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

\[
\text{square}(n) : \text{nat} = n \times n
\]

\[
\text{sum_of_squares: LEMMA} \\
6 \times \text{hsum}(\text{square})(n+1) = n \times (n + 1) \times (2n + 1)
\]

\[
\text{cube}(n) : \text{nat} = n \times n \times n
\]

\[
\text{sum_of_cubes: LEMMA} \\
4 \times \text{hsum}(\text{cube})(n+1) = n \times n \times (n+1) \times (n+1)
\]

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.

```plaintext
definite_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 definite_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

Parametric in value type $T$.

Constructors: leaf and node.

Recognizers: leaf? and node?.

Node accessors: val, left, and right.

```plaintext
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.

\[
\text{binary_tree_node_extensionality: AXIOM} \\
\text{(FORALL (node?\_var: (node?)),} \\
\text{ (node?\_var2: (node?)):} \\
\text{ val(node?\_var) = val(node?\_var2) } \\
\text{ AND left(node?\_var) = left(node?\_var2) } \\
\text{ AND right(node?\_var) = right(node?\_var2) } \\
\text{ IMPLIES node?\_var = node?\_var2)}
\]
Accessor/Constructor Axioms

Asserts that \( \text{val}(\text{node}(v, A, B)) = v \).

\[
\text{binary_tree_val_node: AXIOM}
\begin{align*}
\text{(FORALL (node1_var: T), (node2_var: binary_tree),} \\
\text{(node3_var: binary_tree):} \\
\text{val(node(node1_var, node2_var, node3_var))) = node1_var)}
\end{align*}
\]
An Induction Axiom

Conclude \( \forall A: p(A) \) from \( p(\text{leaf}) \) and \( p(A) \land p(B) \supset p(\text{node}(v, A, B)) \).

```
binary_tree_induction: AXIOM
    (\forall (p: [binary_tree \to boolean]):
        p(\text{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?):

\[
\text{CASES leaf OF}
\]

\[
\begin{align*}
\text{leaf} & : u, \\
\text{node}(a, y, z) & : v(a, y, z)
\end{align*}
\]

\[
\text{ENDCASES}
\]

\[
\text{CASES node(b, w, x) OF}
\]

\[
\begin{align*}
\text{leaf} & : u, \\
\text{node}(a, y, z) & : v(a, y, z)
\end{align*}
\]

\[
\text{ENDCASES}
\]

= u

= v(b, w, x)
**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.

\[
\text{total\_order?(\leq)} : \text{bool} = \text{partial\_order?(\leq)} \land \text{dichotomous?(\leq)}
\]

```
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`.

\[
\text{size}(A) : \text{nat} = \text{reduce}_\text{nat}(0, (\text{LAMBDA } x, i, j: i + j + 1))(A)
\]
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.

\[
\text{ordered}(A) : \text{RECURSIVE bool} = \\
(\text{IF node}(A) \\
\text{THEN (every((}\lambda y: y <= \text{val}(A)), \text{left}(A)) \text{ AND} \\
\quad \text{every((}\lambda y: \text{val}(A) <= y), \text{right}(A)) \text{ AND} \\
\quad \text{ordered}(\text{left}(A)) \text{ AND} \\
\quad \text{ordered}(\text{right}(A))) \\
\text{ELSE TRUE} \\
\text{ENDIF}) \\
\text{MEASURE size}
\]
Insertion

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

```plaintext
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.

\[
\text{ordered?}_\text{insert\_step}: \text{LEMMA} \\
\quad \text{pp}(x) \text{ AND every}(\text{pp}, A) \implies \text{every}(\text{pp}, \text{insert}(x, A))
\]

Proved by induct-and-simplify
Orderedness of insert

ordered?_insert: THEOREM
    ordered?(A) IMPLICATIONS 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.\leq")
  (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.