Tutorial Introduction to PVS: A Verification System for Critical Applications

César Muñoz
Cesar.A.Munoz@nasa.gov
NASA Langley Research Center
Hampton, VA 23681-2199, USA

October 22, 2009

Abstract

PVS (Prototype Verification System)[1] is an interactive environment for the specification and verification of systems. PVS provides a strongly typed specification language, which is based on Higher-Order Logic. The type system of PVS supports: sub-typing, dependent-types, abstract data types, parametric types, records, unions, and tuples. The PVS theorem prover includes decision procedures for a variety of theories such as linear arithmetic, propositional logic, and temporal logic.

This seminar will provide a gentle introduction to the basic and advanced features of PVS, including: theory interpretations, real number proving, batch proving, rapid prototyping, and strategy development. All these features are illustrated with simple examples and exercises.

[1] http://pvs.csl.sri.com

Theory Interpretations

Consistency Relative to PVS

César A. Muñoz

NASA Langley Research Center
Cesar.A.Munoz@nasa.gov

PVS Class 2010

Logic 101

- A *(formal) system* is inconsistent if we can prove both A and $\neg A$.
- A consistency proof is hard. It is easier to prove that a system is consistent relative to another system (believed to be consistent itself).
- Relative consistency is shown by exhibiting model.

Inconsistency, It Can Happen to You

- This axiom is found in a highly referenced paper written in 1999:

```
C     : [real-> int]
MyAx  : AXIOM
  FORALL(x,y:real): x < y IMPLIES C(x) < C(y)
```

- The theory is inconsistent as `MyAx` implies that real numbers are enumerable.

- The inconsistency was revealed by L. Pike using *theory interpretations*.

Theory Interpretations in PVS

- A mechanism to construct models of axiomatic PVS theories, by instantiating uninterpreted types and constants.

- Consistency relative to the PVS logic can be shown via theory interpretations.

Is This Theory Consistent?

(Relative to the PVS System)

```
myTh : THEORY
BEGIN
  C    : [real-> int]
  x,y  : VAR real

  MyAx : AXIOM
    x < y IMPLIES C(x) <= C(y)
END myTh
```

For the Impatient

```
myTh : THEORY
BEGIN
  C    : [real-> int]
  x,y  : VAR real

  MyAx : AXIOM
    x < y IMPLIES
    C(x) <= C(y)
END myTh
```

```
myThi : THEORY
BEGIN
  IMPORTING myTh{{
     C(x:real) := floor(x)
  }}
END myThi

IMP_myTh_MyAx_TCC1: OBLIGATION
  FORALL (x, y: real): x < y
  IMPLIES floor(x) <= floor(y);
```

Isn't that What Theory Parameters are for?

```
myThp[C:[real->int]] : THEORY
BEGIN
  MyAx : AXIOM
    FORALL(x,y:real): x < y
    IMPLIES C(x) <= C(y)
END myThp

myThpi : THEORY
BEGIN
  IMPORTING myThp[floor]
END myThpi
```

No. Theory parameters do not generate proof obligations for the axioms in the source theory.

What is Wrong with myThp1 or myThp2 ?

```
myThp1[C:[real->int]] : THEORY
BEGIN
  ASSUMING
    MyAx : ASSUMPTION
      FORALL(x,y:real): x < y
      IMPLIES C(x) <= C(y)
  ENDASSUMING
END myThp1

myThp2[C:{c:[real->int]| FORALL(x,y:real):
       x < y IMPLIES c(x) <= c(y)}] : THEORY
BEGIN
END myThp2
```

Nothing.

Theory Parameters vs. Theory Interpretation

Theory interpretations largely subsume theory parameters, but

- Theory parameters are intended for the specification of a family of problems.

- Theory interpretations are intended for
  - Checking the consistency of an axiomatic specification.
  - Reification of an abstract data type.
  - Animation of an axiomatic specification.

I. Consistency Checking of an Axiomatic Specification

```
th : THEORY
BEGIN
  T : TYPE+
  i : T
  o : [[T,T]->T]
  x,y,z  : VAR T

  id    : AXIOM x o i = x
  assoc : AXIOM (x o y) o z = x o (y o z)
  inv   : AXIOM EXISTS(y): x o y = i AND y o x = i

  di : LEMMA
    i o x = x
END th
```

Return ThAsParam Return ThAsDecl

A Model or Two (Via Theory Abbreviations)

```
thi : THEORY
BEGIN

  IMPORTING th{{ T:=real,   i:=0, o(a,b:real  ):=a+b }}
               AS th0,
             th{{ T:=nzreal, i:=1, o(a,b:nzreal):=a*b }}
               AS th1

END thi
```

Proof Obligations as TCCs

```
h0_id_TCC1: OBLIGATION FORALL (x:real): x+0 = x;

th0_assoc_TCC1: OBLIGATION FORALL (x,y,z:real):
  x+y+z = x+(y+z);

th0_inv_TCC1: OBLIGATION FORALL (x:real):
  EXISTS (y:real): x+y = 0 AND y+x = 0;

th1_id_TCC1: OBLIGATION FORALL (x:nzreal): x*1 = x;

th1_assoc_TCC1: OBLIGATION FORALL (x,y,z:nzreal):
  x*y*z = x*(y*z);

th1_inv_TCC1: OBLIGATION FORALL (x:nzreal):
  EXISTS (y:nzreal): x*y = 1 AND y*x = 1;
```

To Be or Not to Be Consistent

- Claim: The theory `th` is consistent if TCCs in `thi` can be discharged.
- Remark: The above claim is made at the level of the PVS meta-theory, i.e., it is an external observation rather than a fact formally specified/proven in PVS.
- Note: No TCCs will be generated for an axiom

  ```
  foo : AXIOM 1=0
  ```

  in `th`.
- Question: Why ?

II. Reification

Process of making a concrete type from an abstract data type.

Reminder:

- Abstract data types in PVS, i.e., `DATATYPE`s, are axiomatically defined.
- Enumeration types are abstract data types.

Enumeration Types are Abstract Data Types

```
states : THEORY
BEGIN
  State : TYPE = {idle,waiting,running}
END states

states_as_nat : THEORY
BEGIN
  NatState : TYPE = below[3]
  n        : VAR NatState
  IMPORTING states{{ State       := NatState,
                     idle?(n)    := n=0,
                     waiting?(n) := n=1,
                     running?(n) := n=2,
                     idle        := 0,
                     waiting     := 1,
                     running     := 2}}
END states_as_nat
```

Proof Obligations

```
IMP_states_State_inclusive_TCC1: OBLIGATION
  FORALL (State_var:NatState):
    State_var = 0 OR State_var = 1 OR State_var = 2;

IMP_states_State_induction_TCC1: OBLIGATION
  FORALL (p:[NatState -> boolean]):
    p(0) AND p(1) AND p(2) IMPLIES
    (FORALL (State_var: NatState): p(State_var));
```

Note that `IMP_states_State_induction_TCC1` becomes unprovable if `NatState = nat`.

III. Animation of Axiomatic Specifications

Animation is the execution of a specification to validate its intended semantics.

- Animations in PVS are mostly performed in the Ground Evaluator.
- PVSio is a PVS package that re-implements the interface to the Ground Evaluator: `http://research.nianet.org/~munoz/PVSio`.
- Wait for the talk on PVSio.

Lost in Translation?

- The Emacs command `M-x ppti` displays in a new buffer the result of a theory interpretation.
- Technical Report: *Theory Interpretations in PVS*, S. Owre and N. Shankar, SRI-CSL-01-01. Available from `http://pvs.csl.sri.com/documentation.shtml`.
- PVS Release notes available from http://pvs.csl.sri.com/download.shtml.

Advanced Material

IV. Reification of ADTs

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

Lists as Arrays

```
list_as_array[T:TYPE] : THEORY
BEGIN

  List : TYPE = [#
    length : nat,
    elems  : [below(length)->T]
  #]

  l : VAR List
  t : VAR T
```

Lists Constructors

```
  Null?(l):bool = length(l) = 0

  Null : List = (#
    length := 0,
    elems  := LAMBDA(x:below(0)):epsilon(emptyset[T])
  #)

  Cons?(l):bool = not Null?(l)

  Cons(t,l): List = l WITH [
    `length          := l`length+1,
    `elems(l`length) := t
  ]
```

Interpretation

```
  IMPORTING list[T]{{ list  := List,
                      null? := Null?,
                      cons? := Cons?,
                      null  := Null,
                      cons  := Cons }}
END list_as_arrays
```

PVS Lists are Consistent

```
IMP_list_TCC1: OBLIGATION
  Null?(Null);

IMP_list_TCC2: OBLIGATION
  FORALL (x1: [T, List]):
    Cons?(Cons(x1));

IMP_list_list_null_extensionality_TCC1: OBLIGATION
  FORALL (null?_var:  {x: List | Null?(x)},
          null?_var2: {x: List | Null?(x)}):
    null?_var = null?_var2;
```

Note that the record type where `elems : [nat->T]` does not directly yield a model of `list[T]`.*

*In that case, the model has to be constructed using quotient types.

V. Theories as Parameters

Assume that we want to extend the theory th with a definition for the inverse function:

```
thx [t:THEORY th] : THEORY
BEGIN
  inverse(x:T):{y:T | x o y = i}
END thx
```

To provide an interpretation:

```
thxi : THEORY
BEGIN
  IMPORTING thx[th{{T:=nzreal,i:=1,o(a,b:nzreal):=a*b}}]

  inv_def : LEMMA
    FORALL(a:nzreal): inverse(a) = 1/a
END thxi
```

What is Wrong with thx2?

```
thx2 : THEORY
BEGIN
  IMPORTING th

  inverse(x:T):{y:T | x o y = i}
END thx2
```

Noting. But thx2 does not provide a mechanism to construct an interpretation of th.

VI. Theory Declarations

Assume that we want define a theory like th but with an extra commutativity axiom:

```
thax : THEORY
BEGIN
   t : THEORY = th

   commutativity : AXIOM
     FORALL(x,y:T): x o y = y o x
END thax
```

To provide an interpretation:

```
thaxi : THEORY
BEGIN
  IMPORTING
  thax{{t := th {{ T:=nzreal,i:=1,o(a,b:nzreal):=a*b}} }}
END thaxi
```

What is Wrong with thax2?

```
thax2[t: THEORY th] : THEORY
BEGIN
   commutativity : AXIOM
     FORALL(x,y:T): x o y = y o x
END thax2

thaxi2 : THEORY
BEGIN
  IMPORTING
   thax2[ th {{ T:=nzreal,i:=1,o(a,b:nzreal):=a*b}} ]
END thaxi2
```

Noting. But thaxi2 does not generate a TCC for the commutativity axiom.

More Theory Declarations

- `t1 : THEORY = th {{ T := nzreal }}`
  `t1` is a copy of `th` where `T` is substituted by `nzreal`. All the rest is left uninterpreted. Axioms related to `T` are generated as `t1`'s TCCs.
- `t2 : THEORY = th {{ T = nzreal }}`
  `t2` is a copy of `th` where `T` is defined as `nzreal`. All the rest is left uninterpreted. Axioms related to `T` are generated as `t2`'s TCCs.
- `t3 : THEORY = th {{ T ::= myT }}`
  `t1` is a copy of `th` where `T` is renamed `myT`, which is uninterpreted. No TCCs are generated for `t3`.
- All three kinds of partial interpretations can be given at once.

```
t4 : THEORY =
  th {{ T := real, i ::= e, o(a,b) = a+b }}
```

Same-name Interpretations

The notation

`IMPORTING th :-> thi`

is syntactic sugar for

`IMPORTING th{{ x_1 := thi.x_1, ..., x_n := thi.x_n }}`

where `x_1`,...,`x_n` are identifiers with the same name in `th` and `thi`.

Real Number Proving in PVS

César A. Muñoz

NASA Langley Research Center
Cesar.A.Munoz@nasa.gov

PVS Class 2010

Why Real Number Proving in PVS ?

- There are more numbers than integers in *real* life (despite what model-checkers are telling you).
- Conceptually, it is easier to reason on a continuous framework than on a discrete one.
- A lot of classical results in calculus, trigonometry, and continuous mathematics.
- Sometimes you cannot avoid them: hybrid systems, engineering applications, etc.

I Use a CAS, Why Should I Bother with PVS?

Computer Algebra Systems (CAS):

- Mathematica, Maple, Matlab, Scilab,... offer very powerful symbolic and numerical engines.
- CAS do not aim *soundness*. Singularities and exceptions are well-known problems of CAS.
- CAS do not support specification languages but programming languages.

CAS vs. Theorem Provers

- Real analysis is not a traditional strength of theorem provers.
- Theorem provers and CAS can be integrated in useful ways:
  - Computer algebra systems can be used to perform mechanical simplifications and find potential solutions.
  - Theorem prover are then used to verify the correctness of a particular solution.

I. Real Numbers in PVS

- Reals are defined as an uninterpreted subtype of `number` in the prelude library:

  ```
  real: TYPE+ FROM number
  ```

- All numeric constants are `real`:
  - naturals: `0,1,...`
  - integers: `...,-1,0,1,...`
  - rationals: `...,-1/10,...,3/2,...`

- Decimal notation is supported in PVS 4: The decimal number `3.141516` is syntactic sugar for the rational number `31416/10000`.

PVS's `real` numbers are $\mathbb{R}$

(Rather than floating point numbers)

- All the standard properties: infinite, non-enumerable, $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$, ...
- Exact arithmetic: `1/3 + 1/3 + 1/3 = 1`.
- The type `real` is unbounded:

```
googol     : real = 10^100
googolplex : real = 10^googol

googol_prop : LEMMA
  googolplex > googol * googol
```

PVS's `real` is Built-in

- Numerical expressions can be automatically reduced by the theorem prover (no need to prove 1+1=2), ...
- ... except for *machine physical limitations*: Try to prove `googol_prop` with `grind`.

You can still prove `googol_prop` using analytical methods.

Subtypes of `real`

```
nzreal  : TYPE+ = {r:real| r /= 0} % Nonzero reals
nnreal  : TYPE+ = {r:real| r >= 0} % Nonnegative reals
npreal  : TYPE+ = {r:real| r <= 0} % Nonpositive reals
negreal : TYPE+ = {r:real| r < 0} % Negative reals
posreal : TYPE+ = {r:real| r > 0} % Positive reals

rat     : TYPE+ FROM real
int     : TYPE+ FROM rat
nat     : TYPE+ FROM int
```

The uninterpreted type `number` is the only `real`'s supertype predefined in PVS: no complex numbers, no hyper-reals, no $\mathbb{R}^{\infty}$, ...

Predefined Operations

```
+, -, *: [real, real -> real]
/: [real, nzreal -> real]
-: [real -> real]

sgn(x:real)  : int = IF x >= 0 THEN 1 ELSE -1 ENDIF
abs(x:real)  : {nny: nnreal | nny >= x} = ...
max(x,y:real): {z: real | z >= x AND z >= y} = ...
min(x,y:real): {z: real | z <= x AND z <= y} = ...
^(x: real,i:{i:int | x /= 0 OR i >= 0}): real = ...
```

. . . and what about $\sqrt{}, \int, \log, \exp, \sin, \cos, \tan, \pi, \lim, \ldots$ *?*

NASA PVS Libraries

Among many others, the following libraries are available at
http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html

- `reals`: Square, square root, quadratic formula, polynomials.
- `analysis`: Real analysis, limits, continuity, derivatives, integrals.
- `series`: Power series, Taylor's theorem.
- `lnexp` and `lnexp_fnd`: Logarithm, exponential, and hyperbolic functions.
- `trig` and `trig_fnd`: Trigonometry.
- `complex`: Complex numbers.
- `float`: Floating point numbers.

To Be Or Not To Be (Fundational) ?

- Axiomatic theories `trig` and `lnexp` typechek faster.
- Fundational theories `trig_fnd` and `lnexp_fnd` have no axioms, and are updated regularly.
- Careful what you wish for:

```
  |-----
{1}   sin(pi / 2) > 1 / 2

Rule? (grind)
Integral rewrites Integral[real](0, 1, atan_deriv_fn)
  to integral(0, 1, atan_deriv_fn)
atan_value rewrites atan_value(1)
  to integral(0, 1, atan_deriv_fn)
atan rewrites atan(1)
  to integral(0, 1, atan_deriv_fn)
pi rewrites pi
  to 4 * integral(0, 1, atan_deriv_fn)
sin_value rewrites sin_value
  to ...
```

II. Low Level Real Number Proving

Real numbers in PVS are axiomatically defined in the PVS prelude:

- Theory `real_axioms`:
  Commutativity, associativity, identity, etc. These properties are known to the decision procedures, so they rarely need to be cited.

- Theory `real_props`:
  Order and cancellation laws. These lemmas are not used automatically by the standard decision procedures.

If You Really Want to Know . . .

```
real_props: THEORY
 BEGIN
  both_sides_plus_le1: LEMMA x + z <= y + z IFF x <= y
  both_sides_plus_le2: LEMMA z + x <= z + y IFF x <= y
  both_sides_minus_le1: LEMMA x - z <= y - z IFF x <= y
  both_sides_minus_le2: LEMMA z - x <= z - y IFF y <= x
  both_sides_div_pos_le1: LEMMA x/pz <= y/pz IFF x <= y
  both_sides_div_neg_le1: LEMMA x/nz <= y/nz IFF y <= x
  ...
  abs_mult: LEMMA abs(x * y) = abs(x) * abs(y)
  abs_div: LEMMA abs(x / n0y) = abs(x) / abs(n0y)
  abs_abs: LEMMA abs(abs(x)) = abs(x)
  abs_square: LEMMA abs(x * x) = x * x
  abs_limits: LEMMA -(abs(x) + abs(y)) <= x + y AND
                    x + y <= abs(x) + abs(y)
END real_props
```

Tip 1: Avoid real_props

```
  |-----
{1}   nnx / (nnx + 1) <= 1

Rule? (grind)

  |-----
{1}   nnx / (1 + nnx) <= 1

Rule? (grind :theories "real_props")
div_mult_pos_le1 rewrites nnx / (1 + nnx) <= 1
  to TRUE
Q.E.D.
```

A Toy Example

```
toy :

  |-----
{1}   x * (1 - x) <= 1

Rule?  (grind :theories "real_props")

toy :

  |-----
{1}   x - x * x <= 1
```

Tip 2: Use `both-sides` To Operate Both Sides of a Formula

(But **only** to **add/subtract** both sides of a formula)

```
toy :

  |-----
{1}   x - x * x <= 1

Rule? (both-sides "-" "1/4")
Applying - 1 / 4 to both sides of an inequality/equality
conjunction, this simplifies to:
toy :

  |-----
{1}   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule?
```

Tip 3: Use case to Prove What You Want . . .

(No what PVS offers you)

```
Rule? (case "x - x * x - 1 / 4 <= 0")
this yields  2 subgoals:
toy.1 :
{-1}  x - x * x - 1 / 4 <= 0
  |-----
[1]   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule? (assert)
This completes the proof of toy.1.

toy.2 :
  |-----
{1}   x - x * x - 1 / 4 <= 0
[2]   x - x * x - 1 / 4 <= 1 - 1 / 4

Rule? (hide 2)
```

. . . And Arrange Expressions With case-replace

```
toy.2 :
  |-----
[1]   x - x * x - 1 / 4 <= 0
Rule? (case-replace
       "x - x * x - 1 / 4 = -(x-1/2)*(x-1/2)"
       :hide? t)
this yields  2 subgoals:
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * (x - 1 / 2) <= 0

Rule? (grind :theories "real_props")
this simplifies to:
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * x - (1 * -(x - 1 / 2)) / 2 <= 0
```

Tip 4: Introduce New Names to Avoid Distribution Laws

```
toy.2.1 :
  |-----
{1}   -(x - 1 / 2) * (x - 1 / 2) <= 0

Rule? (name-replace "X" "(x-1/2)" :hide? nil)
this simplifies to:
toy.2.1 :
{-1}  (x - 1 / 2) = X
  |-----
{1}   -X * X <= 0
```

Finally . . .

```
toy.2.1 :
{-1}  (x - 1 / 2) = X
  |-----
{1}   -X * X <= 0

Rule? (grind :theories "real_props")
This completes the proof of toy.2.1.

toy.2.2 :

  |-----
{1}   x - x * x - 1 / 4 = -(x - 1 / 2) * (x - 1 / 2)
[2]   x - x * x - 1 / 4 <= 0

Rule? (assert)
Q.E.D.
```

Tip 5: Don't Reinvent the Wheel

(Look into the NASA libraries first!)

Theory `reals@quadratic`:

```
quadratic_le_0 : LEMMA
  a*sq(x) + b*x + c <= 0 IFF
  ((discr(a,b,c) >= 0 AND
    ((a > 0 AND x2(a,b,c) <= x AND x <= x1(a,b,c)) OR
     (a < 0 AND (x <= x1(a,b,c) OR x2(a,b,c) <= x)))) OR
   (discr(a,b,c) < 0 AND c <= 0))
```

A Simpler Proof

```
  |-----
{1}   x * (1 - x) <= 1

Rule? (lemma "quadratic_le_0"
                ("a" "-1" "b" "1" "c" "-1" "x" "x"))
      (grind)
Trying repeated skolemization, instantiation, and
if-lifting,
Q.E.D.
```

III. Strategies for Algebraic Manipulations

- Manip: Package for algebraic manipulations of real-valued expressions.
- Developed by B. Di Vito (NASA LaRC).
- Included as part of the PVS NASA Libraries.
- The package consists of:
  - Strategies.
  - Extended notations for formulas and expressions.
  - Emacs extensions.
  - Support functions for strategy developers.

Manip Strategies: Basic Manipulations

Strategy	**Description**
`(swap-rel fnums)`	Swap sides and reverse relations
`(swap! expr-loc)`	$x \circ y \Rightarrow y \circ x$
`(group! expr-loc LR)`	$(x \circ y) \circ z \Rightarrow x \circ (y \circ z)$
`(flip-ineq fnums)`	Negate and move inequalities
`(split-ineq fnum)`	Split $\leq$ ($\geq$) into $<$ ($>$) and $=$

Extended Formula Notation

- Standard
  - `*`: All formulas.
  - `-`: All formulas in the antecedent.
  - `+`: All formulas in the consequent.
- Extended (Manip strategies only)
  - `(^ n1...nk)`: All formulas but `n1,...,nk`
  - `(-^ n1...nk)`: All antecedent formulas but `n1,...,nk`
  - `(+^ n1...nk)`: All consequent formulas but `n1,...,nk`

(Basic) Extended Expression Notation

- Term indexes:
  - `L,R`: Left- or right-hand side of a formula.
  - `n`: `n`-th term from left to right in a formula.
  - `-n`: `n`-th term from right to left in a formula.
  - `*`: All terms in a formula.
  - `(^ n1...nk)`: All terms in a formula but `n1,...,nk`.
- Location references:
  - `(! fnum LR i1...in)`: Term in formula `fnum`, Left- or Right-hand side, at recursive path location `i1...ik`.

Examples

```
{-1}  x * r + y * r + 1 >= r - 1
  |-----
{1}   r = y * 2 * x + 1

Rule? (swap-rel -1)

{-1}  r - 1 <= x * r + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (swap! (! -1 R 1))

{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1
```

```
{-1}  r - 1 <= r * x + y * r + 1
  |-----
[1]   r = y * 2 * x + 1

Rule? (group! (! 1 R 1) R)

[-1]  r - 1 <= r * x + y * r + 1
  |-----
{1}   r = y * (2 * x) + 1

Rule? (flip-ineq -1)

  |-----
{1}   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1
```

```
  |-----
{1}   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1

Rule? (split-ineq 1)
Splitting off the equality case from formula 1,
this yields  2 subgoals:
{-1}  r - 1 = r * x + y * r + 1
  |-----
[1]   r - 1 > r * x + y * r + 1
[2]   r = y * (2 * x) + 1

Rule? (postpone)
  |-----
{1}   r - 1 = r * x + y * r + 1
[2]   r - 1 > r * x + y * r + 1
[3]   r = y * (2 * x) + 1
```

More Strategies

Strategy	**Description**
`(mult-by fnums term)`	Multiply formula by term
`(div-by fnums term)`	Divide formula by term
`(move-terms fnum L|R tnums)`	Move additive terms left and right
`(isolate fnum L|R tnum)`	Isolate additive terms
`(cross-mult fnums)`	Perform cross-multiplications
`(factor fnums)`	Factorize formulas
`(factor!  expr-loc)`	Factorize terms
`(mult-eq fnum fnum)`	Multiply equalities
`(mult-ineq fnum fnum)`	Multiply inequalities

More Examples

```
{-1}  (x * r + y) / pa > (r - 1) / pb
  |-----
{1}   r - y * 2 * x = 1

Rule? (cross-mult -1)

{-1}  pb * r * x + pb * y > pa * r - pa
  |-----
[1]   r - y * 2 * x = 1

Rule? (isolate 1 L 1)

[-1]  pb * r * x + pb * y > pa * r - pa
  |-----
{1}   (r = 1 + y * 2 * x)



{-1}  x * y - pa + na < x * na * pa
{-2}  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * x + 2 * y

Rule? (move-terms -1 L (2 3))

{-1}  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * x + 2 * y

Rule? (factor 1)

[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)
```

```
[-1]  (x * y < x * na * pa + pa - na)
[-2]  r - y * 2 * x = 1
  |-----
{1}   2 * pa = 2 * (x + y)

Rule? (mult-eq -1 -2)

{-1}  (x * y)*(r - y * 2 * x) < (x * na * pa + pa - na)*1
[-2]  (x * y < x * na * pa + pa - na)
[-3]  r - y * 2 * x = 1
  |-----
[1]   2 * pa = 2 * (x + y)

Rule? (div-by 1 "2")

...
  |-----
{1}   (pa = (x + y))
```

The Field Package

- Field: A simplification procedure for the field of real numbers.
- Included as part of the PVS NASA Libraries.
- The package consists of:
  - The strategy `field`.
  - Several *extra-tegies*.

field

```
{-1}  vox > 0
{-2}  s * s - D*D > D
{-3}  s * vix * voy - s * viy * vox /= 0
{-4}  ((s * s - D*D) * voy - D * vox * sqrt(s*s - D*D))/
       (s * (vix * voy - vox * viy)) * s * vox /= 0
{-5}   voy * sqrt(s * s - D*D) - D * vox /= 0
  |-----
{1}  (viy * sqrt(s * s - D*D) - vix * D) /
       (voy * sqrt(s * s - D*D) - vox * D) =
       (D*D - s * s) / (((s * s - D*D) * voy - D * vox *
       sqrt(s * s - D*D)) /
       (s * (vix * voy - vox * viy)) * s * vox) +
       vix / vox

Rule? (field 1)
Q.E.D.
```

Some Extra-tegies

Strategy	Description
(grind-reals)	grind + real_props
(cancel-by fnum term)	Cancel a common term in a formula
(skoletin fnum)	Skolemize let-in expressions
(skeep fnum)	Skolemize with same variable names
(neg-formula fnum)	Negate a formula
(add-formula fnum fnum)	Add two formulas

Forget Tip 1 and Tip 4, Use grind-reals

```
  |-----
{1}   (x - 1 / 2) * (x - 1 / 2) >= 0

Rule? (grind-reals :nodistrib 1)

Q.E.D.
```

cancel-by

```
{-1}  4 * (pa * pb) + (pa * 6) * pa = pa * ((c + 1) * 2)
  |-----
{1}   2 * pb + 3 * pa = c

Rule? (cancel-by -1 "2*pa")

{-1}  (3 * pa) + (2 * pb) = 1 + c
  |-----
{1}   2 * pa = 0
{2}   3 * pa + 2 * pb = c
```

PVS's Let-in Expressions

- Let-in expressions are used in PVS to introduce local definitions.

- They are automatically unfolded by the theorem prover.

```
  |-----
{1}   LET a = a * y + 2 IN
        LET b = a + x IN
          LET c = a + b IN -b + 4 * a * c / 2 = 0

Rule? (assert)

  |-----
{1}   (32 + 8 * (x*x * y*y) + 4 * (x*x*y) + 16 * (x*y) +
        16 * (x*y) + 8*x) / 2 + -(2 + x*y + x) = 0
```

Let-in Expressions Go Wild

```
  |-----
{1}   LET a = (x + 1) IN LET b = a * a IN
      LET c = b * b IN c * c >= a

Rule? (assert)

  |-----
{1} 1 + x + (x*x*x*x*x*x*x*x + x*x*x*x*x*x*x*x)
        + (x*x*x*x*x*x*x*x + x*x*x*x*x*x*x*x)
        + (x*x*x*x*x*x*x*x + x*x*x*x*x*x*x*x)
      ...
        + (x*x + x)
        + (x*x + x)
        + (x*x + x)
        >= 1 + x
```

Tip 8. To unfold a let-in, use skoletin

```
  |-----
{1}   LET a = (x + 1) IN LET b = a * a IN
      LET c = b * b IN c * c >= a

Rule? (skoletin 1)
{-1}  a = (x + 1)
  |-----
{1}   LET b = a * a IN LET c = b * b IN c * c >= a

Rule? (skoletin* 1)
{-1}  c = b * b
{-2}  b = a * a
[-3]  a = (x + 1)
  |-----
{1}   c * c >= a
```

More examples

```
  |-----
{1}   FORALL (nnx: nnreal, x: real):
        nnx > x - nnx*nnx AND x + 2 * nnx*nnx >= 4 * nnx
        IMPLIES nnx > 1

Rule? (skeep)
{-1}  nnx > x - nnx*nnx
{-2}  x + 2 *nnx*nnx >= 4 * nnx
  |-----
{1}   nnx > 1

Rule? (neg-formula -1)
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1
```

```
{-1}  nnx*nnx - x > -nnx
[-2]  x + 2 * nnx*nnx >= 4 * nnx
  |-----
[1]   nnx > 1

Rule? (add-formulas -1 -2)
{-1}  3 * (nnx*nnx) > -nnx + 4 * nnx
  |-----
[1]   nnx > 1

Rule? (cancel-by -1 "nnx")

Q.E.D.
```

IV. Strategies for Specialized Domains

- Linear arithmetic via Yices.
- Numerical calculations via Interval.

Yices

- Yices is a SMT (Satisfiability Modulo Theories) solver developed at SRI.
- Background theories supported by Yices:
  - Linear arithmetic (real and integer): addition and multiplication by scalar.
  - Arrays.
  - Uninterpreted functions.
  - Datatypes.
  - Bit vectors.
  - Quantifiers.

yices

```
  |-----
{1}   EXISTS (x, y: real): x /= y

Rule? (yices)
(assert
  (not (exists ( x_1::real y_2::real) (/=  x_1 y_2))))

Result = unsat
Logical context is inconsistent. Use (reset) to reset.
unsat
Yices translation of negation is unsatisfiable
Simplifying with Yices,
Q.E.D.
```

yices

```
  |-----
{1}   EXISTS (x, y: real):
      1 <= 3 * x - 3 * y AND 3 * x - 3 * y <= 2

Rule? (yices)
Yices translation of negation is unsatisfiable
Simplifying with Yices,

Q.E.D.
```

Behind this QED there is **no** *"real" PVS proof!*

The Interval Package

- A package for interval analysis.
- Exact real calculations including trigonometric and transcendental functions.
- http://shemesh.larc.nasa.gov/people/cam/Interval

numerical

```
  |-----
{1}   sin(6 * pi / 180) + sqrt(2) <= 2.11

Rule? (numerical)

Evaluating formula using numerical approximations,
Q.E.D.
```

Behind this QED there is a "real" PVS proof!

instint

```
{-1}  x ## [| 0, 2 |]
  |-----
{1}   sqrt(x) + sqrt(3) < 315 / 100

Rule? (instint)
Proving that an expression is in a given interval,
Q.E.D.
```

instint with Splitting

```
{-1}  x ## [| 0, 1 |]
  |-----
{1}   x * (1 - x) ## [| 0, 1 / 3 |]

Rule? (instint)
{-1}  x * (1 - x) ## Mult([|0, 1|], Sub([|1|], [|0, 1|]))
{-2}  x ## [|0, 1|]
  |-----
{1}   Mult([|0, 1|], Sub([|1|], [|0, 1|])) < [|0, 1 / 3|]

Rule? (undo)
Rule? (instint :splitting 6)
Q.E.D.
```

Essential Tools for the Real Practitioner

- PVS NASA libraries.
- Manip/Field packages.
- Depending on your application: Yices and Interval.

ProofLite: Batch Proof Checking and Literate Proving in PVS

César A. Muñoz

NASA Langley Research Center
Cesar.A.Munoz@nasa.gov

PVS Class 2010

The PVS Theorem Prover

- PVS is a powerful interactive theorem prover.
- PVS provides a powerful batch mode too (but mainly for expert users).
- Why do *normal users* need a batch mode ?

Scenario 1

After several weeks we have finished the development of an Interval library in PVS: 10 files, 322 lemmas.

- We want to double check the status of all lemmas.
- A new version of PVS is available. We want to recheck all the proofs.

Scenario 2

- We want to write proof scripts in the same file where we have the PVS specification.
- We are working on a third party application that generates PVS specifications *and PVS proofs*.

The ProofLite Package

- Package for non-interactive proof checking and proof scripting in PVS:
  - Utility for running the theorem prover in batch mode: `proveit`.
  - A proof scripting notation where proof scripts reside in `.pvs` files.
- Suitable for batch generation of specifications and proof scripts.
- Download: http://shemesh.larc.nasa.gov/people/cam/ProofLite

The `proveit` Utility

```
% proveit --importchain --clean --packages Field -a Interval/top.pvs
Processing Interval/top.pvs. Writing output to file top.out.

 Proof summary for theory interval
    IMP_sigma_TCC1.....................proved - complete
    sharp_Proper.......................proved - complete
    Proper_sharp.......................proved - complete
    specialbrackets_TCC1...............proved - complete
    Lt_Ge..............................proved - complete
    Le_Gt..............................proved - complete
    Abs_TCC1...........................proved - complete
    Abs_TCC2...........................proved - complete
    ...
    Theory totals: 156 formulas, 156 attempted, 156 succeeded
 ...
 Grand Totals: 322 proofs, 322 attempted, 322 succeeded (122.73 s)
```

ProofLite Scripts

- ProofLite scripts are written in PVS files using the special comment form:

```
l1: LEMMA a*a >= 0
%|- l1 : PROOF (grind) QED
```

- ProofLite scripts can extend to multiple lines:

```
l2: LEMMA (nza/2)*(2/nza) = 1
%|- l2 : PROOF
%|-   (then (skosimp)
%|-         (grind))
%|- QED
```


Sharing ProofLite Scripts

Several lemmas can share the same ProofLite script:

```
l3: LEMMA a*a >= 0

l4: LEMMA (nza/2)*(2/nza) = 1

%|- l3 : PROOF
%|- l4 : PROOF
%|-   (grind)
%|- QED
```

ProofLite Scripts for Name-Matching Formulas

- Name-matching formulas can share the same ProofLite script.
- The symbol * stands for an arbitrary sequence of one or more characters, e.g.,

```
%|- *TCC* : PROOF
%|-   (grind)
%|- QED
```


Macro Scripts

- Name-matching lemmas can be used to create macro scripts.
- The symbol $0 refers to the name of the lemma and the symbol $*n* refers to *n*-th matching string from left to right, e.g.,

```
l_5_6 : LEMMA EXISTS (a) :  5 < a AND a < 6

l_6_7 : LEMMA EXISTS (a) :  6 < a AND a < 7

%|- l_*_* : PROOF
%|-   (then (skip-msg "Proving Lemma: $0")
%|-         (inst 1 "$1 + ($2 - $1)/2")
%|-         (grind))
%|- QED
```

Parametric Scripts

- Parametric scripts have the form:

```
%|- <script_name>[e1;...;en]: PROOF
%|-   <steps>
%|- QED
```

- The symbol `#n` is substituted by e_n, e.g.,

```
l_8 : LEMMA EXISTS (a,b) :  a+b = 8
l_9 : LEMMA EXISTS (a,b) :  a+b = 9
%|- l_8[2;6] : PROOF
%|- l_9[4;5] : PROOF
%|-   (then (skip-msg "Proving Lemma: $0")
%|-         (inst 1 "#1" "#2")
%|-         (grind))
%|- QED
```


Installing ProofLite Scripts

Interactively

- ProofLite scripts in the current theory.
  - Without overriding old proofs:
    `M-x install-prooflite-scripts-theory` (`C-c it`).
  - Overriding old proofs:
    `M-x install-prooflite-scripts-theory!` (`C-c !t`).

- ProofLite scripts at the cursor position.
  - Without overriding old proofs:
    `M-x install-prooflite-script` (`C-c ip`).
  - Overriding old proofs:
    `M-x install-prooflite-script!` (`C-c !p`).

Installing ProofLite Scripts

In batch mode

`proveit` automatically installs ProofLite scripts on untried formulas (and on tried formulas if the option `--force` is used).

Creating ProofLite Scripts from Proofs

ProofLite scripts can be created from proofs in two ways:

- Place the cursor on the formula for which you want to create a ProofLite script and issue the Emacs command: `M-x insert-prooflite-script` (`C-c 2p`). The ProofLite script is automatically inserted after the formula.

- Issue the command: `M-x display-prooflite-script` (`C-c dp`) and enter the name of a formula. The ProofLite script of that formula is displayed in the buffer "ProofLite".

Conclusion

- The basic capabilities provided by ProofLite are already available in proof assistants such as Coq, HOL, etc.
- The ProofLite scripting notation also supports several forms of proof sharing and proof reuse.
- Proof scripts provide a simple mechanism to write user defined strategies.
- *Batch Proving and Proof Scripting in PVS*, César Muñoz, NASA Contract Report, `http://hdl.handle.net/2060/20070012333`.

PVSio: A Rapid Prototyping Tool for PVS

César A. Muñoz

NASA Langley Research Center
Cesar.A.Munoz@nasa.gov

PVS Class 2010

A Trivia Question

- In addition to prove mathematical theories, what else can you do in PVS ?
- To animate them!

Animation

- What: Animation is the process of executing a specification to validate its intended semantics.
- Why: It is cheaper, faster, more fun to test a specification than to prove it.
- How: The PVS ground evaluator.

PVS as a Declarative Programming Language

Most specifications in PVS are functional:

```
sqrt_newton(a:nnreal,n:nat): recursive posreal =
  if n=0 then a+1
  else let r=sqrt_newton(a,n-1) in
    (1/2)*(r+a/r)
  endif
  measure n+1
```

PVS as a Calculator

```
|-----
{1}   sqrt_newton(2, 3) <= 3 / 2

Rule? (grind)

...
sqrt_newton rewrites sqrt_newton(2, 3)
  to (1/2)*(2/(3*((1/2)*(1/2)) + (1/2)*(2/(3*(1/2)
  + (1/2)*(2/3))) + (1/2)*(1/2)*(2/3)))
  + 3*((1/2)*(1/2)*(1/2)) + (1/2)*(1/2)*(2/(3*(1/2)
  + (1/2)*(2/3))) + (1/2)*(1/2)*(1/2)*(2/3)
Trying repeated skolemization, instantiation, and
if-lifting,
Q.E.D.
```

PVS is an Inefficient Calculator!

```
  |-----
{1}   2 < sqrt_newton(2, 10) * sqrt_newton(2, 10)

Rule? (grind)

...
sqrt_newton rewrites sqrt_newton(2, 4)
  to (1/2) * (2 / ((1/2) * (2 / (3 * ((1/2) * (1/2))
  + (1/2) * (2/(3 * (1/2) + (1/2) * (2/3)))
  + (1/2) * (1/2) * (2/3)))
  + 3 * ((1/2) * (1/2) * (1/2))))
  + ...
```

The PVS Ground Evaluator

- An experimental feature of PVS 3.x.
- An efficient Lisp code generator for PVS functional specifications.
- A read-eval-loop interface available with the Emacs command `M-x pvs-ground-evaluator`.
- Remark: The ground evaluator is not integrated into the theorem prover.

An Efficient Calculator . . .

```
<GndEval> "sqrt_newton(2,3) <= 3/2"
==>
TRUE

<GndEval> "2 < sqrt_newton(2,10) * sqrt_newton(2,10)"

==>
TRUE
```

. . . with a Poor Interface

```
<GndEval> "sqrt_newton(2,10)"

==>
106854041125800542495773099620277025175306170088676005050 92
624567599571273090520553619648095761832386318805390738103 27
270637139651714658235752986741761590590866587906685398566 65
003418661673043593439606033431706584881164409983476684441 99
983686538759126038708139700443953973427284872836266393035 83
382899371027555772333046373835945759772882491255347900260 95
542941439658998944011413943276015695324890773234847928444 85
539928427382590741387820229684450437162379033859897869490 83
471731623224703430657/755572170772397944964839262527264855 5
571173428458843215513318834629907352217519115613298261778 04
687915531346855493348073078968123567523914573023555322568 50
771206171730831540313524856910300210308020618231529918595 23
939785461317894299181774172751814809171189000327511471361 08
```

I. PVSio

- An alternative interface to the PVS Ground Evaluator.
- Implemented as a PVS package (prelude library extension).
- Safely integrated into the theorem prover.
- Already part of the PVS distribution.

- Note: The current PVS release 4.1 misses two files. Check http://research.nianet.org/~munoz/PVSio.

More than a Pretty Face

- A predefined set of PVS functions for input/output operations, side-effects, unbounded-loops, exceptions, string manipulations, and floating point arithmetic
- A high level interface for extending PVS programming language features.
- A tool for rapid prototyping.

M-x pvsio

```
+---
| PVSio-2.a (11/23/04)
| ...
+---

<PVSio> println(sqrt_newton(2,10));

1.4142135
==>
TRUE
```

Input Operations

```
<PVSio> let x = read_real in
        println("sqrt("+x+")="+sqrt_newton(x,10));

subtype TCC for x: FORALL (x: rat): x = query_real(empty)
IMPLIES x >= 0
Evaluating in the presence of unproven TCCs may be unsound

10

sqrt(10)=3.1622777
==>
TRUE
```

Floating Points and a Random Surprise

```
<PVSio> SQRT(2);

==>
1.4142135

<PVSio> RANDOM = RANDOM;

==>
FALSE

<PVSio> let r=RANDOM in r = r;

==>
TRUE
```

Furthermore

- String manipulations.
- Streams and files.
- Unbounded loops.
- Exceptions.
- Local and global variables.
- Basic parsing and lexing.
- PVS parsing and typechecking.

II. PVSio Semantic Attachments

- A high-level interface to the the PVS Common Lisp machine.
- A user-friendly mechanism for extending the ground evaluator.
- Lisp functions attached to uninterpreted PVS functions.

User-defined Attachments

- How:

```
(defattach theory.name
   doc-string
   body)
```

- Where: `pvs-attachments` or `<user>/.pvs-attachments`.
- Example:

```
;; File: pvs-attachments
(defattach my_cosh.cosh (x)
   "Hyperbolic cosine of X"
   (cosh x))
```

III. PVSio Animations

Recall the theory interpretation example:

```
maxl_ax : THEORY
BEGIN
 IMPORTING list[real]

 maxl : [list->real]

 l : VAR list
 x : VAR real

 Maxl : AXIOM
   member(x,l) implies
   x <= maxl(l)
END maxl_ax
```

```
maxl_th : THEORY
BEGIN
 IMPORTING list[real]

 maxl(l:list) : RECURSIVE real =
   cases l of
     null : 0,
     cons(a,r) : max(a,maxl(r))
   endcases
   MEASURE l by <<

END maxl_th
```

First Step: Build an Interpretation

Second Step: PVSio Bells and Whistles

```
test : THEORY
BEGIN

  IMPORTING maxl_th,
            maxl_ax{{ maxl :=  maxl }}
END test
  main : void =
    println("Testing the function maxl") &
    let s = query_line("Enter a list of real numbers:") in
    let l = str2pvs[list[real]](s) in
    let m = maxl(l) in
      println("The max of "+s+" is "+m)
END test
```

Final Step: Test It

```
<PVSio> main;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, 5, 3, 2 :)
The max of (: -1, -2, 5, 3, 2 :) is 5
==>
TRUE

<PVSio> main;
Testing the function maxl
Enter a list of real numbers:
(: -1, -2, -3, -4 :)
The max of (: -1, -2, -3, -4 :) is 0
==>
TRUE
```

For the Emacs-Allergic: PVSio Applications

```
$ pvsio test:main
Testing the function maxl
Enter a list of real numbers:
(: 5, 4 ,3 ,2 :)
The max of (: 5, 4 ,3 ,2 :) is 5
==>
TRUE
```

III. PVSio in the Theorem Prover

- PVSio safely enables the ground evaluator in the theorem prover.
- Ground expressions are translated into Lisp and evaluated in the PVS Lisp engine.
- The theorem prover only trusts the Lisp code automatically generated from PVS functional specifications.
- Semantic attachments are always considered harmful for the theorem prover.

The Strategy eval-formula

Evaluation of ground expressions via the ground evaluator:

```
  |-----
{1}   2 < sqrt_newton(2, 10) * sqrt_newton(2, 10)

Rule? (eval-formula 1)

Q.E.D.
```

Fast and Sound

Well, as Sound as the Lisp Engine

```
|-------
{1}   RANDOM /= RANDOM
Rule? (eval-formula 1)

Function stdmath.RANDOM is defined as a semantic attachment
It cannot be evaluated in a formal proof.

No change on: (eval-formula 1)
```

Applications

- CD3D, KB3D: Conflict detection and resolution algorithms for distributed air traffic management.
- SATS: A discrete model of NASA's Small Aircraft Transpiration System for High Volume Operations.
- BESC: A basic explicit safety checker for transition systems.

References

- Documentation: `http://research.nianet.org/~munoz/PVSio`.
- Rapid prototyping in PVS, César Muñoz, NASA Contract Report, `http://hdl.handle.net/2060/20040046914`.
- Efficiently Executing PVS, N. Shankar, SRI Technical Report.
- Evaluating, Testing, and Animating PVS Specifications Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave Stringer-Calvert, SRI Technical Report, `http://www.csl.sri.com/users/rushby/abstracts/attachments`.

Strategy Writing in PVS

César A. Muñoz

NASA Langley Research Center
Cesar.A.Munoz@nasa.gov

PVS Class 2010

PVS Strategies

- A conservative mechanism to extend theorem prover capabilities by defining new proof commands, i.e.,
- User defined strategies do not compromise the soundness of the theorem prover.

I. PVS Strategy Language

- Atomic (blackbox) proof rules are called *rules* in PVS.
- Non-atomic (glassbox) proof rules are called *strategies* in PVS.

Henceforth, we use strategy to refer both glassbox strategies and atomic rules.

Basic Steps

- Any proof command, e.g., `(ground)`, `(case ...)`, etc.
- `(skip)` does nothing.
- `(skip-msg message)` prints `message`.
- `(fail)` fails the current goal and reaches the next backtracking point.
- `(label label fnums)` labels formulas `fnums` with string `label`.
- `(unlabel fnums)` unlabels formulas `fnums`.

Combinators

- Sequencing: `(then step1 ...stepn)`.
- Branching: `(branch step (step1 ...stepn))`.
- Binding local variables:
  `(let ((var1 lisp1) ...(varn lispn)) step)`.
- Conditional: `(if lisp step1 step2)`.
- Loop: `(repeat step)`.
- Backtracking: `(try step step1 step2)`.

Sequencing

- `(then step1 ...stepn)`:
  Sequentially applies `stepi` to *all the subgoals* generated by the previous step.
- `(then@ step1 ...stepn)`:
  Sequentially applies `stepi` to *the first subgoal* generated by the previous step.

Branching

- `(branch step (step1 ...stepn))`:
  Applies `step` and then applies `stepi` to the *i*'th subgoal generated by `step` . If there are more subgoals than steps, it applies `stepn` to the subgoals following the *n*'th one.
- `(spread step (step1 ...stepn))`:
  Like branch, but applies `skip` to the subgoals following the *n*'th one.

Binding Local Variables

- `(let ((var1 lisp1) ...(varn lispn)) step)`:
  Allows local variables to be bound to Lisp forms (`vari` is bound to `lispi`).
- Lisp code may access the proof context using the PVS Application Programming Interface (API).

Conditional and Loops

- `(if lisp step1 step2)`:
  If `lisp` evaluates to `NIL` then applies `step2`. Otherwise, it applies `step1`.
- `(repeat step)`:
  Iterates `step` (while it does something) on the the first subgoal generated at each iteration.
- `(repeat* step)`:
  Like `repeat`, but carries out the repetition of `step` along *all the subgoals* generated at each iteration.*

*Note that `repeat` and `repeat*` are potential sources of infinite loops.

Backtracking

- Backtracking is achieved via `(try step step1 step2)`.
- Informal (but naive) explanation: Tries `step`, if it *does nothing*, applies `step2` to the new subgoals. Otherwise, applies `step1`.
- The behavior of `try` is far more complex:
  - What is the meaning of "does nothing"?
  - How does the backtracking feature work?

To Do or Not to Do

step does nothing usually means that no subgoals are generated (but this is not enough).

`step` does nothing when

- it behaves as `skip`.
- the proof context before and after `step` is exactly the same.
- PVS says so:

```
Rule?  step
No change on:  step
```

II. Writing your Own Strategies

- New strategies are defined in a file named `pvs-strategies` in the current context. PVS automatically loads this file when the theorem prover is invoked.
- Strategies may be defined in an arbitrary file *my_own_strategies*. In this case, the file can be loaded with the command `(load "my_own_strategies")` in the file `pvs-strategies`.
- The `IMPORTING` clause loads the file `pvs-strategies` if it is defined in the imported library.

Caveats

- PVS only loads `pvs-strategies` when this file has been updated. If we modify *my_own_strategies*, we also have to *touch* `pvs-strategies`, so that PVS automatically loads the modifications.
- Beware of name clashes: Loading a strategy definition file overwrites previous strategies with the same name.

Strategy Definition

- A strategy definition has the form:

```
(defstep name (parameters)
  step
  help-string  format-string)
```

- E.g., "Hello World" in PVS:

```
(defstep hello-world ()
   (skip-msg "Hello World")
   "Prints 'Hello World' and does nothing else"
   "Printing 'Hello World'")
```

"Hello World" in PVS

In the theorem prover:

```
Rule? (hello-world)
Printing 'Hello World'
Hello World
No change on: (hello-world)

Rule? (help hello-world)
(hello-world/$) :
    Prints 'Hello World' and does nothing else
```

Blackbox vs. Glassbox

- `defstep` generates a (blackbox) rule `name` and a (glassbox) strategy `name$`.
- `defhelper`: Same as `defstep` but for internal use only – excluded from standard user interface.
- `defstrat`: Defines a glassbox strategy `name`. Does not take the `format-string` argument.

Defining a Finite Loop

In pvs_strategies:

```
(defstrat for (n step)
  (if (<= n 0)
      (skip)
      (let ((m (- n 1)))
           (then@ step (for m step))))
  "Repeats step n times")
```

Using a Finite Loop

In the theorem prover:

```
ex1 :
  |-----
{1}   sqrt(sq(x)) + sqrt(sq(y)) + sqrt(sq(z)) <= x+y+z

Rule? (for 2 (rewrite "sqrt_sq_abs"))
...

  |-----
{1}   abs(x) + abs(y) + sqrt(sq(z)) <= x+y+z
```

References

- Documentation: PVS Prover Guide, N. Shankar, S. Owre, J. Rushby, D. Stringer-Calvert, SRI International: `http://www.csl.sri.com/pvs.html`.
- Proceedings of STRATA 2003: `http://hdl.handle.net/2060/20030067561`.
- Examples:
  - Manip: `http://shemesh.larc.nasa.gov/people/bld/manip.html`.
  - Field: `http://research.nianet.org./~munoz/Field`.
- Programming: Lisp The Language, G. L. Steele Jr., Digital Press. See, for example, `http://www.supelec.fr/docs/cltl/clm/node1.html`.

III. The *Behaves As* Relation ($\Rightarrow$)

$$\frac{\texttt{step} \Rightarrow \texttt{(fail)}}{\texttt{(try step step1 step2)} \Rightarrow \texttt{(fail)}}$$

$$\frac{\texttt{step} \Rightarrow \texttt{(skip)}}{\texttt{(try step step1 step2)} \Rightarrow \texttt{step2}}$$

$$\frac{\texttt{step1} \Rightarrow \texttt{(fail)}}{\texttt{(try step step1 step2)} \Rightarrow \texttt{(skip)}}$$

$$\frac{\text{otherwise}}{\texttt{(try step step1 step2)} \Rightarrow \texttt{step1}}$$

$$\frac{\texttt{stepi} \Rightarrow \texttt{(fail)}}{\texttt{(...stepi ...)} \Rightarrow \texttt{(fail)}}$$

Furthermore, `fail` *does not* propagate outside blackbox rules.

Example

What does `(try (grind) (fail) (skip))` do ?

- if `(grind)` $\Rightarrow$ `(skip)`, then `(skip)`
- if `(grind)` $\not\Rightarrow$ `(skip)`, then `(skip)`
- if `(grind)` finishes the proof, then `Q.E.D.`

It either completes the proof with `(grind)`, or does nothing.

IV. PVS Strategies are Written in Lisp

- Arbitrary Lisp expressions (functions, global variables, etc.) can be included in a strategy file.
- PVS's data structures are based on various Common Lisp Object System (CLOS) classes. They are available to the strategy programmer through global variables and accessory functions.

Proof Context: Global Variables

`*ps*`	Current proof state
`*goal*`	Goal sequent of current proof state
`*label*`	Label of current proof state
`*par-ps*`	Current parent proof state
`*par-label*`	Label of current parent
`*par-goal*`	Goal sequent of current parent
`*+*`	Consequent sequent formulas
`*-*`	Antecedent sequent formulas
`*new-fmla-nums*`	Numbers of new formulas in current sequent
`*current-context*`	Current typecheck context
`*module-context*`	Context of current module
`*current-theory*`	Current theory

PVS Context: Accessory Functions

- `(select-seq (s-forms *goal*) fnums)` retrieves the sequent formulas `fnums` from the current context.
- `(formula seq)` returns the expression of the sequent formula `seq`.
- `(operator expr)`, `(args1 expr)`, and `(args2 expr)` return the operator, first argument, and second argument, respectively, of expression `expr`.

PVS Context: Recognizers

Negation	`(negation? expr)`
Disjunction	`(disjunction? expr)`
Conjunction	`(conjunction? expr)`
Implication	`(implication? expr)`
Equality	`(equation? expr)`
Equivalence	`(iff? expr)`
Conditional	`(branch? expr)`
Universal	`(forall-expr? expr)`
Existential	`(exists-expr? expr)`

Formulas in the antecedent are negations.

Gold Mining in PVS

- In the theorem prover the command `LISP` evaluates a Lisp expression.
- In Lisp, `show` (or `describe`) displays the content and structure of a CLOS expression. The generic `print` is also handy.

Example

```
  |-----
{1}   sqrt(sq(x)) + sqrt(sq(y)) + sqrt(sq(z)) >= x+y+z

Rule? (LISP (show
      (formula (car (select-seq (s-forms *goal*) 1)))))

sqrt(sq(x)) + sqrt(sq(y)) + sqrt(sq(z)) >= x + y + z is
an instance of #<STANDARD-CLASS INFIX-APPLICATION>:
 The following slots have :INSTANCE allocation:
  OPERATOR            >=
  ARGUMENT            (sqrt(sq(x))+sqrt(sq(y))+sqrt(sq(z)),
                       x + y + z)
  ...
```

V. A Non-(Completely-)Trivial Example

- Assume we have a goal $e_1 = e_2$.
- Our strategy is to use an injective function f such that $f(e_1) = f(e_2)$. Then, by injectivity, $f(e_1) = f(e_2)$ implies $e_1 = e_2$.
- For instance, to prove

  ```
  {-1}  cos(x) > 0
    |-----
  {1}   sqrt(1 - sq(sin(x))) = cos(x)
  ```

  we square both sides formula {1}, i.e., $f \equiv$ sq.[†]

[†] The function sq is injective for non-negative reals.

both-sides-f

```
(defstep both-sides-f (f &optional (fnum 1))
  (let ((eqs (get-form fnum)))
       (if (equation? eqs)
           (let ((case-str (format nil "~a(~a) = ~a(~a)"
                                   f (args1 eqs)
                                   f (args2 eqs))))
                (case case-str))
           (skip)))
  "Applies function named F to both-sides of equality FNUM"
  "Applying ~a to both-sides of ~a")

(defun get-form (fnum)
  (formula (car (select-seq (s-forms *goal*) fnum))))
```

Using both-sides-f

```
Rule? (both-sides-f "sq")
Applying sq to both-sides of 1,
this yields  3 subgoals:
ex2.1 :
{-1}  sq(sqrt(1 - sq(sin(x)))) = sq(cos(x))
[-2]  cos(x) > 0
  |-----
[1]   sqrt(1 - sq(sin(x))) = cos(x)

ex2.2 :
[-1]  cos(x) > 0
  |-----
{1}   sq(sqrt(1 - sq(sin(x)))) = sq(cos(x))
[2]   sqrt(1 - sq(sin(x))) = cos(x)
```