Formal Methods for Secure Systems

Master of Science in Computer Engineering

Introduction to the PVS (Prototype Verification System) language

Andrea Domenici

Department of Information Engineering
University of Pisa, Italy

April 13, 2020

DISCLAIMER

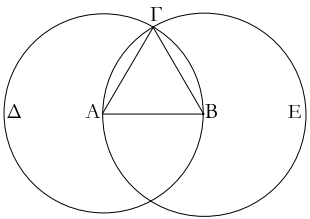
The PVS language is quite complex. Most constructs have several variants, which may just be alternative syntactic forms or have semantic differences.

Obviously, these slides cover only a part of the language.

+: one or more; *: zero or more

- <letter>: A...Z|a...z (ASCII)
- <digit>: 0...9
- <number>: <digit>+ | <digit>+ . <digit>+
- <identifier>: <letter> | (<letter>|<digit>|_|?)*
- <special symbol>: many!
- <reserved word>: <keyword> | <spelled-out operator>
- <keyword>>: <letter> | (<letter>|_)+
- <spelled-out operator>: AND | ANDTHEN | FALSE | IF | IFF | IMPLIES | NOT | o | OR | ORELSE | TRUE | WHEN | XOR
- <symbolic operator>: <special symbol>+

α΄



Ἐπὶ τῆς δοθείσης εὐθείας πεπερασμένης τρίγωνον ἰσόπλευρον συστήσασθαι.

Ἔστω ἡ δοθεῖσα εὐθεῖα πεπερασμένη ἡ ΑΒ.

Δεῖ δὴ ἐπὶ τῆς ΑΒ εὐθείας τρίγωνον ἰσόπλευρον συστήσασθαι.

Κέντρῳ μὲν τῷ Α διαστήματι δὲ τῷ ΑΒ κύκλος γεγράφθω ὁ ΒΓΔ, καὶ πάλιν κέντρῳ μὲν τῷ Β διαστήματι δὲ τῷ ΒΑ κύκλος γεγράφθω ὁ ΑΓΕ, καὶ ἀπὸ τοῦ Γ σημείου, καθ' ὃ τέμνουσιν ἀλλήλους οἱ κύκλοι, ἐπί τὰ Α, Β σημεῖα ἐπεζεύχθωσαν εὐθεῖαι αἱ ΓΑ, ΓΒ.

Καὶ ἐπεὶ τὸ Α σημεῖον κέντρον ἐστὶ τοῦ ΓΔΒ κύκλου, ἴση ἐστὶν ἡ ΑΓ τῇ ΑΒ· πάλιν, ἐπεὶ τὸ Β σημεῖον κέντρον ἐστὶ τοῦ ΓΑΕ κύκλου, ἴση ἐστὶν ἡ ΒΓ τῇ ΒΑ. ἐδείχθη δὲ καὶ ἡ ΓΑ τῇ ΑΒ ἴση· ἑκατέρα ἄρα τῶν ΓΑ, ΓΒ τῇ ΑΒ ἐστιν ἴση. τὰ δὲ τῷ αὐτῷ ἴσα καὶ ἀλλήλοις ἐστὶν ἴσα· καὶ ἡ ΓΑ ἄρα τῇ ΓΒ ἐστιν ἴση· αἱ τρεῖς ἄρα αἱ ΓΑ, ΑΒ, ΒΓ ἴσαι ἀλλήλαις εἰσίν.

Ἰσόπλευρον ἄρα ἐστὶ τὸ ΑΒΓ τρίγωνον. καὶ συνέσταται ἐπὶ τῆς δοθείσης εὐθείας πεπερασμένης τῆς ΑΒ· ὅπερ ἔδει ποιῆσαι.

NON-PARAMETRIC PVS THEORIES

A PVS specification is composed of one or more *theories*.

```
<name>: THEORY
BEGIN
    <imports>
    <type declarations>
    <constant and variable declarations>
    <formulae>
END <name>
```

Constructs of different classes may be interleaved (e.g., a type declaration may follow a variable declaration), but every symbol must be declared before it is used.

Note: *Datatypes* are a shorthand for a special kind of theories. They may occur both outside or inside plain theories.

Context and IMPORTING clauses

The *context* of a theory is the set of theories directly visible from that theory.

The default context of a theory T consists of

- the theories defined in the *prelude* (part of the PVS installation);
- the theories preceding T in the same file;
- the theories defined in the other PVS files in the same directory containing T's file.

IMPORTING clauses are used to make the definitions of some theories visible to the importing theory. If an imported library belongs to the context pf the importing theory, the clause has the form

```
IMPORTING <theory name>
```

Otherwise, the name of the *library* containing the imported theory must be given:

```
IMPORTING <library name>@<theory name>
```

A library is a filesystem directory containing PVS files. The environment variable `PVS_LIBRARY_PATH` specifies paths to search for variables.

TYPE SYSTEM: type expressions and type declarations

A *type expression* defines the structure of a type.
A type expression is a type name or an expression of the forms shown in the next slides.
Type expressions occur in type, constant, and variable *global* declarations:

```
<type name>: TYPE <type expression>
<constant name>: <type expression>
<variable name>: VAR <type expression>
```

Type expressions also occur in *local* variable declarations, i.e., within formulae, e.g.,

```
a_theorem: THEOREM
    FORALL (x: real): ...
```

TYPE SYSTEM: base types (1)

The pre-defined base types are the Booleans (`boolean`, `bool`), the real numbers (`real`), and the subtypes of the real numbers.

The Booleans are the set `{FALSE, TRUE}`.

The type `real` represents the *mathematical* concept of real numbers. The PVS reals have no limitations on range and precision.

The `reals` can be *non-zero* (`nonzero_real`, `nzreal`), *non-negative* (`nonneg_real`, `nnreal`), *non-positive* (`nonpos_real`, `npreal`), *positive* (`posreal`), or *negative* (`negreal`).

The `rationals` are a subset of `reals` and can be *non-zero* (`nonzero_rational`, `nzrat`), *non-negative* (`nonneg_rat`, `nnrat`), *non-positive* (`nonpos_rat`, `nprat`), *positive* (`posrat`), or *negative* (`negrat`).

TYPE SYSTEM: base types (2)

The integers (`integer`, `int`) are a subset of `rationals` and can be *non-zero* (`nonzero_int`, `nzint`), *non-negative* (`nonneg_int`), *non-positive* (`nonpos_int`), *positive* (`posint`), or *negative* (`negint`).

The types of even and odd integers are `even_int` and `odd_int`, respectively.

The types `subrange(i, j)` represent the integers in the intervals $[i..j]$, and the types `upfrom(i)` and `above(i)` represent the integers in the intervals $[i..\infty)$ and $[i+1..\infty)$, respectively.

The natural numbers (`naturalnumber`, `nat`) coincide with the type `nonneg_int`. The types `upto(i)` and `below(i)` represent the integers in the intervals $[0..i]$ and $[0..i-1]$, respectively.

TYPE SYSTEM: Type constructors

Enumerations: `flag: TYPE = {red, black, white, green}`

Tuples: `triple: TYPE = [nat, flag, real]`

Records: `point: TYPE = [# x: real, y: real #]`

Subtypes: `posnat: TYPE = {x: nat | x>0}`

If `p` is a predicate (e.g., over `nat`),

```
p_type: TYPE = {x: nat | p(x)}
```

is equivalent to

```
p_type: TYPE = (p).
```

TYPE SYSTEM: function types

The type of a function is specified by its domain and range, e.g.,

```
int2real: TYPE = [int -> real]
intrat2real: TYPE = [int, rational -> real]
int2intint: TYPE = [int -> [int, int]]
```

The following two type expressions are equivalent:

```
[int, rational -> real]
[[int, rational] -> real]
```

If `T` is any type, the following types are the same type:

```
[T -> bool]
pred[T]
setof[T]
```

TYPE SYSTEM: dependent types

Function, tuple, and record types may be *dependent*, i.e., some parts of their definition may depend on a preceding part.

For example, the remainder operator is a function of two natural numbers (numerator and denominator) mapped to a natural number (remainder).

The numerator can be any natural number, the denominator must be non-zero, and the remainder must be strictly less than the denominator. These constraints can be expressed defining the type of the remainder operator as

```
[nat, d: {n: nat | n /= 0} -> {r: nat | r < d}]
```

Note: the type of the PVS `rem` operator is defined in a different way.

Variable DECLARATIONS (1)

Variables can be declared globally or locally, in *binding expressions* (e.g., FORALL).

```
x: VAR real                                % global declaration

th1: THEOREM
    abs(x) >= 0

th2: THEOREM
    FORALL (x: real): abs(x) >= 0 % local declaration
```

Note: the two theorems above are equivalent.

Variable DECLARATIONS (2)

Local declarations hide global ones

```
x: VAR real                          % x is a real
th3: THEOREM
    FORALL (x: nat): abs(x) >= 0 % x is a nat
```

Declarations in an inner scope hide those in an outer one:

```
p(x: nat): bool = x > 0
q(x: int): bool = x < 0
th4: THEOREM
FORALL (x: int):
    (EXISTS (x: nat): p(x))    % x is a nat
    AND
    q(x)                    % x is an int
```

Note: it is better to use different variables in each scope.

Constant DECLARATIONS

User-defined constants may be declared specifying only their type (*uninterpreted* constants) or their type and value (*interpreted* constants).

Axioms can be used to specify properties of uninterpreted cnstants. Note that many properties may be specified more clearly and safely by declaring the constant as belonging to a pre-defined or user-defined subtype.

Function constants: named functions

Named functions have this familiar syntax:

```
<function name>(<formal parameter declarations>): <type expression>
    = <expression>
```

For example:

```
incr(n: int): int = n + 1
foo(n, m: int): int = n + 2*m
```

A named function can be *applied* to *actual argumnts*, i.e., the formal parameters are replaced by the actual arguments and the resulting expression is evaluated, e.g.,

```
funct_appl: LEMMA
    foo(1, 2) = 5
```

Function constants: recursive functions

Recursive fubction constants are declared as follows:

```
<function name>(<formal parameter declarations>):
    RECURSIVE <type expression> = <expression>
    MEASURE <measure expression>
```

The measure expression is used to check if recursion terminates. In most cases, this expression reduces to the variable on which the function recurs, e.g.,

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

Note: This treatment of the concept of measure is vastly oversimplified. Please RTFM.

LAMBDA expressions (1)

A λ-expression defines an *anonymous* function, e.g.,

```
LAMBDA (x: int): x + 1
```

where x is the *formal parameter* of the function.

The following example shows the application of a λ-expression:

```
lambda_appl: LEMMA
    (LAMBDA (x: int): x + 1)(1) = 2
```

A λ-expression may contain occurrences of variables that are not formal parameters, but such variables must be bound in some outer environment, such as the theory's top level, a quantified expression, or another λ-expression, e.g.,

```
FORALL (x: int):
    x > 0 IMPLIES (LAMBDA (y: nat): x + y)(z) > 0

LAMBDA (x: int): LAMBDA (y: int): x + y
```

LAMBDA expressions (2)

λ-expressions can be used to define functions having function types as domain:

```
bar(n: int): [int -> int] = LAMBDA (m: int) n + 2*m
curry: LEMMA
    bar(1)(2) = 5
```

Note: Transforming an *n*-parameter function to *n* one-parameter functions is called *currying*.

Haskell Brooks Curry (1900 – 1982)

By Gleb.svechnikov - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=58344593

Function variables

Function variables can be declared locally or globally:

```
funct_var: LEMMA
    FORALL (f: [real -> nat], x: real): f(x) >= 0
```

Globally declared function variables can be bound to λ-expressions, using an axiom:

```
    g: [int -> [int -> int]]

    g_def: AXIOM
        g = LAMBDA (m: int): LAMBDA (n: int): m + n
```

A named function constant such as

```
    g1(m, n: int): int = m + n
```

is equivalent to (*but different from*) g, when the latter is applied to two arguments:

```
funct_var_const: LEMMA
    FORALL (m, n: int): g(m)(n) = g1(m, n)
```

FORMULAE

As seen in various examples above, a formula declaration has the following syntax:

```
<formula decln> > ::= <formula name> : <formula keyword>
                            <boolean expression>

<formula keyword> ::= AXIOM | ASSUMPTION | OBLIGATION
                    | <theorem keyword>
<theorem keyword> ::= THEOREM | LEMMA | COROLLARY
                    | PROPOSITION | FORMULA | ...
```

"AXIOM" declares (guess what?) axioms. Theorem keyword declare formulae that must be proved and they are all synonims,

"ASSUMPTION" and "OBLIGATION" will be dealt with later on.

PARAMETRIC PVS THEORIES (1)

PVS theories can be *parametric*. Consider, e.g., the group theory shown in the introductory lecture:

```
group : THEORY
BEGIN
  G : TYPE+       % uninterpreted, nonempty
  e : G           % neutral element
  i : [G -> G]    % inverse
  * : [G,G -> G] % binary operation
  x,y,z : VAR G
  associative : AXIOM
    (x * y) * z = x * (y * z)
    % other axioms ...
  inverse_associative : THEOREM
    i(x) * (x * y) = y
END group
```

Parametric PVS theories (2)

The `group` theory defines the abstract *group* concepts and can be used to prove its properties, but it cannot be used to reason about a concrete group such as the integers with addition. This can be fixed by turning `G`, `e`, `i`, and `*` into parameters:

```
param_group[G: TYPE+, e: G, i: [G -> G], * : [G,G -> G]]: THEORY
BEGIN
  x,y,z : VAR G
  associative : AXIOM
    (x * y) * z = x * (y * z)
    % other axioms ...
  inverse_associative : THEOREM
    i(x) * (x * y) = y
END param_group
```

Parametric PVS theories (3)

Theory `param_group` can then be used as in the following example, where we prove that the *minus* integer operator has the same property as the `i` operator in the `param_group` theory.

```
integer_group[int, 0, -, +]: THEORY
BEGIN
    IMPORTING param_group[int, 0, -, +]
    x,y,z : VAR int
    minus_associative: THEOREM
          -x + (x + y) = y
END integer_group
```

Note: Obviously, we can use `int`, `0`, `-`, and `+` without declaring them, because the are declared in the prelude theories.

Parametric PVS theories: assumptions

In the `param_group` theory, the axioms state properties of constants and operators of type `G`, but there is no guarantee that the formal parameter `G` will not be instantiated with an actual type that does not satisfy those axioms.

Assumptions are properties expected of the actual arguments of a parametric theory, and must be *discharged* (proved) when the theory is used. For example:

```
param_group[G: TYPE+, e: G, i: [G -> G], * : [G,G -> G]]: THEORY
BEGIN
  x,y,z : VAR G
  ASSUMING
      associative : ASSUMPTION
        (x * y) * z = x * (y * z)
        % other assumptions ...
  ENDASSUMING
  inverse_associative : THEOREM
    i(x) * (x * y) = y
END param_group
```

Parametric PVS theories: syntax

```
<name> [<formal parameters>] : THEORY
BEGIN
    <assuming part>
    <imports>
    <type declarations>
    <constant and variable declarations>
    <formulae>
END <name>

<assuming part> ::= ASSUMING <declaration>* <assumption>+ ENDASSUMING
<assumption> ::= <formula name> ASSUMPTION <boolean expression>
```

EXPRESSIONS

Boolean operators

Boolean operators

name	symbol	spelled out	priority	associative	overloading
quantifier		FORALL	1	no	no
		EXISTS	1	no	no
equivalence	<=>	IFF	2	right	yes
implication	=>	IMPLIES	3	right	yes
disjunction	\/	OR	4	right	yes
conjunction	&	AND	5	right	yes
negation	~	NOT	6	no	yes

Overloading (redefinition) is allowed both for symbolic and spelled out operators, but *redefining a symbol does not redefine its matching spelled out keyword, and viceversa*.

Arithmetic, relational, and composition operators

The arithmetic operators are - (unary and binary), +, *, \, and ^ (exponentiation).

The relational operators are <, <=, >, and >=.

The equality and inequality operators are = and /=.

The function composition operator is o (lowercase letter 'o').

Tuple expressions

A tuple of type `[t_1, t_2, ..., t_n]` has the form

```
(e_1, e_2, ..., e_n)
```

where the `e`'s are expressions.

Each *i*-th expression is referred to by the *projection* operator `` `i ``.

The first character (`) of a projection operator is *backtick*, or grave accent (ASCII 96).

Record expressions

A record of type `[# a_1: t_1, a_2: t_2, ..., a_n: t_n #]` has the form

```
(# a_1 := e_1, a_2 := e_2, ..., a_n := e_n #)
```

where the e's are expressions.

Each *i*-th expression is referred to by the *accessor* operator ‘`a_i`.

Overriding

Overriding means creating a new record, tuple, or function by changing part of the original one, which remains unchanged. In the following code, For example:

```
complex: TYPE = [# % record type
    r: real,
    i: real
    #]
x: complex = (# r := 1.0, i := 2.0 #)
y: complex = x WITH [ r := -1.0 ]
```

y denotes the complex value $(-1.0, 2.0)$. Note that x is left unchanged.

IF expressions (1)

An IF expression has the form

```
IF <boolean expression>
THEN <expr_1>
ELSE <expr_2>
ENDIF
```

where <expr_1> and <expr_2> must have the same type.

Nested IF expressions can be expressed with ELSIF clauses:

```
IF <boolean expression 1>
THEN <expr_1>
ELSIF <boolean expression 2>
THEN <expr_2>
    ...
ELSIF <boolean expression n>
THEN <expr_n>
ELSE <expr_(n + 1)>
ENDIF
```

IF expressions (2)

The IF ternary operator is treated by the prover as if defined as follows:

```
if_theory[T: TYPE+]: THEORY
BEGIN
    x, y: T

    IF: [bool, T, T -> T]

    if_def: AXIOM
        IF(TRUE, x, y) = x
        AND
        IF(FALSE, x, y) = y
END if_theory
```

Note: the definition of the IF operator is actually built into the prover.

COND expressions

COND expressions are similar to the *switch* statements of imperative programming languages:

```
COND
     <boolean expression 1> -> <expr_1>,
     <boolean expression 2> -> <expr_2>,
        ...
     <boolean expression n> -> <expr_n>
ENDCOND
```

The disjunction of all Boolean expressions must be a tautology (*coverage* condition), i.e., *at least* one of them can be satisfied, and they must be pairwise disjoint, i.e., *only* one of them can be satisfied (*disjointness* condition).

LET expressions

LET expressions are used to introduce new variables, equate them to expressions, and accordingly expand their occurrences in the expression following the IN keyword:

```
LET <variable 1> = <expr_1>,
    <variable 2> = <expr_2>,
        ...
    <variable n> = <expr_n>
IN <expression>
```

For example:

```
LET v = x + y,
    w = x - y
IN v*w
```

Note: each expression in the LET part can refer to variables introduced the preceding equations.