Datatypes in PVS[a]

N. Shankar
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory
SRI International
Menlo Park, CA

[a] Joint work with Sam Owre. Funded by NASA Contract NAS1-18969 and SRI.

Overview

- Recursive datatypes like lists, stacks, queues, binary trees, leaf trees, and abstract syntax trees, are commonly used in specification.
- Manual axiomatizations for datatypes can be error-prone.
- Verification system 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[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
```


An Extensionality Axiom per Constructor

```
  binary_tree_leaf_extensionality: AXIOM
        (FORALL (leaf?_var: (leaf?), leaf?_var2: (leaf?)):
           leaf?_var = leaf?_var2);
```

```
  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 two axioms (which imply 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
```


A Recursion Combinator

```
reduce_nat(leaf?_fun:nat, node?_fun:[[T, nat, nat] -> nat]):
    [binary_tree -> nat] =
  LAMBDA (binary_tree_adtvar: binary_tree):
    CASES binary_tree_adtvar OF
      leaf: leaf?_fun,
      node(node1_var, node2_var, node3_var):
          node?_fun(node1_var,
                    reduce_nat(leaf?_fun,
                               node?_fun)
                      (node2_var),
                    reduce_nat(leaf?_fun,
                               node?_fun)
                      (node3_var))
      ENDCASES;
```


A Generic Recursion Combinator

```
binary_tree_adt_reduce[T: TYPE, range: TYPE]: THEORY
  BEGIN

  IMPORTING binary_tree_adt[T]

  reduce(leaf?_fun: range, node?_fun: [[T, range, range] -> range]):
      [binary_tree[T] -> range] =
    LAMBDA (binary_tree_var: binary_tree[T]):
      CASES binary_tree_var OF
        leaf: leaf?_fun,
        node(node1_var, node2_var, node3_var):
            node?_fun(node1_var,
                      reduce(leaf?_fun,
                             node?_fun)(node2_var),
                      reduce(leaf?_fun,
                             node?_fun)(node3_var))
        ENDCASES
  END binary_tree_adt_reduce
```


The Subterm Ordering

The subterm relation checks whether one object occurs as a (not necessarily proper) subterm of another object.

```
subterm(x: binary_tree, y: binary_tree): boolean =
  x = y
    OR CASES y OF
         leaf: FALSE,
         node(node1_var, node2_var, node3_var):
             subterm(x, node2_var) OR subterm(x, node3_var)
         ENDCASES
```

The proper subterm relation << can be easily defined using subterm.

Other Useful Combinators: `every` and `some`

```
every(p: PRED[T])(a: binary_tree): boolean =
  CASES a OF
    leaf: TRUE,
    node(node1_var, node2_var, node3_var):
        p(node1_var)
          AND every(p)(node2_var) AND every(p)(node3_var)
    ENDCASES
```

```
some(p: PRED[T])(a: binary_tree): boolean =
  CASES a OF
    leaf: FALSE,
    node(node1_var, node2_var, node3_var):
        p(node1_var)
          OR some(p)(node2_var) OR some(p)(node3_var)
    ENDCASES
```


A `map` combinator

```
binary_tree_adt_map[T: TYPE, T1: TYPE]: THEORY
  BEGIN

  IMPORTING binary_tree_adt

  map(f: [T -> T1])(a: binary_tree[T]): binary_tree[T1] =
    CASES a OF
      leaf: leaf[T1],
      node(node1_var, node2_var, node3_var):
          node[T1](f(node1_var),
                   map(f)(node2_var), map(f)(node3_var))
      ENDCASES

  END binary_tree_adt_map
```


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.

```
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))
```

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))
```


Proof of `insert` property

```
ordered?_insert_step :
  |-------
{1}   (FORALL (A: binary_tree[T], pp: pred[T], x: T):
         pp(x) AND every(pp, A) IMPLIES every(pp, insert(x, A)))

Rule? (induct-and-simplify "A")
every rewrites every(pp!1, leaf)
  to TRUE
insert rewrites insert(x!1, leaf)
  to node(x!1, leaf, leaf)
every rewrites every(pp!1, node(x!1, leaf, leaf))
  to TRUE
   ⋮
By induction on A, and by repeatedly rewriting and simplifying,
Q.E.D.
Run time  = 16.63 secs.
Real time = 38.10 secs.
```


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))
```

Automated Datatype Simplifications

```
binary_props[T : TYPE] : THEORY
  BEGIN
  IMPORTING binary_tree_adt[T]
  A, B, C, D: VAR binary_tree[T]
  x, y, z: VAR T
  leaf_leaf:  LEMMA leaf?(leaf)
  node_node:  LEMMA node?(node(x, B, C))
  leaf_leaf1: LEMMA A = leaf IMPLIES leaf?(A)
  node_node1: LEMMA A = node(x, B, C) IMPLIES node?(A)
  val_node:   LEMMA val(node(x, B, C)) = x
  leaf_node:  LEMMA NOT (leaf?(A) AND node?(A))
  node_leaf:  LEMMA leaf?(A) OR node?(A)
  leaf_ext:   LEMMA (FORALL (A, B: (leaf?)): A = B)
  node_ext:   LEMMA
     (FORALL (A : (node?)) : node(val(A), left(A), right(A)) = A)
  END binary_props
```


Inline Datatypes

```
combinators  : THEORY
  BEGIN
  combinators: DATATYPE
       BEGIN
         K: K?
         S: S?
         app(operator, operand: combinators): app?
       END combinators

  x, y, z: VAR combinators

  reduces_to: PRED[[combinators, combinators]]

  K: AXIOM reduces_to(app(app(K, x), y), x)
  S: AXIOM reduces_to(app(app(app(S, x), y), z),
                     app(app(x, z), app(y, z)))
  END combinators
```


Scalar Datatypes

```
  colors: DATATYPE
    BEGIN
      red: red?
      white: white?
      blue: blue?
    END colors
```

The above verbose inline declaration can be abbreviated as:

```
  colors: TYPE = {red, white, blue}
```


Disjoint Unions

```
disj_union[A, B: TYPE] : DATATYPE
  BEGIN
    inl(left : A): inl?
    inr(right : B): inr?
  END disj_union
```

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
```


Operations on Mutually Recursive Datatypes

```
arith_eval: THEORY
 BEGIN
  IMPORTING arith
  value: DATATYPE
     BEGIN
      bool(b:bool): bool?
      int(i:int): int?
     END value
  eval(a: arith): RECURSIVE
         {v: value | IF expr(a) THEN bool?(v) ELSE int?(v) ENDIF} =
   CASES a OF
    num(n):         int(n),
    sum(n1, n2):    int(i(eval(n1)) + i(eval(n2))),
    eq(n1, n2):     bool(i(eval(n1)) = i(eval(n2))),
    ift(e, n1, n2): IF b(eval(e)) THEN eval(n1) ELSE eval(n2) ENDIF
   ENDCASES
   MEASURE a BY <<
 END arith_eval
```


Recursive Ordinals

- ϵ_0 is the least ordinal x such that $x = \omega^x$.
- It includes $0, 1, \ldots, \omega, \omega{+}1, \ldots, \omega{+}\omega, \ldots, 3{*}\omega, \ldots, \omega^2, \ldots, \omega^\omega, \ldots, \omega^{\omega^{\cdot^{\cdot^{\omega}}}}, \ldots$
- Sub-ϵ_0 ordinals can be represented using the Cantor normal form with ordinals $\alpha_1, \ldots, \alpha_n$ such that $\alpha_1 \leq \ldots \leq \alpha_n < \alpha$, and natural numbers $c_1, \ldots, c_n$ such that $\alpha = c_1 * \omega^{\alpha_1} + c_2 * \omega^{\alpha_2} + \ldots + c_n * \omega^{\alpha_n}$.
- The Cantor normal form can be exploited to build a data structure representing sub-ϵ_0 ordinals and to define a primitive recursive ordering relation.

Recursive Ordinals and Lexicographic Orderings

- A lexicographic order on a pair $\langle m, n \rangle$ can be given by $n * \omega^0 + m * \omega$ which is just $n + m * \omega$.
- The sub-ϵ_0 ordinals includes all nested lexicographic orderings.

Recursive Ordinals

The `ordstruct` datatype that represent ordinal-like structures.

```
ordstruct: DATATYPE
BEGIN
 zero: zero?
 add(coef: posnat, exp: ordstruct, rest: ordstruct): nonzero?
END ordstruct
```

What is missing is the assertion that the exponents appear in non-increasing order.

The `ordstruct` `zero` represents 0.

The `ordstruct` `add(c, alpha, beta)` represents $c * \omega^{ord(\texttt{alpha})} + ord(\texttt{beta})$.

Ordering Recursive Ordinals

```
ordinals: THEORY
 BEGIN
  i, j, k: VAR posnat
  m, n, o: VAR nat
  u, v, w, x, y, z: VAR ordstruct
  size: [ordstruct->nat] =
     reduce[nat](0, (LAMBDA i, m, n: 1 + m+n));
  <(x, y): RECURSIVE bool =
     CASES x OF
        zero: NOT zero?(y),
        add(i, u, v): CASES y OF
                        zero: FALSE,
                        add(j, z, w): (u<z) OR
                                      (u=z) AND (i<j) OR
                                      (u=z) AND (i=j) AND (v<w)
                      ENDCASES
     ENDCASES
   MEASURE size(x);
```


Normalized Ordinal Representation

`add(i, u, v) < add(j, z, w)` if either `u < z`, or `u` $\equiv$ `z` and either `i < j`, or `i = j` and `v < w`.

The < relation is a total ordering on normalized ordinals.

```
ordinal?(x): RECURSIVE bool =
  CASES x OF
    zero: TRUE,
    add(i, u, v): (ordinal?(u) AND ordinal?(v) AND
                    CASES v OF
                      zero: TRUE,
                      add(k, r, s): r < u
                    ENDCASES)
  ENDCASES
 MEASURE size

ordinal: NONEMPTY_TYPE = (ordinal?)
```


The Lexicographic Ordering

```
lex2(m, n): ordinal =
  (IF m=0
     THEN IF n = 0
             THEN zero
             ELSE add(n, zero, zero)
          ENDIF
     ELSIF n = 0 THEN add(m, add(1,zero,zero),zero)
     ELSE add(m, add(1,zero,zero), add(n,zero, zero))
   ENDIF)

lex2_lt: LEMMA
  lex2(i, j) < lex2(m, n) =
   (i < m OR (i = m AND j < n))
```

Admitting Multiple Recursion

The classic Ackermann function can be admitted with a lexicographic measure constructed using `lex2`.

```
ackermann: THEORY
BEGIN
i, j, k, m, n: VAR nat

ack(m,n): RECURSIVE nat =
  (IF m=0 THEN n+1
          ELSIF n=0 THEN ack(m-1,1)
                    ELSE ack(m-1, ack(m, n-1))
        ENDIF)
  MEASURE lex2(m, n)

   ⋮
END ackermann
```


Ackermann Termination TCCs

```
% Termination TCC generated (line 9) for ack
ack_TCC2: OBLIGATION
      (FORALL (m, n): NOT m = 0 AND n = 0
         IMPLIES lex2(m - 1, 1) < lex2(m, n));

% Termination TCC generated (line 10) for ack
ack_TCC5: OBLIGATION
      (FORALL (m, n):
         NOT m = 0 AND NOT n = 0
        IMPLIES lex2(m, n - 1) < lex2(m, n));

% Termination TCC generated (line 10) for ack
ack_TCC6: OBLIGATION
      (FORALL (v: [[nat, naturalnumber] -> nat], m, n):
         NOT m = 0 AND NOT n = 0
             IMPLIES lex2(m - 1, v(m, n - 1)) < lex2(m, n));
```


Issues in Datatypes

- Can subtyping on *positive* datatype parameters be lifted to the datatypes? E.g., `binary_tree[nat]` should be a subtype of `binary_tree[int]`.
- Is there a larger class of datatypes for which axioms can be generated in a systematic manner?
- Combining explicit and implicit induction: PVS datatypes introduce *explicit* induction schemes. *Implicit* methods (e.g., SPIKE, Focus) use term orderings to perform simultaneous induction on mutually recursive definitions.
- Datatype refinement. More generally, theory refinement (e.g., IMPS, Ehdm).

Conclusions

- The PVS datatype mechanism succinctly captures a large class of useful datatypes.
- The 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.
- HOL [Melham89, Gunter93] has a similar datatype mechanism as a provably conservative extension of the base logic.