Basic PVS Specification Language Features

Ben L. Di Vito

NASA Langley Research Center
Formal Methods Team

b.l.divito@larc.nasa.gov
phone: (757) 864-4883
fax: (757) 864-4234
http://shemesh.larc.nasa.gov/~bld

NASA Langley PVS Training Course
22–25 April 2003

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
- Numbers are composed of digits only — no floating point format
- Strings are enclosed in double quotes
- Reserved words are not case sensitive
  - Examples: FORALL exists BEGIN end
- Many special symbols
  - Examples: `[# #] -> (: :) >=`

Expressions

PVS allows many operators and constructors for use in forming expressions

- Equality relations
- Arithmetic expressions
- Logical expressions, formulas
- Conditional expressions
- Function application
- Lambda abstraction
- Override expressions
- Record construction, component access
- Tuple construction, component access
- LET and WHERE expressions
- Set expressions
- Lists and strings
- Pattern matching on data types
- Name resolution

Every expression must be properly typed

- Typechecker emits TCCs if it's unsure

Equality relations

Equality operations are defined for any type

- Two operators available:

  `x = y`

  `z /= 7`

- Both sides of an equality/inequality must be of compatible types

  `x * y = 4` is valid

  `true /= 4` is illegal

- A (dis)equality is legal if there is a common supertype
- TCCs may be generated when subtypes are involved
- Equality on function values entails special techniques when proving
  - Use of *extensionality* inference rule

Arithmetic Expressions

PVS has the usual assortment of arithmetic operations

- Relational operators:

  `<, <=, >, >=`

- Binary operators:

  `+, -, *, /, ^`

- Unary operators:

  `-`

- Numeric constants are limited to integers
  - Decimal point format is not available
  - Can construct rational numbers
  - Examples: `1/2, 22/7`
- Base type for arithmetic is real
  - Subtypes built in for naturals, integers, etc.
  - Automatic coercions performed when needed

Logical Expressions and Formulas

Logical expressions may be used to construct both propositional and predicate calculus formulas

- Logical constants: `true` and `false`
- Propositional connectives:
  - Negation: `NOT`
  - Conjunction: `AND`, `&`
  - Disjunction: `OR`
  - Implication: `=>`, `IMPLIES`
  - Equivalence: `<=>`, `IFF`
- Quantified formulas:
  - Universal: `FORALL x: P(x)`, also with `ALL`
  - Existential: `EXISTS x: Q(x)`, also with `SOME`
- A few other synonyms and operators are available

Conditional Expressions

Conditional expressions come in two basic varieties

- IF expressions:

```
IF a THEN b ELSE c ENDIF
```

- Evaluates to either `b` or `c` according to the value of boolean expression `a`
- Subexpressions `b` and `c` must have compatible types
- Type of resulting expression is the common supertype of `b` and `c`
- The ELSE clause is not optional
- Also can have multiple tests and branches:

```
IF x < 0 THEN -1 ELSIF x = 0 THEN 0 ELSE 1 ENDIF
```

- Can include any number of ELSIF clauses

Conditional Expressions (Cont'd)

- COND expressions:

```
COND m = n -> n,
     m > n -> gcd(m - n, n),
     m < n -> gcd(m, n - m)
ENDCOND
```

- Allows multiway conditional evaluation similar to IF expressions containing ELSIF clauses
- PVS generates coverage and disjointness TCCs to ensure expression is well formed
  - *Disjointness:* at most one case applies
  - *Coverage:* at least one case applies
- COND expressions are used in table-based specifications

Tabular Expressions

Complex conditional expressions can be put in the form of tables:

```
TABLE %-----------------------------------------%
      |[  m = n  |     m > n     |     m < n    ]|
      %-----------------------------------------%
      |     n    | gcd(m - n, n) | gcd(m, n - m) |
      %-----------------------------------------%
ENDTABLE
```

- Semantically equivalent to COND expressions
- More complex forms also available
- Can directly express many types of tables used in practice
- Well-formedness analysis is available through TCC mechanism

Function Application

Function application can be a little more involved than normal when higher-order features are present

- Basic function application:

  `f(x)    g(y, z)    h(0, f(a)) + 1`

- Infix operators can be applied in prefix style

  `+(x, y)    *(y, -(z, 1))`

- Expressions can evaluate to functions, which are then applied to other expressions

`f: [nat -> [real -> real]]`	allows	`f(1)(x)`
`g: [nat,nat -> [real -> real]]`	allows	`g(2,3)(h(z))`
`h: [nat,real -> [bool,int -> real]]`	allows	`h(0, f(a))(true, 39)`

- Signatures of functions and corresponding types are used to sort things out
- Function being applied could be given as the value of a variable, which looks the same as regular application

  `f(x), g(y, z)` if `f` and `g` are variables of suitable function types

Lambda Abstraction

Lambda expressions allow writing function-valued expressions without having to explicitly introduce named functions

- Typical examples:

```
LAMBDA j: 0
LAMBDA i: table(i)
LAMBDA x,y: x + 2 * y
LAMBDA (p: prime): 2^p - 1
```

- Evaluates to a function of n arguments with a signature derived from the variable types and expression types
- Lambda expressions can be used wherever a function value of the appropriate type is used
  - As part of defining expressions for larger functions
  - As a value supplied to data structure update operations
  - As the function being applied to one or more arguments
  - Example: `(LAMBDA (p: prime): 2^p - 1)(3) = 7`
- Lambda expressions pop up a lot because of PVS's orientation toward function types and higher-order logic

Function Overriding

Another way to construct new function values is to override/update an existing function value to create a new one

- Basic form:

```
f WITH [(0) := 2, (1) := 3]
table WITH [(i) := g(i)]
matrix WITH [(i)(j) := x * y]
r WITH [`a := 1, `b(1)`c := 0]
f WITH [(-1) |-> g(0)]
```

- Evaluates to a new function formed from the original that differs on one or more elements of its domain
- `|->` form extends domain of function, resulting in a different type
- Useful for specifying state-changing operations on large data objects
- Meaning is best visualized by considering function update and then application:

```
(f WITH [(i) := a])(j) =
   IF i = j THEN a ELSE f(j) ENDIF
```

Record Operations

PVS has facilities for record construction, field selection, and updates

- Record construction:

```
(# ready := true, timestamp := T + 1, count := 0 #)
```

- Field selection is similar to the familiar `r.ready` notation from programming languages:

```
IF r`ready THEN r`timestamp ELSE 0 ENDIF
```

- Field selection is also possible using function application:

```
IF ready(r) THEN timestamp(r) ELSE 0 ENDIF
```

- Record update:

```
r WITH [ready := false, timestamp := current]
```

  - Evaluates to `r` with two of its fields updated as indicated

Tuple Operations

Tuple construction, field selection, and updates are similar to those of records

- Tuple construction:

```
(true, T + 1, 0)
```

- Tuple selection is similar to record field selection:

```
IF t`1 THEN t`2 ELSE 0 ENDIF
```

- Tuple update:

```
t WITH [`1 := false, `2 := current]
```

  - Evaluates to `t` with two of its components updated as indicated

LET and WHERE Expressions

Two expression types are used to introduce named subexpressions

- Basic form:

```
LET x = 2, y: nat = x * x IN f(x, y) + y
```

- LET variables are local to the LET expression
- Within the IN part, variables denote values as if the subexpressions were substituted in their place
- WHERE form is analogous:

```
f(x, y) + y WHERE x = 2, y: nat = x * x
```

- There is also a tuple form to implicitly name components:

```
LET (x, y, z) = t IN x + y * z
```

- LET and WHERE expressions are useful for modeling sequential computation steps
- LET is more typical but WHERE is useful with tables

Misc. Expressions

Several other expression types are available in PVS

- *Coercions* alert the typechecker to type membership

```
a/b :: int
```

  - Assuming b divides a

- Sets are represented in PVS as predicates over a base type

- Set expressions:

```
{n: int | n < 10}
```

  - Equivalent to `LAMBDA (n: int): n < 10`

- List constructors:

```
(: 1, 2, 3, 4 :)
```

  - Equivalent to `cons(1, cons(2, ... null))`

- String constants:
  - `"A character string"`

Pattern Matching on Data Types

A special construct is available for working with abstract data types

- The CASES construct enables a kind of "pattern matching" on DATATYPE-introduced values

```
CASES list OF
   cons(elt, rest): append(reverse(rest),
                           cons(elt, null))
   ELSE null
ENDCASES
```

- Allows conditional selection of alternative expressions
  - Based on the form of a value with respect to its DATATYPE definition
  - One clause per constructor

Name Resolution

When names have been imported from multiple theories, name conflicts or ambiguity may result

- The same name may be imported from different theories
- Or, the same name may be imported from different theory *instances*
- Three ways to reference "`name`" declared in theory "`thy`":
  1. `name`
  2. `name[params]`
  3. `thy[params].name`
- Method 1 works when there are no conflicts
- Method 2 works for some clashes
- Method 3 is guaranteed to be unambiguous

Function Declaration

Named functions are declared using the constant declaration mechanism

- A function is simply a constant whose type is a function type
- As with simple data constants, function declarations may be either interpreted or uninterpreted
- Typical uninterpreted function declarations:

```
abs(x): nat
max: [int, int -> int]
gcd(m, n): nat
ordered(s: num_list): bool
scalar_mult(a, (v: vector)): real
```

- Such undefined functions may be referenced freely in PVS specifications
  - But there is nothing to expand during proofs

Function Definition

Functions are *defined* by giving interpreted function declarations

- Typical function definitions:

```
abs(x): nat = IF x < 0 THEN -x ELSE x ENDIF
time(m: minute, s: second): nat = m * 60 + s
device_busy(d: control_block): bool = NOT d`ready
scalar_mult(a, V): vector = LAMBDA i: a * V(i)
```

- Type of defining expression must be contained in declared result type of function
- Result type may be any PVS type
- Function types allowed for arguments and result
- Recursive definitions allowed with special syntax provided
  - But no mutual recursion across two or more definitions
- Rules are designed to ensure *conservative extension* of theory
- *Macros* are a variant of constant/function declarations
  - They are expanded at typecheck time

Recursive Function Definitions

Recursive definitions have a special form

- Recursion must be signaled so the system can check for well-foundedness of the definition, i.e, that recursion is always bounded

```
factorial(n): RECURSIVE nat =
    IF n = 0 THEN 1 ELSE n * factorial(n-1) ENDIF
  MEASURE LAMBDA n: n
```

- A measure function must be provided
  - Measure must strictly decrease on every recursive call
  - Termination TCCs may be generated if this cannot be established
  - Shortcuts allowed for simple measures: `MEASURE n`
- A special form also exists to deal with DATATYPE situations
- *Inductive definitions* are a related concept

Formula Declarations

Various kinds of formulas may be included in a theory

- A formula declaration is a named logical formula (boolean expression)

```
transitive:  AXIOM x < y AND y < z => x < z
distrib_law: LEMMA x * (y + z) = x * y + x * z
friendly_skies: THEOREM
         mode(aircraft) = cruise IMPLIES
         altitude(aircraft) > 1000
```

- Formulas may contain free variables
  - PVS assumes the universal closure:

```
FORALL x,y,z: x * (y + z) = x * y + x * z
```

- Declared formulas may be submitted to the theorem prover
  - PVS tracks the proof status of formulas
- Multiple spellings available
  - LEMMA, THEOREM, CONJECTURE, etc.
  - All semantically equivalent except AXIOM and POSTULATE

Special Formulas about Types

PVS allows special formulas to specify type attributes of function applications

- Judgements are lemmas about (sub)types that get applied automatically during type checking
  - They can obviate many TCCs that would otherwise be generated
- Constant judgements can narrow the type of an expression

```
even_plus_even_is_even: JUDGEMENT +(e1,e2) HAS_TYPE even_int
odd_plus_even_is_odd:   JUDGEMENT +(o1,e2) HAS_TYPE odd_int
```

- Subtype judgements express type relationships

```
JUDGEMENT posrat SUBTYPE_OF nzrat
JUDGEMENT nzrat SUBTYPE_OF nzreal
```

- Possible interactions with various type conversion features
  - Extensions, restrictions, etc.