functions (map, append, flatten and reverse) defined.
  - Basic properties follow easily: map distributes over composition, append is associative, reverse is involutive, ...
- Using length induction, we prove some theorems from Paulson's SML texbook.
- We prove a polymorphic insertion sort to be correct.
  - Correctness is specified using HOL inductive relations of *ordered* and *permutation*.

Reasoning about well-founded SML datatypes is similar to reasoning about inductive types in Type Theory of total functions.

(Positive) Recursive Datatypes: Streams

- What about datatypes that lack the usual induction axiom, e.g.

```
datatype 'a SEQ = SCONS of 'a * (unit -> 'a SEQ)
```

- We use the well known characterization that (`'a SEQ`, `SCONS`) is the initial algebra of the functor

```
ST X = 'a ** (UNIT -> X).
```

- From this we define the inverse of `SCONS`, and the SML case operator.
- Now, standard stream functions can be defined: `shd`, `stl`, `snth`, `sdrop`, `sTAKE`.
- However we cannot yet prove *stream extensionality*

```
(ALL n. snth $ n $ s = snth $ n $ t) ==> s = t
```

  which is operationally true.

Time for one more general purpose axiom.

A Final General Axiom

- Reflecting observational equivalence:

```
(ALL (C::'a=>UNIT). C x = C y) ==> x = y
```

- Several equivalent formulations involving observational order or observational limits.

Back to Stream Extensionality

- Stream extensionality follows from this axiom.
- Consider a program, `rmZZs`, that removes all zeros from a `NAT SEQ`. Using stream extensionality we can prove

```
lemma "snth $ n $ (rmZZs $ s) ~= ZZ"
```

Proved

- Equivalence of our presentation with Pitts' coinduction characterization.

In Progress

- Equivalence with final coalgebra property.
- Uniform treatment of all positive recursive datatypes.
  - Must derive structural induction for the well-founded datatypes.