Abstract Datatypes[1]

Alwyn E. Goodloe

NASA Langley Formal Methods Group

a.goodloe@nasa.gov

11 October 2012

[1] Material in this lecture derived from NASA/CR-97-206264 *Abstract Datatypes in PVS*, by Sam Owre and Natarajan Shankar, November 1997.

Abstract Datatype (ADT) Uses in PVS

- Recursive Types
  - Lists
  - Stacks
  - Trees
  - Syntax
  - ...
- Enumerated Types
- Disjoint Unions (case-variant records)

ADT Syntax

```
<name>[<type parameters>]: DATATYPE
  BEGIN
   <constructor> : <recognizer>
   . . .
   <constructor>(<accessor>:<type>, ...):<recognizer>
  END <name>
```

- Constructor and accessor names must be disjoint
- If `<name>` used in type of accessor, it must occur positively
- Declaration is not contained in a PVS theory (an alternate form may be used)
- PVS automatically generates file `<name>_adt.pvs`

Inline declaration

```
<theory>[<theory parameters>]: THEORY
 BEGIN

   ...

  <name>: DATATYPE
    BEGIN
     <constructor> : <recognizer>
     . . .
     <constructor>(<accessor>:<type>, ...):<recognizer>
    END <name>

   ...
```

- No type parameters allowed if declared in a theory
- PVS does not generate `<name>_adt.pvs` file
- However, the theory is implicitly available

Example ADT: Stacks

```
stack[T:TYPE]: DATATYPE
 BEGIN
  empty: empty?
  push(top:T, pop:stack) : non_empty?
 END stack
```

- Constructors: empty, push
- Accessors: top, pop
- Recognizers: empty?, non_empty?

Automatically generated facts

```
%%% ADT file generated from stacks

stack_adt[T: TYPE]: THEORY
  BEGIN

  stack: TYPE

  empty?, non_empty?: [stack -> boolean]

  empty: (empty?)

  push: [[T, stack] -> (non_empty?)]

  top: [(non_empty?) -> T]

  pop: [(non_empty?) -> stack]
```

Definition by cases

Each ADT allows a construct for definition by cases, allowing a form a pattern matching on datatype constructors.

```
ord(x: stack): upto(1) =
  CASES x OF
   empty: 0,
   push(push1_var, push2_var): 1
  ENDCASES
```

The cases construct is implicitly axiomatized to ensure that the constructors are disjoint.

Extensionality Axioms

```
stack_empty_extensionality: AXIOM
      (FORALL (empty?_var: (empty?),
               empty?_var2: (empty?)):
         empty?_var = empty?_var2);

stack_push_extensionality: AXIOM
      (FORALL (non_empty?_var: (non_empty?),
               non_empty?_var2: (non_empty?)):
         top(non_empty?_var) = top(non_empty?_var2)
             AND pop(non_empty?_var) = pop(non_empty?_var2)
             IMPLIES non_empty?_var = non_empty?_var2);

stack_push_eta: AXIOM
      (FORALL (non_empty?_var: (non_empty?)):
         push(top(non_empty?_var), pop(non_empty?_var))
           = non_empty?_var);
```

Accessor-Constructor Axioms

```
stack_top_push: AXIOM
      (FORALL (push1_var: T, push2_var: stack):
         top(push(push1_var, push2_var)) = push1_var);

stack_pop_push: AXIOM
      (FORALL (push1_var: T, push2_var: stack):
         pop(push(push1_var, push2_var)) = push2_var);
```

These are automatically applied whenever PVS does a beta-reduction. (A beta reduction occurs whenever assert is applied)

Structural Induction Schema

```
stack_induction: AXIOM
      (FORALL (p: [stack -> boolean]):
         p(empty)
             AND
           (FORALL (push1_var: T, push2_var: stack):
              p(push2_var) IMPLIES
               p(push(push1_var, push2_var)))
        IMPLIES
       (FORALL (stack_var: stack): p(stack_var)));
```

Proper Subterms

```
<<(x: stack, y: stack): boolean =
   CASES y OF
     empty: FALSE,
     push(push1_var, push2_var):
        x = push2_var OR x {<<} push2_var
     ENDCASES;

 stack_well_founded: AXIOM well_founded?[stack](<<);
```

NOTE: Definition of << is recursive, but has no measure provided. None of the recursive definitions in stack_adt.pvs have a measure provided. The file is read-only, so the user cannot modify it.

Definition by Recursion

The automatically generated ADT file contains several recursion combinators. These are generally not used in practice. The usual schema for definition by recursion is available for abstract datatypes. For example, the depth of a stack could be defined by:

```
depth(s:stack): RECURSIVE nat =
  CASES s OF
    empty: 0,
    push(a,s1): 1 + depth(s1)
  ENDCASES
  MEASURE s BY <<
```

Every and Some

For each positive type parameter, PVS generates combinators every and some:

```
every(p: PRED[T])(a: stack): boolean =
  CASES a OF
    empty: TRUE,
    push(push1_var, push2_var):
        p(push1_var) AND every(p)(push2_var)
    ENDCASES;

some(p: PRED[T])(a: stack): boolean =
  CASES a OF
    empty: FALSE,
    push(push1_var, push2_var):
        p(push1_var) OR some(p)(push2_var)
    ENDCASES;
```

Map combinator

If all type parameters occur positively, a map combinator is generated:

```
map(f: [T -> T1])(a: stack[T]): stack[T1] =
  CASES a OF
    empty: empty,
    push(push1_var, push2_var):
        push(f(push1_var), map(f)(push2_var))
    ENDCASES;
```

Example: Enumerated types

The PVS declaration:

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

is an abbreviation for

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

Induction on enumerated types

Suppose you have a proof goal:

```
(FORALL (c: colors): P(c))
```

The proof command `(INDUCT "c")` splits this into three goals: `P(red)`, `P(white)`, and `P(blue)`.

Binary Trees

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

Ordered Binary Trees

```
orderedBTree [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

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

Every For Trees

```
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;
```

Predicate On Trees

```
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
```

Insert

```
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 size(A)
```

Structural Induction on Ordered Trees

```
ord_insert_step:LEMMA
pp(x) AND every(pp,A) IMPLIES every(pp, insert(x,A))
```

Prove using

```
(induct-and-simplify "A")
```

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

Proof is more intricate:

```
(induct-and-simplify "A" :rewrites "ord_insert_step")
(rewrite "ord_insert_step")
(typepred "<=’’)
(grind :if_match  all)
```

Disjoint Union Types (case-variant records)

The PVS prelude includes the following example of a disjoint union type:

```
union[T1, T2: TYPE]: DATATYPE
  BEGIN
    inl(left: T1): inl?
    inr(right: T2): inr?
  END union
```

Co-tuples

However, with PVS 3.0 and later, there is an alternative means for declaring disjoint union types.
Consider the following declaration:

```
disj_sum: TYPE = [ int + bool + [int -> bool]]
```

This behaves almost as if the declaration were:

```
disj_sum: DATATYPE
   BEGIN
      in_1(out_1: int): in?_1
      in_2(out_2: bool): in?_2
      in_3(out_3: [int -> int]): in?_3
   END disj_sum
```

Maybe

In the programming language Haskell, the Maybe functor type class is a means of being explicit that you are not sure that a function will be successful when it is executed. In PVS, we can represent this type as a disjoint union as follows:

```
Maybe[T:TYPE] : DATATYPE
BEGIN
  None : none?
  Some(some:T): some?
END Maybe
```

Mutually Recursive Datatypes

- Useful for language definition
- Not directly admissible in PVS
- Most can be accomodated in a datatype with subtypes

Example: Arithmetic Expressions

```
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
  ite(e:expr,t1:term,t2:term): ite? :term
 END arith
```

Subtypes effect on arith_adt.pvs

The generated file has the following additional declarations

```
expr((x: arith)): boolean = eq?(x);

expr: TYPE = {x: arith | eq?(x)}

term((x: arith)): boolean = num?(x) OR sum?(x) OR ite?(x);

term: TYPE = {x: arith | num?(x) OR sum?(x) OR ite?(x)}
```

An Evaluator for 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))),
    ite(e,n1,n2) :  IF b(eval(e))
                    THEN eval(n1)
                    ELSE eval(n2) ENDIF
   ENDCASES
   MEASURE a BY <<
```

Summary

- General mechanism for defining a class of recursive types
  - Lists, stacks, trees, etc.
- Same mechanism used for enumerated types and disjoint sum types
- Augmented with subtypes to provide limited form of mutual recursion

Co-datatypes

- PVS 3.x added a capability for describing co-algebraic datatypes
- Structure is similar to ADTs
- Feature is currently undocumented

The following declaration illustrates the definition of lazy lists (possibly infinite). It automatically generates the file `llist_codt.pvs`.

```
llist [T:Type]: CODATATYPE
  BEGIN
    lnull: lnull?
    lcons(car: T, cdr: llist): lcons?
  END llist
```