

Introduction to PVS

Proving with computer assistance

Adam Koprowski

22 May 2008

A.Koprowski@tue.nl

HG 6.78

PVS: Prototype Verification System

<http://pvs.csl.sri.com>

Specification and verification system consisting of:

- formal specification language,
- model checker,
- theorem prover,
- documentation, administrative tools etc.

Both academic and industrial:

- verification of Javacard applets (LOOPtool, Nijmegen),
- hardware verification (microprocessors, ATM switches),
- protocol specification and verification,
- formal mathematics,
- safety-critical systems (NASA)
- and many more:

`http://pvs.csl.sri.com/users.shtml`

- Downloadable from `http://pvs.csl.sri.com` (Windows not supported!).
- Last version: 4.1.
- Available on `svstud` (via Exceed).
- Emacs interface.

PVS: the system & its logic

PVS: the system

- Implemented in LISP (more than 50.000 lines).
- Theories written and edited in text files (`*.pvs`).
- Proofs created interactively and saved as LISP data-structure (`*.prf`).

PVS: the logic

- Based on extensions to typed λ -calculus
- and *classical*, typed higher-order logic.
- Extensions allow for subset types.

Reliability: unlike Coq, PVS does not have a small kernel (**de Bruijn principle**)

\implies in previous version, $0 = 1$ has been proven

- Type variables: $T : \text{Type}$, $T : \text{Type}^+$.
- Base types: `bool`, `nat`, `real`.
- Abstract data-types: `Stack`, `List`, `Tree`.
- Function types: $[\text{int}, \text{nat} \rightarrow \text{bool}]$.
- Enumeration types: $\{\text{red}, \text{green}, \text{blue}\}$.
- Tuple types: $[A, B]$.
- Dependent record types: $[\# a : A, b : B(b) \#]$.
- Subset types: $\{i : \text{nat} \mid i > 1\}$.

- Basic expressions:

`TRUE : bool 0, 23 + 5, 17 * 10 : int`

- Function abstraction and application:

`(LAMBDA (i, j : nat) : i + j) : [nat, nat -> nat]
f(i, j)`

- Logic:

`AND, OR, NOT, IMPLIES, IFF, =, / =, FORALL, EXISTS`

- Conditionals:

`IF c THEN e1 ELSE e2 ENDIF`

- Records:

`point WITH ['x := 24]`

- Subtypes:

`Interval(m, n : int) : TYPE = {i : int | m <= i <= n}
/ : [int, {n : int | n / = 0} -> int]`

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

This recursive definition generates two type checking conditions:

- for type-consistency: $IF\ n \neq 0\ THEN\ n - 1 \geq 0$
- for termination: $IF\ n \neq 0\ THEN\ n - 1 < n$

Such conditions are called **TCCs** (Type Checking Conditions).
They:

- are generated for recursive definitions and subtypes and
- most of them can be automatically discarded by PVS.

Type-checking in PVS is not decidable!

- PVS developments are organized into **theories**.
- Theories consist of definitions, declarations, axioms and lemmas.
- Theories can be parameterized.
- Prelude contains a (large) number of predefined theories.

```
wf_induction [T : TYPE, <: (well_founded?[T])] : THEORY  
  BEGIN  
    wf_induction : LEMMA ...  
  END wf_induction
```

PVS sequents

PVS proof obligations are presented as **sequents**:

$$\begin{array}{l} \{-1\} \quad A \\ \{-2\} \quad B \\ |----- \\ [1] \quad S \\ [2] \quad T \end{array}$$

The sequent stands for: $A \wedge B \implies S \vee T$ with:

- negatively numbered antecedents/assumptions above the line,
- positively numbered consequents/goals below the line.
- PVS maintains a proof tree of such sequents.

Note that the following two sequents are equivalent!

$$\begin{array}{l} \{-1\} \quad A \\ |----- \\ \dots \end{array} \quad \text{and} \quad \begin{array}{l} \dots \\ |----- \\ [1] \quad \text{NOT } A \end{array}$$

- `(undo)` - undo the last step in the proof.
- `(quit)` - quit the current proof.
- `(postpone)` - go to the next proof obligation.
- `(help)`, `(help postpone)` - get help.

Comparison of tactics

Coq	PVS
intro, intros	(flatten), (skolem)
apply	(lemma), (use)
unfold	(expand)
simpl	(beta), (simplify)
induction	(induct), (induction-and-simplify)
auto, tauto	(grind), (prop), (assert)
rewrite	(rewrite), (replace)
Undo	(undo)

```
      {-1} A
{-1} A AND B      (flatten -1)      {-1} A
|-----          |-----          {-2} B
...              ...
```

```
...
|-----          (flatten)          |-----
[1] A OR B      [1] A
                [2] B
```

```
...
|-----          (flatten)          |-----
[1] A => B      [-1] A
                [1] B
```

```

...
|----- (split 1) |----- and |-----
[1] A AND B       [1] A           [1] B

```

```

{-1} A OR B       {-1} A       {-1} B
|----- (split -1) |----- and |-----
...               ...           ...

```

Tactics for propositional logic

- `(flatten [fnum])` - flatten assumptions `(A AND B)` and goals `(A OR B)`.
- `(split [fnum])` - split proof obligations based on assumptions `(A OR B)` or goals `(A AND B)`.
- `(case "formula")` - case distinction on the formula, ex. `(case "x > 0")`.
- `(lift-if [fnum])` - lift branching to the top level of the formula, i.e.: `f(IF b THEN e1 ELSE e2 ENDIF)` becomes: `IF b THEN f(e1) ELSE f(e2) ENDIF`.
- `(prop)` - automatic strategy for propositional logic.

```

...
|----- (skolem! 1) |-----
[1] FORALL (x:A) : P(x)          [1] P(x!1)

```

```

{1} EXISTS (x:A) : P(x)          P(x)
|----- (skolem -1 "x") |-----
...                               ...

```

Notes:

- `(skosimp)` does `(skolem!)` and `(flatten)`.
- `(skosimp*)` does that repeatedly.

\dots
 $| \text{-----} (\text{inst } 1 \text{ "w"}) | \text{-----}$
 $[1] \text{ EXISTS } (x:A) : P(x) \qquad [1] P(w)$

$\{-1\} \text{ FORALL } (x:A) : P(x) \qquad \{-1\} P(w)$
 $| \text{-----} (\text{inst } -1 \text{ "w"}) | \text{-----}$
 $\dots \qquad \dots$

Tactic for predicate logic

- `(skolem! [fnum])` - introduce skolem constants for goals `FORALL (x) P` or assumptions `EXISTS (x) P`.
- `(skolem [fnum] "name_1" ... "name_N")` - allows you to choose the name(s) for those constants.
- `(inst [fnum] "exp_1" ... "exp_N")` - instantiate assumptions `FORALL (x) P` or provides witness for goal `EXISTS (x) P`.
- `(inst? [fnum])` - PVS will guess the expression; works only for very simple cases.

Tactics for equational reasoning

- `(expand "name" [fnum] [n])` - expand n 'th occurrence of `name` in formulas `fnum` (default: all occurrences in all formulas)
- `(replace fnum [fnums] LR/RL)` use assumptions of the form $l = r$ to replace `l` by `r` in formulas `fnums`.
- `(assert)` - built-in decision procedure for equality.

Using lemmas:

- `(lemma "name")` - add lemma `name` as an assumption.
- `(rewrite "name" [fnums] RL)` - like `(replace)` but using a lemma instead of an assumption.

Induction:

- `(induct "n")` - perform induction on variable `n`. The goal should be of the form `FORALL (... , n:nat, ...): P(n)`.
- `(induct-and-simplify "n")` - ditto, but performs simplifications after applying induction.

Automation:

- `(prop)` - decision procedure for propositional logic.
- `(assert)` - decision procedure for equality logic.
- `(grind)` - a powerful “catch-all” automation strategy (note: if it does not succeed completely it can transform the goal to unprovable/unreadable form).

PVS demo

Acknowledgments to Erik Poll.

This PVS introduction (and the demo file) are based on his one-day introduction course given in Eindhoven during IPA days.

```
http://www.cs.ru.nl/~erikpoll/Teaching/PVS/  
index.html
```