

Strategy Writing in PVS

César A. Muñoz

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



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.

Outline

PVS Strategy Language

Writing your Own Strategies

PVS Strategies and Lisp

An Example

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

Branching

- ▶ (**branch** step (step1 ...stepn)):
Applies step and then applies step_{*i*} to the *i*'th subgoal generated by step . If there are more subgoals than steps, it applies step_{*n*} 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*

The Semantics of `try`

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

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

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

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

$$\frac{\text{step}_i \Rightarrow (\text{fail})}{(\dots \text{step}_i \dots) \Rightarrow (\text{fail})}$$

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

Example

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

- ▶ if `(grind) ⇒ (skip)`, then `(skip)`
- ▶ if `(grind) ≉ (skip)`, then `(skip)`
- ▶ if `(grind)` finishes the proof, then Q.E.D.

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

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>.
- ▶ Programming: Lisp The Language, G. L. Steele Jr., Digital Press. See, for example,
<http://www.supelec.fr/docs/cltl/clm/node1.html>.

PVS Strategies and 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

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

...

An 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

$$\{-1\} \quad \cos(x) > 0$$

|-----

$$\{1\} \quad \text{sqrt}(1 - \text{sq}(\sin(x))) = \cos(x)$$

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

[†]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)