# Abstract Data Types

Mariano M. Moscato

National Institute of Aerospace
Mariano.Moscato@nianet.org

## (Incomplete) Declaration Syntax

$$
\begin{array}{l}
\langle \textit{adt name} \rangle\texttt{: DATATYPE} \\
\texttt{BEGIN} \\
\quad \left[ \langle \textit{constructor}_i \rangle \ \left[ \texttt{(} \left[ \langle \textit{accessor}_{i,j} \rangle \texttt{:} \ \langle \textit{type} \rangle \right]^{j=1\cdot\cdot n}_{(,)} \texttt{)} \right] \texttt{:} \ \langle \textit{recognizer}_i \rangle \right]^{i=1\cdot\cdot c} \\
\texttt{END} \ \langle \textit{adt name} \rangle
\end{array}
$$

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

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

## (Incomplete) Declaration Syntax

$$
\begin{array}{l}
\langle \textit{adt name} \rangle\texttt{: DATATYPE} \\
\texttt{BEGIN} \\
\quad \left[ \langle \textit{constructor}_i \rangle \; \left[ \texttt{(} \left[ \langle \textit{accessor}_{i,j} \rangle \texttt{:} \; \langle \textit{type} \rangle \right]_{(,)}^{j=1..n} \texttt{)} \right] \texttt{:} \; \langle \textit{recognizer}_i \rangle \right]^{i=1..c} \\
\texttt{END} \; \langle \textit{adt name} \rangle
\end{array}
$$

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

- 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

# CASES Expressions

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

$$
\begin{array}{l}
\texttt{CASES}\ x\ \texttt{OF} \\
\quad \left[ \langle constructor_i \rangle \ \left[ \texttt{(} \left[ \langle variable_{i,j} \rangle \right]_{(,)}^{j=1 \cdot\cdot n_i} \texttt{)} \right] \texttt{:}\ \langle expression \rangle \right]^{i \in [1 \cdot\cdot c]} \\
\quad \left[ \texttt{ELSE}\ \langle expression \rangle \ \right] \\
\texttt{ENDCASES}
\end{array}
$$

- 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
```

# Implicit Declarations

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
  - using the accessed values to build a new element, results in the same element

- 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*
- 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)`
  - indicates if `x` participates in the construction of `y`

- In the Phonebook example:

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

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

- In the Phonebook example:

```
<<:  (strict_well_founded?[Phonebook]) =
    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](<<)
```

- (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:

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

# Recursive Definitions

- 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 <<
```

# Formal Parameters

## (Incomplete) Declaration Syntax

$$\langle \textit{adt name} \rangle \left[ \texttt{[} \left[ \langle \textit{arg name}_i \rangle \texttt{:}\ [\ \texttt{TYPE}\ |\ \langle \textit{type} \rangle\ ] \right]_{(,)}^{i=1\cdots a} \texttt{]} \right] \texttt{: DATATYPE}$$

$$\texttt{BEGIN}$$

$$\left[ \langle \textit{constructor}_i \rangle\ \left[ \texttt{(} \left[ \langle \textit{accessor}_{i,j} \rangle \texttt{:}\ \langle \textit{type} \rangle \right]_{(,)}^{j=1\cdots n} \texttt{)} \right] \texttt{:}\ \langle \textit{recognizer}_i \rangle \right]^{i=1\cdots c}$$

$$\texttt{END}\ \langle \textit{adt name} \rangle$$

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

- 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
```

# Formal Parameters

- 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
```

# Subtypes

## (Incomplete) Declaration Syntax

$$\langle \textit{adt name}\rangle \Big[\texttt{[}\big[\langle \textit{arg name}_i\rangle\texttt{: } [\ \texttt{TYPE} \mid \langle \textit{type}\rangle\ ]\big]^{i=1\cdots a}_{(,)}\texttt{]}\Big]\texttt{: DATATYPE}$$
$$\Big[\texttt{WITH SUBTYPES } \big[\langle \textit{subadt name}_i\rangle\big]^{i=1\cdots s}_{(,)}\Big]$$
$$\texttt{BEGIN}$$
$$\Big[\langle \textit{constructor}_i\rangle\ \Big[\texttt{(}\big[\langle \textit{accessor}_{i,j}\rangle\texttt{: } \langle \textit{type}\rangle\big]^{j=1\cdots n}_{(,)}\texttt{)}\Big]\texttt{:}\langle \textit{recognizer}_i\rangle\ \Big[\texttt{:}\langle \textit{subadt name}_k\rangle\Big]\Big]^{i=1\cdots c}$$
$$\texttt{END } \langle \textit{adt name}\rangle$$

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

- 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

# Subtypes
## Example

```
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?)`

# Remarks

- Datatype definition can be a *top-level* declaration (as a theory)
  - If declared in its own file, 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

$$\langle \textit{adt name} \rangle \Big[ \texttt{[} \big[ \langle \textit{arg name}_i \rangle \texttt{:}\ [\ \texttt{TYPE} \mid \langle \textit{type} \rangle\ ] \big]_{(,)}^{i=1 \cdots a} \texttt{]} \Big] \texttt{:}\ \texttt{DATATYPE}$$

$$\Big[ \texttt{WITH SUBTYPES}\ \big[ \langle \textit{subadt name}_i \rangle \big]_{(,)}^{i=1 \cdot\cdot s} \Big]$$

$$\texttt{BEGIN}$$

$$\Big[ \texttt{IMPORTING}\ \big[ \langle \textit{theory name}_i \rangle \big]_{(,)}^{i=1 \cdot\cdot t} \Big]$$

$$\Big[ \texttt{BEGIN ASSUMING}\ \big[ \langle \textit{assumption}_i \rangle \big]^{i=1 \cdot\cdot a'}\ \texttt{END ASSUMING}\ \Big]$$

$$\Big[ \langle \textit{constructor}_i \rangle\ \Big[ \texttt{(} \big[ \langle \textit{accessor}_{i,j} \rangle \texttt{:}\ \langle \textit{type} \rangle \big]_{(,)}^{j=1 \cdot\cdot a''} \texttt{)} \Big] \texttt{:}\ \langle \textit{recognizer}_i \rangle \Big[ \texttt{:} \langle \textit{subadt name}_k \rangle \Big]\ \Big]^{i=1 \cdot\cdot c}$$

$$\texttt{END}\ \langle \textit{adt name} \rangle$$

Where $a, s, t, a', a'', c \in \mathbb{N}$

# Enumerated Types

- 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
```

# Disjoint Unions

- Also know as cotuple, coproduct or sum types

**Declaration Syntax**

$\langle \textit{type name} \rangle$`: TYPE = [` $\left[\langle \textit{type expr} \rangle\right]_{(+)}^{i=1 \cdot\cdot n}$ `]`

- Examples

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

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

# Disjoint Unions

- Also know as cotuple, coproduct or sum types

**Declaration Syntax**

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

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

Injector `IN_`$i$ `:` `[`$\langle \mathit{type\ expr} \rangle_i$ `->` $\langle \mathit{type\ name} \rangle$`]`

Recognizer `IN?_`$i$`:` `[`$\langle \mathit{type\ name} \rangle$ `-> bool]`

Extractor `OUT_`$i$`:` `[(IN?_`$i$`) ->` $\langle \mathit{type\ expr} \rangle_i$`]`

`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_halve_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_2(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: 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_2(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
MEASURE expr BY <<
```

# Summary