Strategy Writing in PVS

César A. Muñoz

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

PVS Class 2012

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.

Fermat's Last Theorem (Bounded Version)

Prove the following lemma:

```
bounded_FLT3 : LEMMA
  FORALL (a,b,c:posnat):
    a <= 3 AND b <= 3 and c <= 3 IMPLIES
      a^3+b^3 /= c^3
```

- Formalize Wiles' general proof in PVS and instantiate it to $n = 3$ or
- prove each one of the 27 cases.


```
{-1}  a <= 3
{-2}  b <= 3
{-3}  c <= 3
 |-----
{1}  a ^ 3 + b ^ 3 /= c ^ 3

Rule? (case "a=1 AND b=1 AND c=1")(flatten)
{-1}  a = 1
{-2}  b = 1
{-3}  c = 1
...
  |-----
{1}  a ^ 3 + b ^ 3 /= c ^ 3

Rule? (replaces (-1 -2 -3))(eval-formula)
This completes the proof of bounded_FLT3.1.
```

Repeat this 26 times!

Strategies

Strategies enable proof scripting:

- Programatic tasks, e.g., `(case "a=1 AND b=1 AND c=1")`, `...`, `(case "a=3 AND b=3 AND c=3")`.
- Repetitive tasks, e.g., `(flatten)(replaces ...)(eval-formula ...)`.

Strategy Language: 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`.

Strategy Language: 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)`.

Strategy Language: 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.

Strategy Language: 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 explanation: Tries step, if it *does nothing*, applies step2 to the new subgoals. Otherwise, applies step1.
- What does (try (grind) (fail) (skip)) do ?

Example

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

- if `(grind)` does nothing then `(skip)`
- if `(grind)` does something (without finishing the proof) 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.
- The `IMPORTING` clause loads the file `pvs-strategies` if it is defined in the imported library.

Strategies and Rules

Strategies can be expanded into more elementary steps.

- Some strategies have a $-form for expanding their definitions, e.g., `grind$`.

- Some strategies are automatically expanded in the proof script, e.g., `repeat`.

Proof commands that cannot be expanded into elementary steps are called *rules* and cannot be defined by regular users.

Strategy Definitions

- `defstep` defines a strategy and its $-form:

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

- `defhelper` defines a strategy that is excluded from the standard user interface.

```
(defhelper name (parameters &optional parameters)
  step
  help-string  format-string)
```

- `defstrat` defines strategy that expands automatically.

```
(defstrat name (parameters &optional parameters)
  step
  help-string)
```

Example: 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

```
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
```

Example: bFLT3

```
{-1}  a <= 3
{-2}  b <= 3
{-3}  c <= 3
{-4}  a ^ 3 + b ^ 3 = c ^ 3
  |-----
Rule? (bflt3 ...)
```

In pvs-strategies:

```
(defstep bflt3 (a b c)
  ...
  "Checks a^3+b^3 /= c^3 for 0 < a,b,c <= 3"
  "Checking a^3+b^3 /= c^3 for 0 < a,b,c <= 3")
```



```
(defstep bflt3 (a b c)
  (let ((casestr (format nil "a=~a AND b=~a AND c=~a"
                         a b c)))
    (spread (case casestr)
            (...)))
  "Checks a^3+b^3 /= c^3 for 0 < a,b,c <= 3"
  "Checking a^3+b^3 /= c^3 for 0 < a,b,c <= 3")
```

```
(defstep bflt3 (a b c)
  (let ((casestr (format nil "a=~a AND b=~a AND c=~a"
                          a b c)))
    (spread (case casestr)
      ((then (flatten)(replaces (-1 -2 -3))
             (eval-formula -4))
       (if (< c 3) (let ((nc (+ c 1))) (bflt3 a b nc))
         (if (< b 3) (let ((nb (+ b 1))) (bflt3 a nb 1))
           (if (< a 3) (let ((na (+ a 1))) (bflt3 na 1 1))
             (grind))))))))
  "Checks a^3+b^3 /= c^3 for 0 < a,b,c <= 3"
  "Checking a^3+b^3 /= c^3 for 0 < a,b,c <= 3")
```



```
(defstep bflt3 (&optional (a 1) (b 1) (c 1))
  (let ((casestr (format nil "a=~a AND b=~a AND c=~a"
                          a b c)))
    (spread (case casestr)
    ((then (flatten)(replaces (-1 -2 -3))
           (eval-formula -4))
       (if (< c 3) (let ((nc (+ c 1))) (bflt3 a b nc))
         (if (< b 3) (let ((nb (+ b 1))) (bflt3 a nb))
           (if (< a 3) (let ((na (+ a 1))) (bflt3 na))
             (grind))))))))
  "Checks a^3+b^3 /= c^3 for 0 < a,b,c <= 3"
  "Checking a^3+b^3 /= c^3 for 0 < a,b,c <= 3")
```

```
{-1}  a <= 3
{-2}  b <= 3
{-3}  c <= 3
{-4}  a ^ 3 + b ^ 3 = c ^ 3
  |-----

Rule? (bflt3)
Checking a^3+b^3 /= c^3 for 0 < a,b,c <= 3,
Q.E.D.

Run time  = 0.86 secs.
Real time = 3.29 secs.
```


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`.

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)
  ...
```

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 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 2 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)
```