Strategy Development in PVS

César A. Muñoz

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

PVS Tutorial 2017

PVS Strategies

- Strategies are user-defined proof scripts that conservatively extend the theorem prover capabilities.
- Strategies do not compromise the soundness of the theorem prover (as long as the proof context is accessed as read-only).

Nomenclature

- Proof Rule: Atomic (trusted) prover command, e.g., `split`, `skolem`, `hide`, `metit`, etc.
- Strategy: A proof command that expands into one or more atomic steps, e.g., `grind`, `ground`, etc.
  - Black Box: Proof command that behaves as an atomic step but can be expanded, e.g., `grind`, `ground`, `interval`, etc.
  - Glass Box: A command that is always expanded, e.g., strategy combinators such as `then`, `if`, etc. **Black box strategies alway have a "$"-glass box variant**, e.g., `grind$`, `ground$`, `interval$`, etc.
  - Tactic: A strategy defined inside a proof.

Strategy Language: Basic Steps

- Any proof command, e.g., `(ground)`, `(case ...)`, etc.
- `(skip)` does nothing.
- `(printf format ...)` prints a formatted `message`.
- `(comment message)` adds a persistent comment to the proof branch.
- `(relabel label fnums)` labels formulas `fnums` with `label`.
- `(delabel fnums)` unlabels formulas in `fnums`.
- ...and more!

Examples

Basic Steps

```
{-1}  0 <= deg
{-2}  deg <= 90
  |-----
[1]   sin_0_90(deg) <= sin_deg(deg)

Rule? (skip)
No change on: (skip)
...

Rule? (printf "Hello ~a" "world")
Hello world
...

Rule? (comment "Important branch")
;;; Important branch
...
```

Examples

Basic Steps

```
{-1}  0 <= deg
{-2}  deg <= 90
  |-----
[1]   sin_0_90(deg) <= sin_deg(deg)

Rule? (skip)
No change on: (skip)
...

Rule? (printf "Hello ~a" "world")
Hello world
...

Rule? (comment "Important branch")
;;; Important branch
...
```

Examples

Basic Steps

```
{-1}  0 <= deg
{-2}  deg <= 90
  |-----
[1]   sin_0_90(deg) <= sin_deg(deg)

Rule? (skip)
No change on: (skip)
...

Rule? (printf "Hello ~a" "world")
Hello world
...

Rule? (comment "Important branch")
;;; Important branch
...
```

Examples

Basic Steps

```
{-1}  0 <= deg
{-2}  deg <= 90
  |-----
[1]   sin_0_90(deg) <= sin_deg(deg)

Rule? (skip)
No change on: (skip)
...

Rule? (printf "Hello ~a" "world")
Hello world
...

Rule? (comment "Important branch")
;;; Important branch
...
```

Examples

Basic Steps

```
{-1}  0 <= deg
{-2}  deg <= 90
  |-----
[1]   sin_0_90(deg) <= sin_deg(deg)

Rule? (skip)
No change on: (skip)
...

Rule? (printf "Hello ~a" "world")
Hello world
...

Rule? (comment "Important branch")
;;; Important branch
...
```

Examples

Basic Steps

```
{-1}  0 <= deg
{-2}  deg <= 90
  |-----
[1]   sin_0_90(deg) <= sin_deg(deg)

Rule? (skip)
No change on: (skip)
...

Rule? (printf "Hello ~a" "world")
Hello world
...

Rule? (comment "Important branch")
;;; Important branch
...
```

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

- `(try step step1 step2)`:
  Tries `step`, if it *skips* or *fails*, applies `step2`. Otherwise, applies `step1`.
- `(else step1 step2)`:
  Tries `step1`, if it *skips* or *fails*, applies `step2`. Otherwise, skips.
- `(fail)`:
  Fails the current goal and reaches the innermost `try`.
- A failure that is not caught propagates within the scope of a black-box strategy and then skips.
- Failures propagate beyond glass-box strategies.

Backtracking Example

- What does `(else (then (grind) (fail)) (skip))` do ?
- It either discharges the current sequent or does nothing.

```
test :

  |-----
{1}   x * x + x >= 1
Rule? (else (then (grind)(fail)) (skip))
...
No change on: (skip)
test :

  |-----
[1]   x * (1 + x) >= 1
```

Backtracking Example

- What does `(else (then (grind) (fail)) (skip))` do ?
- It either discharges the current sequent or does nothing.

```
test :

  |-----
{1}   x * x + x >= 1
Rule? (else (then (grind)(fail)) (skip))
...
No change on: (skip)
test :

  |-----
[1]   x * (1 + x) >= 1
```

Creating Fresh Labels

- `(with-fresh-labels bindings steps)`:
  Creates fresh labels and binds them to formulas specified in `bindings`. Then, sequentially applies `steps` to all branches. All created labels are removed before the strategy exits. `bindings` is s list of the form `((var1 fnum1) ...(varn fnumn))`
- `with-fresh-labels@` is a variant of `with-fresh-labels` that applies `steps` to the main branch.
- Example:

```
(with-fresh-labels
  ((l 1) (m -1))
  (inst? l :where m))
```

Creating Fresh Names

- `(with-fresh-names bindings steps)`:
  Creates fresh names and binds them to expressions specified in `bindings`. Then, sequentially applies `steps` to all branches. All created names are removed before the strategy exits. `bindings` is s list of the form `((var1 fnum1) ...(varn fnumn))`
- `with-fresh-names@` is a variant of `with-fresh-names` that applies `steps` to the main branch.
- Example:

```
(with-fresh-names
  ((e "x+2") (f "sqrt(x)"))
  (inst 1 e f))
```

Writing your Own Strategies

- New strategies are defined in a file named `pvs-strategies` in the current context.
- PVS automatically loads this file every time the theorem prover is invoked.
- The `IMPORTING` clause automatically loads any file `pvs-strategies` in the importing chain.

Strategy Definitions

- `defstep` defines a black-box strategy and its glass-box $-form:

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

- `defhelper` defines a black-box strategy that is excluded from the standard user interface:

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

- `defstrat` defines a glass-box strategy:

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

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()
  (else (then (grind)(fail)) (skip))
  "Tries GRIND. UNDO, if it doesn't discharge
   the sequent"
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()
  (else (then (grind)(fail)) (skip))
  "Tries GRIND. UNDO, if it doesn't discharge
   the sequent"
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()
  (else (then (grind)(fail)) (skip))
  "Tries GRIND. UNDO, if it doesn't discharge
   the sequent"
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()
  (else (then (grind)(fail)) (skip))
  "Tries GRIND. UNDO, if it doesn't discharge
   the sequent"
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()
  (else (then (grind)(fail)) (skip))
  "Tries GRIND. UNDO, if it doesn't discharge
   the sequent"
  "Trying GRIND")
```

Example: try-grind

In pvs-strategies:

```
(defstep try-grind ()
  (else (then (grind)(fail)) (skip))
  "Tries GRIND. UNDO, if it doesn't discharge
   the sequent"
  "Trying GRIND")
```

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

Rule? (help try-grind)
(try-grind/$) :
Tries GRIND. UNDO, if it doesn't discharge the sequent

Rule? (try-grind)
No change on: (try-grind)
test :

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

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

Rule? (help try-grind)
(try-grind/$) :
Tries GRIND. UNDO, if it doesn't discharge the sequent

Rule? (try-grind)
No change on: (try-grind)
test :

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

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

Rule? (help try-grind)
(try-grind/$) :
Tries GRIND. UNDO, if it doesn't discharge the sequent

Rule? (try-grind)
No change on: (try-grind)
test :

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

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

Rule? (help try-grind)
(try-grind/$) :
Tries GRIND. UNDO, if it doesn't discharge the sequent

Rule? (try-grind)
```
No change on: (try-grind)
```
test :

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

```
test.1 :

{-1}  x >= 0
  |-----
[1]   x * (1 + x) >= 0

Rule? (try-grind)
Trying GRIND,

This completes the proof of test.1.
```

```
test.1 :

{-1}  x >= 0
  |-----
[1]   x * (1 + x) >= 0

Rule? (try-grind)
```
Trying GRIND,

```
This completes the proof of test.1.
```

Glass-Box Strategies

- `try-grind` is a black-box strategy. Therefore, it is saved in the proof even when it skips.
- When `try-grind` skips, it would be better not to save the command `grind`, which is expensive.
- Two alternatives:
  - Use `try-grind$` instead of `try-grind`.
  - Define `try-grind` as a glass-box strategy:

```
(defstrat try-grind ()
  (else (then (grind)(fail)) (skip))
  "Tries GRIND. UNDO, if it doesn't discharge
   the sequent")
```

Example: Defining A Finite Loop Combinator

In pvs-strategies:

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

```
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
```

```
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
```

Tactic Definitions

- A *tactic* is a strategy defined inside a proof (arguments are optional):

```
(deftactic name args step)
```

- The scope of a tactic is the branch where it is defined.
- Example:

```
Rule? (deftactic mytactic (the (flatten)(split)(assert)))
Defining local tactic mytactic,
...

Rule? (mytactic)
...
```

Tactic Definitions

- A *tactic* is a strategy defined inside a proof (arguments are optional):

  ```
  (deftactic name args step)
  ```

- The scope of a tactic is the branch where it is defined.
- Example:

```
Rule? (deftactic mytactic (the (flatten)(split)(assert)))
Defining local tactic mytactic,
...

Rule? (mytactic)
...
```

Tactic Definitions

- A *tactic* is a strategy defined inside a proof (arguments are optional):

  ```
  (deftactic name args step)
  ```

- The scope of a tactic is the branch where it is defined.
- Example:

```
Rule? (deftactic mytactic (the (flatten)(split)(assert)))
```
Defining local tactic mytactic,
```
...

Rule? (mytactic)
...
```

Tactic Definitions

- A *tactic* is a strategy defined inside a proof (arguments are optional):

  ```
  (deftactic name args step)
  ```

- The scope of a tactic is the branch where it is defined.
- Example:

```
Rule? (deftactic mytactic (the (flatten)(split)(assert)))
Defining local tactic mytactic,
...

Rule? (mytactic)
...
```

Level of Strategy Development

Easy Repetitive tasks, e.g., `(flatten)(assert)(replace -1)...(flatten)(assert)(replace -1)`.

Medium Programatic tasks, e.g., `(case "0 <= n AND n < 1")`, ..., `(case "1023 <= n AND n < 1024")`.

Advanced Control flow, e.g., implementation of a new proof combinator.

Expert Proof search, e.g., implementation of a decision procedure or an heuristic method.

Using Lisp to Access PVS Proof Context

- 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

- `(extra-get-formula fnum)` retrieves the formula `fnum` from the current context.
- `(extra-get-expr exprloc)` retrieves the expression referenced by `exprloc`.
- `(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.

Fresh Labels and Names

- When new names and labels are needed in a strategy, fresh identifiers should be used to avoid clashes with existing ones.
- (`freshname prefix`): Creates a valid name, with a given `prefix`, that is fresh in the current sequent.
- (`freshlabel prefix`): Creates a valid formula label, with a given `prefix`, that is fresh in the current sequent.
- Caveat: Fresh labels and names should be removed before exiting the scope where they were created since there is no guarantee that the same identifiers will be created when the proof is rerun.

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 (extra-get-formula 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)
  ...
```

Example

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

Rule? (lisp (show (extra-get-formula 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)
  ...
```

Strategy Development Pitfalls

- Strategies may not terminate, e.g.,

  ```
  Rule? (repeat (case "0=0"))
  ```

- Strategies may be non-deterministic, e.g.,

  ```
  Rule? (let ((n (freshname "n"))) (name n "10"))
  ```

- Non-deterministic black-box strategies are not robust, i.e., they may fail when the proof is rerun.
- If non-determinism is unavoidable, use glass-box strategies.
- If fresh identifiers are needed, use the robust glass-box strategies `with-fresh-labels` and `with-fresh-names`.

Strategy Development Pitfalls

- Strategies may not terminate, e.g.,

  ```
  Rule? (repeat (case "0=0"))
  ```

- Strategies may be non-deterministic, e.g.,

  ```
  Rule? (let ((n (freshname "n"))) (name n "10"))
  ```

- Non-deterministic black-box strategies are not robust, i.e., they may fail when the proof is rerun.
- If non-determinism is unavoidable, use glass-box strategies.
- If fresh identifiers are needed, use the robust glass-box strategies `with-fresh-labels` and `with-fresh-names`.

Strategy Development Pitfalls

- Strategies may not terminate, e.g.,

  ```
  Rule? (repeat (case "0=0"))
  ```

- Strategies may be non-deterministic, e.g.,

  ```
  Rule? (let ((n (freshname "n"))) (name n "10"))
  ```

- Non-deterministic black-box strategies are not robust, i.e., they may fail when the proof is rerun.
- If non-determinism is unavoidable, use glass-box strategies.
- If fresh identifiers are needed, use the robust glass-box strategies `with-fresh-labels` and `with-fresh-names`.

Strategy Development Pitfalls

- Strategies may not terminate, e.g.,

  ```
  Rule? (repeat (case "0=0"))
  ```

- Strategies may be non-deterministic, e.g.,

  ```
  Rule? (let ((n (freshname "n"))) (name n "10"))
  ```

- Non-deterministic black-box strategies are not robust, i.e., they may fail when the proof is rerun.
- If non-determinism is unavoidable, use glass-box strategies.
- If fresh identifiers are needed, use the robust glass-box strategies `with-fresh-labels` and `with-fresh-names`.

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`.