Declarations and Types in the PVS Specification Language

Ben Di Vito

NASA Langley Research Center
Formal Methods Team

b.divito@nasa.gov

NASA Langley – NIA Short Course on PVS

9–12 October 2012

Declarations

Named entities are introduced in PVS by means of declarations.

- User-defined language units such as constants, variables, types, and functions are introduced through a series of declarations.
- Examples:

```
seconds_per_hour: nat = 3600
minute:           TYPE = {m: nat | m < 60}
before, after:    VAR minute
```

- Collections of related declarations are grouped together into PVS *theories*.
- A set of predefined theories called the *prelude* is available as the user's starting point.

Declarations (Cont'd)

- Named items used in a declaration must have already been declared previously.
  - No forward references
  - Note the order in the example above
- A declared entity is visible throughout the rest of the theory in which it is declared.
  - It may also be exported to other theories (variables excepted).
  - Variables can be introduced using local bindings, with much more limited scope.

Kinds of Declarations

PVS specification language allows a variety of top-level declarations.

- Type declarations
- Variable declarations
- Constant declarations
- Recursive definitions
- Macros
- Inductive/coinductive definitions
- Formula declarations
- Judgements
- Conversions
- Library declarations
- Auto-rewrite declarations

There are also importing directives.

Theories

Specifications are modularized in PVS by organizing them into theories.

- Declarations within a theory may freely use earlier declarations within that same theory.
- Declarations from other theories may be used when properly imported.

```
IMPORTING sqrt, real_sets[nonneg_real]
```

  - Default rule: all declared entities (other than variables) are exportable.
- Theories may be parameterized so that specialized instances can be created.
  - Theory parameters include constants and types.
  - Constitutes a powerful mechanism for creating generic theories that are readily reused.
- Named items imported from different theories may clash, requiring name resolution.

Theories (Cont'd)

General form for theories:

```
My_Theory [<parameters>]: THEORY
BEGIN
  <assuming part>
  <declaration>
        .
        .
        .
  <declaration>
END My_Theory
```

- PVS allows multiple theories per file.
- In normal usage, we recommend only one theory per file.

Variables

Logical variables in PVS are used to express other declared entities.

- Basic form of a variable declaration:

```
name_1,...,name_n: VAR <data type>
```

- Scope extends to end of theory.
- Variables in PVS are *not* the same concept as programming language variables.
  - PVS variables are logical or mathematical variables.
  - They range over a (possibly infinite) set of values.
  - No notion of program state is inherent in these variables.
- Variables are not exportable outside of their containing theories.
  - Each theory declares its own variables.

Local Bindings

Local variables are also possible in PVS.

- Local bindings are embedded within declarations for larger containing units:

```
delta_time(current:  system_time,
           previous: system_time): system_time =  . . .
```

- The scope of such local variables is limited to the containing unit.
- Local bindings can *shadow* previous bindings or declarations in the containing scope.
- Local variables or bindings may be used in several PVS constructs:
  - Quantifiers
  - LAMBDA expressions
  - LET and WHERE expressions
  - Type expressions

Constants

Named constants may be introduced as needed for use in other declarations.

- Basic forms of a constant declaration:

```
name: <type> = <value>
name: <type>
```

- A constant may be either:
  - *Interpreted* (having a definite value) or
  - *Uninterpreted* (value left unspecified)
- Practical consequences of this choice:
  - When the value is specified, it is available for use in proofs.
  - If unspecified, anything proved using the constant will be true for any legitimate value it could have.

Constants (Cont'd)

- Declaring a constant requires that its type be nonempty.
- Like variables, constants are not the same concept as programming language constants.
- Function declarations are special cases of constant declarations.
  - A function declaration is a constant having a function type in the higher-order logic framework of PVS.

Type Concepts

PVS provides a rich set of type capabilities.

- A type is considered to be a (possibly infinite) set of values.
- Types may be declared in one of several ways:
  - As uninterpreted types with no assumed characteristics
  - As instances of predefined or user-defined types
  - Through mechanisms for creating types for structured data objects
  - Through a mechanism for creating *subtypes*
  - Through a mechanism for creating abstract data types
- Higher-order logic plays a big role in the type system.
  - Function types are used to model common concepts such as arrays.
- Interpreted types are declared using *type expressions*.
- PVS uses *structural equivalence* not name equivalence.

Predefined Types

PVS provides some basic predefined types for use in declarations.

- Boolean values: `bool`
  - Includes the constants `true` and `false`
  - Accompanied by the usual boolean operations
- Integers: `int` and `nat`
  - `int` includes the full set of integers from negative to positive infinity.
  - `nat` includes the nonnegative subset of `int`.
  - Accompanied by the usual constants and operations.
  - `int` and `nat` also have various subtypes declared in the prelude:

    ```
    posnat, posint, negint, ...
    ```
  - Can also specify subranges of `nat`, e.g.:

    ```
    below(8) :  0, ..., 7      upto(8)   :  0, ..., 8
    above(8) :  9, 10, ...     upfrom(8) :  8, 9, ...
    ```

Predefined Types (Cont'd)

- Rational numbers: `rational`
  - Axiomatizes the true mathematical concept of rationals.
  - Rational constants are sometimes used to approximate real constants.
- Real numbers: `real`
  - Axiomatizes the true mathematical concept of reals.
  - Different from the programming notion of floating point numbers.
  - Axioms for real number field taken from Royden.
- All axioms and derived properties for the predefined types are extensively enumerated and documented in the prelude.
  - The prelude itself is written in PVS notation.
  - Prelude extensions are also possible.

Uninterpreted Types

Types may be named and left unspecified.

- Basic form of an uninterpreted type declaration:

```
name: TYPE
```

  - Identifies a named type without assuming anything about the values.
  - Only operation allowed on objects of this type is comparison for equality.

- Alternate form of uninterpreted type:

  `name: NONEMPTY_TYPE` or `name: TYPE+`

  - Difference is the assumption of nonemptiness.

- One uninterpreted type may be a subtype of another:

```
name_2: FROM NONEMPTY_TYPE name_1
```

  - Some subset of `name_1`'s values may be used in the new type.

Predicate Subtypes

Often we need to derive types as subsets of other types.

- PVS allows predicate subtypes to be declared directly:

```
posint:   TYPE = {n: int | n > 0}
index:    TYPE = {n: int | 1 <= n AND n <= num_units}
                 CONTAINING 1
fraction: TYPE = {x: real | -1 < x AND x < 1}
oddint:   TYPE = {n: int | odd?(n)}
```

- All properties of the parent type are inherited by the subtype.
- A constraining predicate is provided to identify which elements are contained in the subset.
- A CONTAINING clause may be added to show nonemptiness.
- Type correctness conditions (TCCs) may be generated to impose a nonemptiness requirement.

Enumeration Types

The familiar concept of enumeration type is available in PVS.

- Basic declarations:

```
color:       TYPE = {red, white, blue}
flight_mode: TYPE = {going_up, going_down}
```

- Value identifiers become constants of the type.
  - The constants are considered distinct.
  - Axioms are generated that state these inequalities.
  - Example: `red /= white`
  - An inclusion axiom states that the explicit constants exhaust the type.
- Constant identifiers may be used in expressions.

Function Types

A key feature of PVS and its style of formalization is the function-type capability.

- Functions types are declared using explicit domain and range types:

```
status:       TYPE = [LRU_id -> bool]
operator:     TYPE = [int, int -> int]
operator:     TYPE = FUNCTION[int, int -> int]
control_bank: TYPE = ARRAY[LRU_id -> control_block]
```

- Reserved words FUNCTION and ARRAY provide alternate forms with equivalent meaning.
- A value of a function type is a mathematical object: any legitimate function having the required signature.
  - Values may be constructed using LAMBDA expressions.
  - This feature is fully higher order: domain and range types may themselves be function types.

Function Types (Cont'd)

Function types make the language very expressive and allow some rather sophisticated mathematics to be formalized directly.

- Functions types are also the primary means in PVS of modeling structured data objects such as vectors and arrays.
- Consider an array type in a procedural programming language notation:

```
memory: ARRAY address OF word
```

- This would be represented in PVS with a function type:

```
memory: [address -> word]
```

- Array access in a programming language is typically denoted `M[a]`
  - In PVS we use function application: `M(a)`

More on Predicates and Types

Certain types involving predicates are treated as special cases.

- A predicate type can be declared explicitly or using a shorthand:

```
nat_pred: TYPE = [nat -> bool]
nat_pred: TYPE = pred[nat]
nat_pred: TYPE = setof[nat]
```

- Predicate subtypes also can be specified using a shorthand:

```
prime?(n: nat): bool = ...
primes: TYPE = {n: nat | prime?(n)}
primes: TYPE = (prime?)
```

- Personal taste dictates which way to declare types.
  - Explicit method for novices vs. shorthand for experts.
  - Shorthand notations pop up a lot, however.
  - Need to be able to recognize them.

Tuple Types

Structured data objects in the form of tuples can be modeled using tuple types.

- Declarations include types for each element:

```
pair:      TYPE = [int, int]
position: TYPE = [real, real, real]
two_bits: TYPE = [bool, bool]
```

- Instances are easily specified:

```
(1, 2, 3)
```

- Tuple elements are organized positionally.

  (1, 2) $\neq$ (2, 1)

- Elements are extracted using special notation or predefined projection functions.

Record Types

Similarly structured data objects can be modeled using record types.

- Declarations include types for each element:

```
pair:      TYPE = [# left: int, right: int #]
vector:    TYPE = [# x: real, y: real, z: real #]
ctl_block: TYPE =
    [# active: bool, timestamp: TOD, status: op_mode #]
```

- Instances are easily specified:

```
(# x := 1, y := 2, z := 3 #)
```

- Record elements are organized by keyword.

```
(# left := 1, right := 2 #) =
(# right := 2, left := 1 #)
```

- Elements are extracted using special notation or function application based on the element names.

Other Type Concepts

Two additional typing mechanisms are available in PVS.

- Abstract data types are introduced by giving a scheme for defining constructors and access functions.

```
list[base: TYPE]: DATATYPE
    BEGIN
      null: null?
      cons (car: base, cdr: list) : cons?
    END list
```

- This declaration causes axioms and derived functions to be generated based on the DATATYPE scheme.
  - Example: induction axiom usable within the prover.
- CODATATYPE is also available for coalgebraic formalization.

Other Type Concepts (Cont'd)

- *Dependent types* offer another powerful typing concept:

```
date1: TYPE = [ yr: year, mon: month,
                {d: posnat | d <= days(mon, yr)} ]
date2: TYPE = [# yr: year, mon: month,
                 day: {d: posnat | d <= days(mon, yr)} #]
```

- These declarations introduce a tuple and a record structure where the type of component `day` depends on the *values* of month and year that precede it in the structure.

- Allows complex data type dependencies to be modeled, obviating the messy specifications that would be necessary without this feature.

- Can also be used in other contexts such as function arguments.

```
ratio(x, y: real, z: {z: real | z /= x}): real =
     (x - y) / (x - z)
```

- TCCs are generated as needed to ensure well-formed values.

Lexical Rules

PVS has a conventional lexical structure.

- Comments begin with '%' and go to the end of the line.
- Identifiers are composed of letters, digits, '?', and '_'.
  - They must begin with a letter.
  - They are case sensitive.
- Integers are composed of digits only.
- Rationals can be written as ratios or with decimal notation.
  - 2.718 is equivalent to 2718/1000
  - Leading zeros are required: 0.866
  - No floating point formats

Lexical Rules (Cont'd)

- Strings are enclosed in double quotes.
- Reserved words are not case sensitive.
  - Examples: FORALL exists BEGIN end
- Many special symbols
  - Examples: `[# #] -> (: :) >=`

Examples of Declarations

```
major_mode_code:    TYPE = nat
mission_time:       TYPE = real
GPS_id:             TYPE = {n: nat | 1 <= n & n <= 3}

receiver_mode:      TYPE = {init, test, nav, blank}
AIF_flag:           TYPE = {auto, inhibit, force}

M50_axis:           TYPE = {Xm, Ym, Zm}
IMPORTING           vectors[M50_axis]
M50_vector:         TYPE = vector[M50_axis]

position_vector:    TYPE = M50_vector
velocity_vector:    TYPE = M50_vector

GPS_predicate:      TYPE = [GPS_id -> bool]
GPS_positions:      TYPE = [GPS_id -> position_vector]
GPS_velocities:     TYPE = [GPS_id -> velocity_vector]
GPS_times:          TYPE = [GPS_id -> mission_time]
```

Sample Declarations (Cont'd)

```
vectors [index_type: TYPE]: THEORY
BEGIN

vector:         TYPE = [index_type -> real]

i,j,k:          VAR index_type
a,b,c:          VAR real
U,V:            VAR vector

zero_vector:       vector = LAMBDA i: 0
vector_sum(U, V):  vector = LAMBDA i: U(i) + V(i)
vector_diff(U, V): vector = LAMBDA i: U(i) - V(i)
scalar_mult(a, V): vector = LAMBDA i: a * V(i)

 . . .

END vectors
```

Sample Declarations (Cont'd)

```
matrices [row_type, col_type: TYPE]: THEORY
BEGIN

vector:         TYPE = [col_type -> real]
matrix:         TYPE = [row_type -> vector]

vector_2:       TYPE = [row_type -> real]
matrix_2:       TYPE = [col_type -> vector_2]

i:              VAR row_type
j:              VAR col_type
a,b,c:          VAR real
U,V:            VAR vector
M,N:            VAR matrix

 . . .

END matrices
```