Higher-Order Functions and Recursive Types in PVS

Higher-Order functions and Recursive Types in PVS

Sam Owre

owre@csl.sri.com

URL: http://www.csl.sri.com/~owre/

Computer Science Laboratory

SRI International

Menlo Park, CA

Higher Order Logic

Overview

- Variables and quantification in first-order logic range over ordinary datatypes such as numbers, and functions and predicates are fixed (constants).

- Higher order logic allows variables to range over functions and predicates as well.

- Higher order logic requires strong typing for consistency, otherwise, we could define $R(x) = \neg x(x)$, and derive $R(R) = \neg R(R)$.

- Higher order logic can express a number of interesting concepts and datatypes that are not expressible within first-order logic.

Higher Order Summation

```
hsummation : THEORY

  BEGIN
  n: VAR nat
  f : VAR [nat -> nat]

  hsum(f)(n): RECURSIVE nat =
   (IF n = 0 THEN f(0) ELSE f(n-1) + hsum(f)(n - 1) ENDIF)
  MEASURE n

  hsum_id: LEMMA hsum(id)(n+1) = (n * (n+1))/2
  ⋮
```

`hsum_id` proved by `induct-and-simplify`.

Variations on Summation

```
square(n): nat = n*n

sum_of_squares: LEMMA
  6 * hsum(square)(n+1) = n * (n + 1) * (2*n + 1)

cube(n): nat = n*n*n

sum_of_cubes: LEMMA
  4 * hsum(cube)(n+1) = n*n*(n+1)*(n+1)

END hsummation
```

Both lemmas proved by `induct-and-simplify`.

Parametric Summation

Theory parameters can also be used for schematic definition.

```
psummation  [f : [nat -> nat] ] : THEORY
  BEGIN

  n: VAR nat

  psum(n): RECURSIVE nat =
   (IF n = 0 THEN f(0) ELSE f(n-1) + psum(n - 1) ENDIF)
  MEASURE n

  END psummation
```

Using Parametric Summation

The parametric theory can be imported either with specific parameters or generically.

```
check_psummation: THEORY
  BEGIN
   IMPORTING psummation
   n : VAR nat

   check: LEMMA psum[id[nat]](n + 1) = (n * (n + 1))/2

  END check_psummation
```

`check` proved by `induct-and-simplify`.

Induction in Higher Order Logic

```
p: VAR [nat -> bool]

nat_induction: LEMMA
  (p(0) AND (FORALL j: p(j) IMPLIES p(j+1)))
      IMPLIES (FORALL i: p(i))
```

`nat_induction` is derived from well-founded induction, as are other variants like structural recursion, measure induction.

Higher-Order Specification: Functions

```
functions [D, R: TYPE]: THEORY
 BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D

  extensionality_postulate: POSTULATE
     (FORALL (x: D): f(x) = g(x)) IFF f = g
  congruence: POSTULATE f = g AND x1 = x2 IMPLIES f(x1) = g(x2)
  eta: LEMMA (LAMBDA (x: D): f(x)) = f

  injective?(f): bool =
     (FORALL x1, x2: (f(x1) = f(x2) => (x1 = x2)))
  surjective?(f): bool = (FORALL y: (EXISTS x: f(x) = y))
  bijective?(f): bool = injective?(f) & surjective?(f)
  ⋮
 END functions
```

Sets are Predicates

```
sets [T: TYPE]: THEORY
 BEGIN
  set: TYPE = [t -> bool]
  x, y: VAR T
  a, b, c: VAR set

  member(x, a): bool = a(x)

  empty?(a): bool = (FORALL x: NOT member(x, a))

  emptyset: set = {x | false}

  subset?(a, b): bool = (FORALL x: member(x, a) => member(x, b))

  union(a, b): set = {x | member(x, a) OR member(x, b)}
  ⋮
 END sets
```

Useful Higher Order Datatypes: Finite Sets

Finite sets: Predicate subtypes of sets that have an injective map to some initial segment of nat.

```
finite_sets_def[T: TYPE]: THEORY
 BEGIN
  x, y, z: VAR T
  S: VAR set[T]
  N: VAR nat

  is_finite(S): bool = (EXISTS N, (f: [(S) -> below[N]]):
                            injective?(f))

  finite_set: TYPE = (is_finite) CONTAINING emptyset[T]
  ⋮
END finite_sets_def
```

Recursive Datatypes

Overview

- Recursive datatypes like lists, stacks, queues, binary trees, and abstract syntax trees, are commonly used in specification.
- Manual axiomatizations for datatypes can be error-prone.
- Verification systems should (and many do) automatically generate datatype theories.
- The PVS `DATATYPE` construct introduces recursive datatypes that are *freely generated* by given constructors, *including* lists, binary trees, abstract syntax trees, but *excluding* bags and queues.
- The PVS proof checker automates various datatype simplifications.

The `list` Datatype

The type `list` is parametric in its element type `T`.

There are two *constructors* `null` and `cons` with corresponding *recognizers* `null?` and `cons?`.

`cons` has two fields corresponding to the accessors `car` of type `T` and `cdr` which is recursively of type `list[T]`.

```
list[T: TYPE] : DATATYPE
 BEGIN
  null: null?
  cons (car: T, cdr: list): cons?
 END list
```

Binary Trees

Parametic in value type `T`.

Constructors: `leaf` and `node`.

Recognizers: `leaf?` and `node?`.

`node` accessors: `val`, `left`, and `right`.

```
binary_tree[T: TYPE] : DATATYPE
BEGIN
  leaf: leaf?
  node(val: T, left: binary_tree, right: binary_tree): node?
END binary_tree
```

Theories Axiomatizing Binary Trees

The `binary_tree` declaration generates three theories axiomatizing the binary tree data structure:

- `binary_tree_adt`: Declares the constructors, accessors, and recognizers, and contains the basic axioms for extensionality and induction, and some basic operators.
- `binary_tree_adt_map`: Defines `map` operations over the datatype.
- `binary_tree_adt_reduce`: Defines a recursion scheme over the datatype.

Datatype axioms are already built into the relevant proof rules, but the defined operations are useful.

```
binary_tree_adt[T: TYPE]: THEORY
  BEGIN
  binary_tree: TYPE
  leaf?, node?: [binary_tree -> boolean]
  leaf: (leaf?)
  node: [[T, binary_tree, binary_tree] -> (node?)]
  val: [(node?) -> T]
  left: [(node?) -> binary_tree]
  right: [(node?) -> binary_tree]
    ⋮
  END binary_tree_adt
```

Predicate subtyping is used to precisely type constructor terms and avoid misapplied accessors.

An Extensionality Axiom per Constructor

Extensionality states that a node is uniquely determined by its accessor fields.

```
binary_tree_node_extensionality: AXIOM
  (FORALL (node?_var: (node?)),
          (node?_var2: (node?)):
     val(node?_var) = val(node?_var2)
       AND left(node?_var) = left(node?_var2)
         AND right(node?_var) = right(node?_var2)
       IMPLIES node?_var = node?_var2)
```

Accessor/Constructor Axioms

Asserts that `val(node(v, A, B)) = v`.

```
binary_tree_val_node: AXIOM
  (FORALL (node1_var: T), (node2_var: binary_tree),
          (node3_var: binary_tree):
     val(node(node1_var, node2_var, node3_var)) = node1_var)
```

An Induction Axiom

Conclude `FORALL A: p(A)` from `p(leaf)` and
$\mathtt{p}(\mathtt{A}) \wedge \mathtt{p}(\mathtt{B}) \supset \mathtt{p}(\mathtt{node}(\mathtt{v}, \mathtt{A}, \mathtt{B}))$.

```
binary_tree_induction: AXIOM
  (FORALL (p: [binary_tree -> boolean]):
     p(leaf)
       AND
       (FORALL (node1_var: T), (node2_var: binary_tree),
               (node3_var: binary_tree):
             p(node2_var) AND p(node3_var)
            IMPLIES p(node(node1_var, node2_var, node3_var)))
       IMPLIES (FORALL (binary_tree_var: binary_tree):
                  p(binary_tree_var)))
```

Pattern-matching Branching

The `CASES` construct is used to branch on the outermost constructor of a datatype expression.

We implicitly assume the disjointness of `(node?)` and `(leaf?)`:

```
CASES leaf OF                          =  u
      leaf : u,
      node(a, y, z) : v(a, y, z)
      ENDCASES

CASES node(b, w, x) OF                 =  v(b, w, x)
      leaf : u,
      node(a, y, z) : v(a, y, z)
      ENDCASES
```

Useful Generated Combinators

```
reduce_nat(leaf?_fun:nat, node?_fun:[[T, nat, nat] -> nat]):
  [binary_tree -> nat] = ...
```

```
every(p: PRED[T])(a: binary_tree):  boolean = ...
```

```
some(p: PRED[T])(a: binary_tree):  boolean = ...
```

```
subterm(x, y: binary_tree):  boolean = ...
```

```
map(f: [T -> T1])(a: binary_tree[T]):  binary_tree[T1] = ...
```

Ordered Binary Trees

Ordered binary trees can be introduced by a theory that is parametric in the value type as well as the ordering relation.

The ordering relation is subtyped to be a total order.

```
total_order?(<=): bool = partial_order?(<=) & dichotomous?(<=)
```

```
obt [T : TYPE,  <= : (total_order?[T])] : THEORY
BEGIN
IMPORTING binary_tree[T]

A, B, C: VAR binary_tree
x, y, z: VAR T
pp: VAR pred[T]
i, j, k: VAR nat
  ⋮
END obt
```

The `size` Function

The number of nodes in a binary tree can be computed by the `size` function which is defined using `reduce_nat`.

```
size(A) : nat =
  reduce_nat(0, (LAMBDA x, i, j: i + j + 1))(A)
```

The Ordering Predicate

Recursively checks that the left and right subtrees are ordered, and that the left (right) subtree values lie below (above) the root value.

```
ordered?(A) : RECURSIVE bool =
  (IF node?(A)
   THEN (every((LAMBDA y: y<=val(A)), left(A)) AND
         every((LAMBDA y: val(A)<=y), right(A)) AND
         ordered?(left(A)) AND
         ordered?(right(A)))
   ELSE TRUE
   ENDIF)
  MEASURE size
```

Insertion

Compares x against root value and recursively inserts into the left or right subtree.

```
insert(x, A): RECURSIVE binary_tree[T] =
  (CASES A OF
    leaf: node(x, leaf, leaf),
    node(y, B, C): (IF x<=y THEN node(y, insert(x, B), C)
                            ELSE node(y, B, insert(x, C))
                    ENDIF)
   ENDCASES)
  MEASURE (LAMBDA x, A: size(A))
```

Insertion Property

The following is a very simple property of `insert`.

```
ordered?_insert_step: LEMMA
   pp(x) AND every(pp, A) IMPLIES every(pp, insert(x, A))
```

Proved by `induct-and-simplify`

Orderedness of `insert`

```
ordered?_insert: THEOREM
   ordered?(A) IMPLIES ordered?(insert(x, A))
```

is proved by the 4-step PVS proof

```
(""
 (induct-and-simplify "A" :rewrites "ordered?_insert_step")
 (rewrite "ordered?_insert_step")
 (typepred "obt.<=")
 (grind :if-match all))
```

Mutually Recursive Datatypes

PVS does not directly support mutually recursive datatypes.

These can be defined as subdatatypes (e.g., `term`, `expr`) of a single datatype.

```
arith: DATATYPE WITH SUBTYPES expr, term
 BEGIN
  num(n:int): num?              :term
  sum(t1:term,t2:term):   sum?  :term
% ...
  eq(t1: term, t2: term): eq?   :expr
  ift(e: expr, t1: term, t2: term): ift? :term
% ...
 END arith
```

Summary

- The PVS datatype mechanism succinctly captures a large class of useful datatypes by exploiting predicate subtypes and higher-order types.
- Datatype simplifications are built into the primitive inference mechanisms of PVS.
- This makes it possible to define powerful and flexible high-level strategies.
- The PVS datatype is loosely inspired by the Boyer-Moore Shell principle.
- Other systems HOL [Melham89, Gunter93] and Isabelle [Paulson] have similar datatype mechanisms as a provably conservative extension of the base logic.