

# Abstract Data Types

Mariano M. Moscato

National Institute of Aerospace  
Mariano.Moscato@nianet.org



## (Incomplete) Declaration Syntax

```

<adt name>: DATATYPE
BEGIN
  [ <constructorii,j(,)j=1..n) ] : <recognizerii=1..c
END <adt name>

```

$c, n \in \mathbb{N}$

- Given as a collection of *constructors*
  - with associated *recognizers* and *accessors*
- No name repetition allowed
  - $adt\ name \neq constructor_i, adt\ name \neq accessor_{i,j}, adt\ name \neq recognizer_i$
  - $i \neq k \Rightarrow constructor_i \neq constructor_k$
  - $i \neq k \Rightarrow recognizer_i \neq recognizer_k$
  - $i \neq k \Rightarrow constructor_i \neq recognizer_k$
  - $j \neq k \Rightarrow accessor_{i,j} \neq accessor_{i,k}$



### (Incomplete) Declaration Syntax

```
<adt name>: DATATYPE
BEGIN
  [ <constructori> [ ( [ <accessori,j>: <type> ](,)j=1..n ) ]: <recognizeri> ]i=1..c
END <adt name>
```

$c, n \in \mathbb{N}$

#### ■ Example: Phonebook

```
Phonebook: DATATYPE
BEGIN
  empty_phonebook: empty?
  add_phone(
    name:string,
    phone:string,
    pb:Phonebook): add_phone?
END Phonebook
```

#### ■ constructors:

empty\_phonebook, add\_phone

#### ■ recognizers:

empty?, add\_phone?

#### ■ accessors:

name, phone, pb



- PVS provides support for a simple form of pattern-matching

```

CASES x OF
  [ constructori [ ( [ variablei,j ](,)j=1..ni ) : expression ]i ∈ [1..c]
  [ ELSE expression ]
ENDCASES
    
```

- ELSE can only be present if not all constructors were mentioned
- If some case is missing, a specific TCC is generated on typechecking
  - when no ELSE clause has been provided



```
last_inserted_was?(n: string, pb:Phonebook): bool =  
  CASES pb OF  
    empty_phonebook: FALSE,  
    add_phone(name,phone,phonebook): (n = name)  
  ENDCASES
```



During typechecking an ADT definition PVS generates:

- Definitions for the type, constructors, recognizers, and accessors
  - As uninterpreted declarations
- Additional operators
  - `subterm`, `<<`, `reduce_nat`, `reduce_ordinal`
- Several axioms



- *extensionality* for constant constructors
  - there is only one bottom element for every constant constructor

- In the Phonebook example:

```
Phonebook_empty_phonebook_extensionality: AXIOM
  FORALL (e1: (empty?), e2: (empty?)):
    e1 = e2
```



- *extensionality* for constructors with arguments
  - elements are distinguishable by the accessors

- In the Phonebook example:

```
Phonebook_add_phone_extensionality: AXIOM
  FORALL (v1: (add_phone?), v2: (add_phone?):
    name(v1) = name(v2) AND
    phone(v1) = phone(v2) AND
    pb(v1) = pb(v2)
  IMPLIES v1 = v2
```





- *eta* axiom
  - if the values returned by accessors are used to construct a new element, the same element is constructed.

- In the Phonebook example:

```
Phonebook_add_phone_eta: AXIOM
  FORALL (v: (add_phone?)):
    add_phone(name(v), phone(v), pb(v)) = v
```



- Meaning of accessors
- In the Phonebook example:

```
Phonebook_name_add_phone: AXIOM
  FORALL (n: string, p: string, pb: Phonebook):
    name(add_phone(n, p, pb)) = n
```

```
Phonebook_name_add_phone: AXIOM
  FORALL (n: string, p: string, pb: Phonebook):
    phone(add_phone(n, p, pb)) = p
```

```
Phonebook_name_add_phone: AXIOM
  FORALL (n: string, p: string, pb: Phonebook):
    phonebook(add_phone(n, p, pb)) = pb
```



- *Inclusion and Disjointness*

- recognizers characterize all the elements of the type

- In the Phonebook example:

```
Phonebook_inclusive: AXIOM
  FORALL (phonebook: Phonebook):
    empty?(phonebook) OR add_phone?(phonebook)
```

```
Phonebook_disjointness: AXIOM
  FORALL (phonebook: Phonebook):
    NOT (empty?(phonebook) AND add_phone?(phonebook))
```



- *Structural Induction Scheme*

- In the Phonebook example:

```
Phonebook_induction: AXIOM
  FORALL (p: [Phonebook -> boolean]):
    (p(empty_phonebook) AND
     (FORALL (name: string, phone: string, pb: Phonebook):
       p(pb) IMPLIES
         p(add_phone(name, phone, pb))))
  IMPLIES (FORALL (phonebook: Phonebook): p(phonebook))
```



- `subterm(x,y: Phonebook): boolean`
  - indicates if `x` participates in the construction of `y`

- In the Phonebook example<sup>1</sup>:

```
subterm(x: Phonebook, y: Phonebook): boolean =  
  x = y OR  
  CASES y  
    OF empty_phonebook: FALSE,  
       add_phone(name, phone, phonebook):  
         subterm(x, phonebook)  
    ENDCASES
```

---

<sup>1</sup>Included for illustrative purposes only, actually not well typed PVS code



- <<
- relational version of `subterm`, along with
- an axiom stating its *well-foundedness*

- In the Phonebook example<sup>2</sup>:

```
<<: [Phonebook,Phonebook -> boolean] =
  LAMBDA (x, y: Phonebook):
    CASES y
      OF empty_phonebook: FALSE,
         add_phone(name, phone, pb):
           x = pb OR x << pb
      ENDCASES
```

Phonebook\_well\_founded: AXIOM strict\_well\_founded?[Phonebook](<<)

<sup>2</sup>Included for illustrative purposes only, actually not well typed PVS code



- (Several) *Reduce* functions
  - reduce an element to a natural number, to an ordinal, or to a range.
  - useful for simplifying the proof of termination of user-defined functions
- In the Phonebook example<sup>3</sup>:

```
reduce_nat(empty_red: nat,  
           add_phone_red: [[string,string,nat]->nat]):  
  [Phonebook -> nat] =  
LAMBDA (phonebook: Phonebook):  
  LET reduce: [Phonebook->nat] =  
    reduce_nat(empty_red, add_phone_red)  
  IN CASES phonebook OF  
    empty_phonebook: empty_red,  
    add_phone(name, phone, pb): add_phone_red(name, phone, reduce(pb))  
  ENDCASES
```

<sup>3</sup>Included for illustrative purposes only, actually not well typed PVS code



- The usual schema for definition by recursion is supported by the implicit definitions
- Example

```
size(phonebook: Phonebook): RECURSIVE nat =  
  CASES phonebook OF  
    empty_phonebook: 0,  
    add_person(n,p,pb): 1+size(pb)  
  ENDCASES  
MEASURE phonebook BY <<
```





## (Incomplete) Declaration Syntax

```

<adt name> [ [ [ <arg namei> : [ TYPE | <type> ] ] ] ]i=1..a(,) : DATATYPE
BEGIN
  [ [ <constructori> [ ( [ [ <accessori,j> : <type> ] ] ]j=1..n(,) ) : <recognizeri> ] ]i=1..c
END <adt name>

```

$c, n, a \in \mathbb{N}$

### ■ Example: Stack

```

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

```



- For every type parameter combinators *every* and *some* are generated
  - “a predicate holds for **every** element included in the structure”
  - “a predicate holds for **some** element included in the structure”
- In the Stack example:

```
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
```



- A *map* combinator is also generated

- In the Stack example:

```
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
```



## (Incomplete) Declaration Syntax

$$\langle \text{adt name} \rangle \left[ \left[ \langle \text{arg name}_i \rangle : [ \text{TYPE} \mid \langle \text{type} \rangle ] \right]_{(,)}^{i=1 \dots a} \right] : \text{DATATYPE}$$

$$\left[ \text{WITH SUBTYPES } \left[ \langle \text{subadt name}_i \rangle \right]_{(,)}^{i=1 \dots s} \right]$$

BEGIN

$$\left[ \langle \text{constructor}_i \rangle \left[ \left( \left[ \langle \text{accessor}_{i,j} \rangle : \langle \text{type} \rangle \right]_{(,)}^{j=1 \dots n} \right) : \langle \text{recognizer}_i \rangle \left[ : \langle \text{subadt name}_k \rangle \right] \right] \right]_{(,)}^{i=1 \dots c}$$

END  $\langle \text{adt name} \rangle$

 $c, n, a, s \in \mathbb{N}$ 

- Abstract datatypes may also define subtypes
- There must be no repetitions in the list of subtype names
- Each constructor must return an element of a particular subtype
- Implicit declarations are also affected by the introduction of subtypes



```
ArithExpr: DATATYPE WITH SUBTYPES NumExpr, BoolExpr
BEGIN

  CONST (x:real)                :constant? : NumExpr
  ADD   (x1,x2:NumExpr)         :addition? : NumExpr

  EQUALS(x1,x2:NumExpr)        :equals?   : BoolExpr
  ITE   (b:BoolExpr, x1,x2:NumExpr):ite?    : NumExpr

END ArithExpr
```

- Some of the implicit declarations

- ArithExpr: TYPE
- NumExpr(x:ArithExpr): boolean =  
constant?(x) OR addition?(x) OR ite?(x)
- NumExpr: TYPE =  
{x:ArithExpr | constant?(x) OR addition?(x) OR ite?(x)}
- BoolExpr(x:ArithExpr): boolean = equals?(x)
- BoolExpr: TYPE = (equals?)



- Datatype definitions can be expressed as *top-level* declarations
  - Not enclosed by a theory definition
  - In such case, most of the implicit declarations are automatically printed out in a read-only separated file
  - *name-of-type\_adt.pvs*
- Also supports IMPORTING and ASSUMING parts
- Imported types, parameter types, and subtypes must appear in *positive* form



$$\langle \text{adt name} \rangle \left[ \left[ \left[ \langle \text{arg name}_i \rangle : [ \text{TYPE} \mid \langle \text{type} \rangle ] \right]_{(,)}^{i=1 \dots a} \right] \right] : \text{DATATYPE}$$

$$\left[ \text{WITH SUBTYPES} \left[ \langle \text{subadt name}_i \rangle \right]_{(,)}^{i=1 \dots s} \right]$$

BEGIN

$$\left[ \text{IMPORTING} \left[ \langle \text{theory name}_i \rangle \right]_{(,)}^{i=1 \dots t} \right]$$

$$\left[ \text{BEGIN ASSUMING} \left[ \langle \text{assumption}_i \rangle \right]_{(,)}^{i=1 \dots a'} \text{ END ASSUMING} \right]$$

$$\left[ \langle \text{constructor}_i \rangle \left[ \left( \left[ \langle \text{accessor}_{i,j} \rangle : \langle \text{type} \rangle \right]_{(,)}^{j=1 \dots a''} \right) : \langle \text{recognizer}_i \rangle \left[ : \langle \text{subadt name}_k \rangle \right] \right] \right]_{(,)}^{i=1 \dots c}$$
END  $\langle \text{adt name} \rangle$ Where  $a, s, t, a', a'', c \in \mathbb{N}$



- PVS supports an useful abbreviation to define enumerated types
- Example

Finger: TYPE = { Thumb, Index, Middle, Ring, Baby}  
is an abbreviation for

```
Finger: DATATYPE
  BEGIN
    Thumb: Thumb?
    Index: Index?
    Middle: Middle?
    Ring: Ring?
    Baby: Baby?
  END
```





- Also known as cotuple, coproduct or sum types

## Declaration Syntax

$$\langle \textit{type name} \rangle : \text{TYPE} = [ [\langle \textit{type expr} \rangle ]_{(+)}^{i=1..n} ]$$

- Examples
  - BoolOrIntOrPred: TYPE = [ bool + int + [int->bool] ]
  - Either[S,T: TYPE]: TYPE = [ S + T ]



- Also known as cotuple, coproduct or sum types

## Declaration Syntax

$$\langle type\ name \rangle : TYPE = [ [\langle type\ expr \rangle]_{(+)}^{i=1..n} ]$$

- For each type in the union, these operators are implicitly generated

**Injector**  $IN\_i : [\langle type\ expr \rangle_i \rightarrow \langle type\ name \rangle]$

**Recognizer**  $IN?\_i : [\langle type\ name \rangle \rightarrow bool]$

**Extractor**  $OUT\_i : [(IN?\_i) \rightarrow \langle type\ expr \rangle_i]$

$in\_i$ ,  $in?\_i$ , and  $out\_i$  are also generated (with same semantics)

- Injectors can be used as discriminators in CASES OF expressions

(!) Union types are not *technically* abbreviations of ADTs but...



```
Either[S,T: TYPE]: TYPE = [ S + T ]
```

```
duplicate_left_half_right(data: Either[nat,real]): real =  
  CASES data OF  
    IN_1(n): 2*n,  
    IN_2(x): x/2  
  ENDCASES
```



```
ArithExpr: DATATYPE WITH SUBTYPES NumExpr, BoolExpr
BEGIN
  CONST (x:real)           :constant? : NumExpr
  ADD   (x1,x2:NumExpr)    :addition? : NumExpr
  EQUALS(x1,x2:NumExpr)   :equals?   : BoolExpr
  ITE   (b:BoolExpr, x1,x2:NumExpr):ite?   : NumExpr
END ArithExpr
```

```
Value: TYPE = [ bool + real ]
```

```
eval(expr: ArithExpr): RECURSIVE Value =
  CASES expr OF
    CONST(x):      IN_2(x),
    ADD(x1,x2):    IN_2( OUT_2(eval(x1)) + OUT_2(eval(x2)) ),
    EQUALS(x1,x2): IN_1( OUT_2(eval(x1)) = OUT_2(eval(x2)) ),
    ITE(b,x1,x2):  IF OUT_1(eval(b))
                  THEN eval(x1)
                  ELSE eval(x2) ENDIF
  ENDCASES
```



```
IMPORTING ArithExpr
```

```
Value: TYPE = [ bool + real ]
```

```
ValueFor(expr: ArithExpr): TYPE =
```

```
{v: Value | IF NumExpr(expr) THEN IN?_2(v) ELSE IN?_1(v) ENDIF}
```

```
eval(expr: ArithExpr): RECURSIVE ValueFor(expr) =
```

```
CASES expr OF
```

```
  CONST(x):      IN_2(x),
```

```
  ADD(x1,x2):    IN_2(OUT_2(eval(x1))+OUT_2(eval(x2))),
```

```
  EQUALS(x1,x2): IN_1(OUT_2(eval(x1))=OUT_2(eval(x2))),
```

```
  ITE(b,x1,x2):  IF OUT_1(eval(b))
```

```
    THEN eval(x1)
```

```
    ELSE eval(x2) ENDIF
```

```
ENDCASES
```

```
MEASURE expr BY <<
```



```
IMPORTING ArithExpr
```

```
Value: DATATYPE BEGIN
  bool(val:bool):bool?
  real(val:real):real?
END Value
```

```
ValueFor(expr: ArithExpr): TYPE =
  {v: Value | IF NumExpr(expr) THEN real?(v) ELSE bool?(v) ENDIF}
```

```
eval(expr: ArithExpr): RECURSIVE ValueFor(expr) =
  CASES expr OF
    CONST(x):      real(x),
    ADD(x1,x2):    real(val(eval(x1))+val(eval(x2))),
    EQUALS(x1,x2): bool(val(eval(x1))=val(eval(x2))),
    ITE(b,x1,x2):  IF val(eval(b))
                  THEN eval(x1)
                  ELSE eval(x2) ENDIF
  ENDCASES
```



- 1 General Form
- 2 Declaration
- 3 CASES Expressions
- 4 Implicit Declarations
- 5 Recursive Definitions
- 6 Formal Parameters
- 7 Subtypes
- 8 Remarks
- 9 Enumerated Types
- 10 Disjoint Unions