Expression Language Features of PVS

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

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 and access
- Tuple construction and 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:

    $$(\forall x \in D : f(x) = g(x)) \supset f = g$$

  - Logic notation:

    $P \supset Q$ means $P \Rightarrow Q$ $\quad (P \text{ implies } Q)$

Arithmetic Expressions

PVS has the usual assortment of arithmetic operations.

- Relational operators: `<, <=, >, >=`
- Binary operators: `+, -, *, /, ^`
- Unary operators: `-`
- Numeric constants are limited to integers and rationals.
  - Decimal point format is available.
  - Can bound or approximate reals using rational numbers.
  - Examples: `1/2, 22/7, 3.14, 0.621`
- 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.
  - Together ensure that exactly 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 are 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)    a - b    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.

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

Function Application (Cont'd)

- 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 argument types and expression types.
- The following declarations are equivalent:

```
square: [real-> real] = LAMBDA (x: real): x * x
square(x: real): real = x * x
```

Lambda Abstraction (Cont'd)

- 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.

- Examples of basic forms:

```
f WITH [0 := 2, 1 := 3]
f WITH [(0) := 2, (1) := 3]
table WITH [(i) := g(i)]
matrix WITH [(i)(j) := x * y]
```

- Each evaluates to a new function formed from the original that differs on one or more elements of its domain.
- A form using symbol |-> extends the domain of the function, resulting in a different type.

```
f WITH [(-1) |-> g(0)]
```

Function Overriding (Cont'd)

- Useful for specifying state-changing operations on large data objects.
- Meaning is best visualized by considering function update and then function application:

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

  - Some prover commands apply this reduction automatically.

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 (two forms allowable):

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

```
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
```

- Selection is also possible using built-in projection functions:

```
IF proj_1(t) THEN proj_2(t) ELSE 0 ENDIF
```

- Tuple update (two forms allowable):

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

```
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 name components implicitly:

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

- LET and WHERE expressions are useful for modeling sequential computation steps.

Misc. Expressions

Several other expression types are available in PVS.

- *Coercions* alert the typechecker to type membership.
- Example: `(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.

Extensible Syntax and Semantics

PVS supports several ways to enhance flexibility and expressibility.

- Function names may be overloaded.
  - Types of arguments are used to disambiguate function instances.
  - Predefined as well as user-defined functions may be overloaded.
  - Even infix operators such as + and * may be overloaded.
- Also, the identifier o is available as a user-definable operator.
  - Example: `fs1 o (fs2 o fs3) = (fs1 o fs2) o fs3`
- Several "outfix" operators are available as well.
  - Three bracket pairs: `[| |]` `(| |)` `{| |}`
  - Function definition example:

    ```
    [||] (a,b,c): real = (a + b + c) / 3
    ```
  - Use in an expression:

    ```
    avg_123: LEMMA  [| 1,2,3 |] = 2
    ```

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]
ordered(s: num_list): bool
```

- Note these are equivalent:

```
gcd: [nat, nat -> nat]
gcd(m: nat, n: nat): nat
```

Function Declaration (Cont'd)

- Note a subtle difference:

```
scalar_mult(a, v: vector): real
scalar_mult(a, (v: vector)): real
```

  - In the second case, the type of a is inherited from the theory.

- Undefined (uninterpreted) functions may be referenced freely in PVS specifications.

  - But there is nothing to expand during proofs.
  - This is perfectly fine and typical for abstract modeling.

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 function's result type.
- Result type may be any PVS type.
- Function types are allowed for arguments and result.
- Recursive definitions are allowed, with special syntax provided.
  - But no mutual recursion across two or more definitions.

Function Definition (Cont'd)

- Rules are designed to ensure *conservative extension* of theory.
  - Adding a function definition cannot make a theory inconsistent.
- *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 always terminates.

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

- A measure function M on one or more variables must be provided.
  - $M(n)$ must strictly decrease on every recursive call.
  - Termination TCCs may be generated if this cannot be established.
  - Shortcuts are 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 logical 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:

    ```
    distrib_law: LEMMA x * (y + z) = x * y + x * z
    ```

    is treated as:

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

Formula Declarations (Cont'd)

- Declared formulas may be submitted to the theorem prover.
  - PVS tracks the proof status of formulas.
  - Changing a formula marks its proof as needing to be rechecked.
- Multiple formula types or "spellings" are available.
  - LEMMA, THEOREM, CONJECTURE, etc.
  - All are semantically equivalent except AXIOM and POSTULATE.

Judgements: 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.
  - Many judgements are provided by the prelude.
  - Users can introduce their own.
- 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
```

Judgements (Cont'd)

- Subtype judgements express type relationships.

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

- There are possible interactions with various type conversion features.
  - Extensions, restrictions, etc.