PVS Tutorial, FM99

John Rushby, Dave Stringer-Calvert, and N. Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

These are the examples that will be used during the first part of the tutorial. They (and several others) are available by following the `ExamplesandTutorials` link from the PVS home page at `http://pvs.csl.sri.com`. You can also download the PVS system from there. This document is not intended to be self-contained: it is intended to help you follow along during the tutorial. If you want to examine the proofs for the lemmas and theorems appearing here, load the appropriate example file into PVS, position the cursor in the formula whose proof you wish to examine, and give the command `M-x step`. The two characters `tab 1` will then step you through the proof one command at a time.

1 Sum

This example is used to introduce the look and feel of PVS. The recursive function `sum_nats` takes a natural number `n` as its argument and returns the sum of the natural numbers up to `n`.

```
sum: THEORY
BEGIN

  sum_nats(n: nat): RECURSIVE nat =
    IF n=0 THEN 0 ELSE n+sum_nats(n-1) ENDIF
  MEASURE n

  test: LEMMA sum_nats(3) = 6

  closed_form: THEOREM FORALL (n:nat): sum_nats(n) = n*(n+1)/2

  bigtest: LEMMA    sum_nats(100)    = 5050
  biggertest: LEMMA sum_nats(200)    = 20100
  hugetest: LEMMA   sum_nats(100000) = 5000050000

END sum
```

Because it is recursive, we must give a `measure` to help establish termination. Proof obligations called Typecheck Correctness Conditions (TCCs) are generated to ensure that the measure decreases across recursive calls, and also that the expression `n-1` is well-defined (i.e., that it is not negative).

We can test this specification by expanding the definition several times to evaluate small values such as `sum_nats(3)`. Then we can use the prover to establish (by induction) the closed-form expression for this sum.

If we try testing larger and larger values, we see that execution by theorem proving is not very efficient: it takes several seconds to evaluate `sum_nats(100)`. PVS has a ground evaluator this purpose; it compiles PVS into Lisp that can easily evaluate `sum_nats(100000)`.

2 Summations

This example demonstrates some of the higher-order features of PVS. The function `summation` takes another function as its argument and sums the value of that function over the natural numbers up to `n`. The function `id[nat]` is a PVS prelude (built-in) function that specifies the identity function on the natural numbers, so that `summation(id[nat], n)` should be the same as `sum_nats(n)`. We prove this fact, and also the closed-form expressions for sums of squares and cubes.

```
summations: THEORY
  BEGIN

   n: VAR nat
   f, g: VAR [nat -> real]

   summation(f, n): RECURSIVE real =
    IF n = 0
      THEN f(0)
      ELSE f(n) + summation(f, n - 1)
    ENDIF
   MEASURE n

  IMPORTING sum
  summation_nats: LEMMA  summation(id[nat], n) = sum_nats(n)
  summation_nats_closed_form: LEMMA
                           summation(id[nat], n) = n*(n+1)/2
...continued
```

```
...continuation

  r: VAR real
  square(r: real): real = r*r

  summation_squares: LEMMA
    summation(square, n) = n * (n + 1) * (2*n + 1) / 6

  cube(r): real = r*r*r

  summation_cubes: LEMMA
    summation(cube, n) = n*n*(n+1)*(n+1)/4
...continued
```

To illustrate additional proof commands, we also prove that the sum of cubes is equal to the square of the sum of naturals.

```
...continuation

  summation_of_cubes_alt: LEMMA
    summation(cube, n) = square(summation(id[nat],n))

  summation_of_cubes_alt2: LEMMA
    summation(cube, n) = square(summation(id[nat],n))

  summation_of_sum: LEMMA
    summation((lambda n: f(n) + g(n)), n) =
      summation(f, n) + summation(g, n)

  subtype_test: LEMMA
    summation(square, summation(id[nat],3)) = 91

  summation_of_nat_is_nat: JUDGEMENT
     summation(g:[nat->nat], n) HAS_TYPE nat

  judgement_test: LEMMA
     summation(square, summation(id[nat], 3)) = 91

END summations
```

The `summations` function is defined over the reals and returns a real value, so if we try to use `summation(id[nat],3)` as the `n` in `summation(square, n)` we encounter a TCC. However, the summation of a `nat`-valued function is always a `nat` and it is better to establish this fact once and for all. We use this to illustrate the use of PVS type judgements.

3 Language Interpreter

The next example introduces PVS Abstract Data Types. We will define a simple programming language for a machine whose memory can store integers and is addressed by numbers in the range `1..1000`.

```
memories: THEORY
BEGIN
  n: nat = 1000
  addrs: TYPE = upto(n)
  memory: TYPE = [addrs -> int]
END memories
...continued
```

Our language has expressions consisting of literal integer constants, “variables” that denote a memory address, and (recursively) sums, differences, and negations.

```
...continuation
exprs: DATATYPE
BEGIN
 IMPORTING memories
 const(n: int): num?
 varbl(a: addrs): vbl?
 +(x,y: exprs): sum?
 -(x,y: exprs): diff?
 ~(x: exprs): minus?
END exprs
...continued
```

Statements consist of assignments, sequential composition, if-then-else, and primitive “for” loops that executed a fixed number of times given by an explicit natural number.

```
...continuation
statements: DATATYPE
BEGIN
 IMPORTING memories, exprs
  assign(a:addrs, e:exprs): assign?
  seq(a,b: statements): seq?
  ifelse(t: exprs, i,e:statements): ifelse?
  for(l: nat, b:statements): for?
END statements
```

Notice that `exprs` and `statements` are not mutually recursive; if they were, we would have to define them together in a single datatype with subtypes. here is an example

```
expression: DATATYPE WITH SUBTYPES term, typ
BEGIN
  base_type(n:nat): base_type? : typ
  funtype(dom: typ, ran: typ): funtype? : typ
  variable(n:nat): variable? : term
  number(num:nat): number? : term
  lam(v: (variable?), ty: typ, ex: term): lam? : term
  app(op: term, arg: term): app? : term
END expression
```

We define the semantics of simple `exprs` in the context of a given memory by means of an interpreter function `valof`. The subterm ordering predicate << on `exprs` is used to establish termination.

```
eval: THEORY
BEGIN
  IMPORTING statements

  valof(v: exprs)(mem: memory): RECURSIVE int =
    CASES v OF
      const(n): n,
      varbl(a): mem(a),
      +(x,y):   valof(x)(mem) + valof(y)(mem),
      -(x,y):   valof(x)(mem) - valof(y)(mem),
      ~(x):     - valof(x)(mem)
    ENDCASES
  MEASURE  v BY <<
...continued
```

We can test our specification by evaluating some simple expressions. The first two, `test1` and `test2` mean the same thing: the latter uses the infix and prefix forms of the subtraction and unary minus functions. We can avoid having to use the constructor `const` each time by specifying it as a `conversion`; if we also specify `varbl` as a `conversion` then this is preferred over `const` (because it comes later) and `test4` does not mean the same as `test3`.

```
...continuation
  arb: memory

  test1: LEMMA valof(-(const(3), ~(const(4))))(arb) = 7
  test2: LEMMA valof(const(3) - ~const(4))(arb) = 7

  CONVERSION const
  test3: LEMMA valof(3 - ~4)(arb) = 7

  CONVERSION varbl
  test4:  LEMMA valof(3 - ~4)(arb) = 7
  test4a: LEMMA valof(3 - ~4)(arb with [(3):=12, (4):=-5]) = 7
...continued
```

The logically next step is to define the semantics of `statements`, but first we must introduce a function that can be used as a measure for that recursive definition.

```
...continuation
  runtime(s: statements): RECURSIVE posnat =
  CASES s OF
    assign(a, e):  1,
    seq(a, b):     runtime(a) + runtime(b),
    ifelse(t,i,e): max(runtime(i),runtime(e))+1,
    for(l,b):      l*runtime(b)+1
  ENDCASES
  MEASURE s BY <<

  exec(s: statements)(mem: memory): RECURSIVE memory =
  CASES s OF
    assign(a, e):  mem with [(a) := valof(e)(mem)],
    seq(a, b):     exec(b)(exec(a)(mem)),
    ifelse(t,i,e): IF valof(t)(mem) /= 0 THEN exec(i)(mem)
                   ELSE exec(i)(mem) ENDIF,
    for(l,b):      IF l = 0 then mem
                   ELSE exec(for(l-1,b))(exec(b)(mem)) ENDIF
  ENDCASES
  MEASURE runtime(s)
...continued
```

We can test these definitions by evaluating some simple statements, and then a program that sums the first `j` natural numbers.

```
...continuation

  init: memory = id[addrs]

  test5: LEMMA
    valof(varbl(3))(exec(assign(3, -(3, ~(4))))(init)) = 7
  test5a: LEMMA
    valof(3)(exec(assign(3, 3 - ~4))(init)) = 7

  zero: memory = 0 % K conversion

  test_sum: LEMMA LET j = 10 IN
    valof(0)(exec(
         for(j+1,seq(assign(0, varbl(0) + varbl(1)),
                     assign(1, varbl(1) + const(1)))))(zero))
      = sum_nats(j)

...continued
```

We can evaluate the expression in `test_sum` for `j = 10` using rewriting, but using the PVS ground evaluator we can do it for `j = 100000` in just a few seconds.

Finally, we prove that the program does indeed compute the same function as `sum_nat`; first we prove the loop invariant, then the desired correctness theorem.

```
...continuation

  program_prop_lemma: LEMMA  FORALL (j:nat),(m:memory):
    valof(0)(exec(
          for(j+1,seq(assign(0, varbl(0) + varbl(1)),
                      assign(1, varbl(1) + const(1)))))(m)) =
          sum_nats(j) + m(0) + (j+1)*m(1)

  program_prop: THEOREM  FORALL (j:nat):
    valof(0)(exec(
          for(j+1,seq(assign(0, varbl(0) + varbl(1)),
                      assign(1, varbl(1) + const(1)))))(zero))
      = sum_nats(j)

END eval
```

That concludes this part of the tutorial. The second part will demonstrate model checking, abstraction, and other more advanced or recent capabilities.