

IPA Course on Formal Methods

An introduction to theorem proving using PVS

Erik Poll

**Digital Security group
Radboud University Nijmegen**

What is a theorem prover and why would you use one?

A first taste of using a theorem prover

Rough plan for the day:

- Introduction to PVS: specification language + prover
- Exercise 1
- Lunch
- A typical computer science application:
state machines in PVS
- Exercise 2
- Demo: use of *automated* theorem proving for program
verification (ESC/Java2)

What is a theorem prover?

A theorem prover is a tool for logical reasoning, like a calculator is a tool for arithmetic.

Theorem provers are *not* as mature or widely used (yet?) as calculators; it takes considerable expertise to use one.

What is a theorem prover?

Theorem provers such as **PVS**, **Isabelle/HOL**, **Coq** are capable of expressing *any piece of mathematics or computer science*. This involves

1. **modelling/specification/definition** of constructs involved
2. **proving** results about them, interactively and/or automatically.

‘theorem prover’ aka ‘proof assistant’ is a bit of a misnomer, as it ignores the first part, which is already interesting in itself.

The first theorem prover was **AUTOMATH**, by de Bruijn & co here at TU/e in 1970’s

There are also less expressive theorem provers (eg. first-order theorem provers or SAT solvers) which provide better automation of proofs.

Why use a theorem prover?

It can give the **highest level of confidence** in correctness, but at **very high cost**: lots of effort by experts

So primarily of interest for applications where cost of failure is highest:

- **safety-critical** systems (eg. Ariane 5)
- **security-critical** systems (eg. Chipknip software)
- **mass-produced** products (eg. Pentium bug)

Example applications

- hardware
- algorithms, esp. distributed or real-time algorithms
- (security) protocols
- programming language theory
formalising programming languages: their type systems, semantics, or program verification logics
- mathematics
eg Four Color Theorem in Coq (Georges Gonthier, 2004)

Theorem proving vs model checking

- + Theorem provers are more expressive**
- + Model checkers can run into limitations due to the state explosion problem; theorem provers don't, and can cope with infinite state spaces.**
 - Model checker can verify dining philisophers for 4 philosophers, theorem proving can do it for arbitrary number.
- Theorem provers are more labour-intensive**
- Model checkers provide better feedback.**
 - Failed modelcheck attempt concrete counterexample trace.
 - Failed proof attempt may be due to missing lemma (invariant), or wrong proof strategy.
- **Model checking can be used as part of theorem proving; indeed, PVS includes a model checker**

The PVS specification language

PVS specification language

The PVS specification language consists of

- a **typed lambda calculus**, similar to functional programming languages à la Haskell or ML, but more expressive

Eg. `reverse : [List -> List]`

- a **typed higher-order logic** on top of this

Eg. `(FORALL (x:List): rev(rev(x)) = x)`

Many theorem provers, notably Isabelle and Coq, are based on similar typed languages, if slightly less baroque.

Types

- Base types `bool, int, real`
- Function types `[bool,int -> int]`
- Enumeration types `{red, white, blue}`
- Tuple types `[A,B]`
- Record types `[# x:int, y:int #]`
- Algebraic datatypes (ADTs) `Stack, List, Tree`
- Subset types `{ i:int | i >= 0 }`

Subset types are peculiar to PVS, and do not exist in for instance Isabelle or Coq.

Expressions

- **basic expressions**

`TRUE, FALSE: bool`

`0, 1, -23, 23+5, 24*5 : int`

- **function abstraction and application**

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

`f(i,j)`

- **tuples and projection**

`(1,true) : [int,bool]`

`tup`2 , proj_2(tup)`

- **records and projection**

`(# x:=1, y:=4 #) : [# x:int, y:int #]`

`point`x`

More expressions

- **let-expressions**

```
LET name = e1 IN e2
```

- **conditionals**

```
IF c THEN e1 ELSE e2 ENDIF
```

```
COND c1 -> e1, ..., cn -> E2 ENDCOND
```

```
COND c1 -> e1, ..., ELSE -> E2 ENDCOND
```

- **record and function updates**

```
point WITH [ `x:=24 ]
```

```
f WITH [ (0):=1 ]
```

Declarations and definitions

- declarations

```
i : int  
A : TYPE
```

- definitions

```
twentyeight : int = 25+3  
Point: TYPE = (# x:int, y:int #)  
p:Point = (# x:=1, y:=4 #)  
square: [int->int] =(LAMBDA (n:nat): n*n)  
pred(n:int) : int = n-1
```

A typed higher-order logic, with

- conjunction, disjunction, negation, implication

AND OR NOT IMPLIES IFF

Alternative syntax: $\&$, \Rightarrow for AND, IMPLIES

- (in)equality

= / =

- (typed) universal/extensional quantification

FORALL EXISTS

Eg. $(\text{FORALL } (i, j : \text{int}) : i > 0 \text{ AND } j > 0 \Rightarrow i * j \neq 0)$

Theories

Specifications are built from **theories** with definitions, declarations and named axioms and lemmas. Eg.

```
MyFirstTheory: THEORY
  BEGIN
    square(n:nat): nat = n*n
    square_nondecreasing: LEMMA
      FORALL (n:nat) : square(n) >= n
    sqrt : [nat-> nat]
    axiom_sqrt: AXIOM
      FORALL (n:nat) :
        square(sqrt(n)) <= n AND n < square(sqrt(n)+1)
  END MyFirstTheory
```

You could also *define* `sqrt` and turn the axiom into a lemma.
(This would be better. Why?)

Theories

Trick to avoid lots of explicit type information

```
SquareTheory : THEORY
BEGIN
  n: VAR nat      % ie. n will range over nat
  square(n) : nat = n * n
  square_nondecreasing : LEMMA
    FORALL (n:nat) : square(n) >= n
  ...
```

Theories can be parameterized, eg

```
stack[A:Type] : THEORY
  ...
```

Recursion

All recursive functions *must be shown to terminate* by supplying a measure function.

```
fac(n:nat) : RECURSIVE nat =  
  IF n=0 THEN 1 ELSE n*f(n-1) ENDIF  
  MEASURE n
```

`fac` is only well-typed if

- measure decreases, ie. $n \neq 0 \Rightarrow n-1 < n$
- measure remains non-negative, ie. $n \neq 0 \Rightarrow n-1 \geq 0$

These are the so-called *type checking conditions (TCCs)*

Here PVS differs from typical functional programming languages!

TCCs

Expressions are only well-typed after all type checking conditions (TCCs) have been proven.

- type checking is undecidable, in principle**
- + but usually PVS prover discharges most TCCs fully automatically, in practice**

Warning: unsolved TCCs may leave inconsistencies in your theories.

Subset types

Subset types, eg

```
nat : TYPE = { i:int | i >= 0 }
```

```
subrange(n,m:int) : TYPE
```

```
    = { i:int | n <=i & i <= m }
```

are useful for partial operations, e.g. division

```
/ : [int, { n:int | n /= 0 } -> int]
```

and also give rise to type checking conditions (TCCs)

Eg

```
average = sum / numbers : int
```

is only well-typed if `numbers /= 0`.

The PVS prover

The PVS prover

Once we have defined – and type-checked! – a theory, we can prove any lemmas and theorems it contains.

Lemmas can be done in any order; PVS keeps track of what has been proved.

Proving is done interactively, by the user giving commands, **tactics**, to the PVS prover.

A tactic

- either solves a proof obligation, or
- gives rise to one or more new, hopefully simpler, proof obligations.

Sequents

PVS proof obligations are *sequents* of the form

[-1] P

[-2] Q

[-3] R

{ 1 } S

{ 2 } T

Intuitive meaning: (P AND Q AND R) => (S OR T)

- negatively numbered *antedecents/assumptions* above line,
- positively numbered *consequents/goals* below line

PVS maintains a *proof tree* of such sequents.

Tactics

The user interacts with the prover by **tactics** (which are actually LISP expressions).

There are *many* tactics, and you can define additional ones yourself.

Below we give an overview of the more common ones.

A full list is included in the 'PVS Prover Guide'.

Basic tactics

- `(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

Propositional logic

The proof obligation

{1} P => Q

after (`flatten 1`) becomes

[-1] P

{1} Q

You can omit the argument -1 and let PVS guess this.

Useful shorthand : `TAB f`

Propositional logic

```
[ -1 ] P1 AND P2
```

```
-----
```

```
...
```

after (`flatten -1`) becomes

```
[ -1 ] P1
```

```
[ -2 ] P2
```

```
-----
```

```
...
```

Propositional logic

Similarly,

...

{1} Q1 OR Q2

after (`flatten 1`) becomes

...

{1} Q1

{2} Q2

Propositional logic

```
[1] P1 OR P2
```

```
-----
```

```
...
```

after (`split 1`) results in two proof obligations

```
[1] P1
```

```
-----
```

```
...
```

```
[1] P2
```

```
-----
```

```
....
```

This also works for antecedents of the form

```
IF c THEN e1 ELSE e2 ENDIF
```

resulting in distinction of the cases `c` and `NOT c`.

Propositional logic (split)

Similarly,

...

{1} P1 AND P2

after (`split 1`) results in two proof obligations

.....

[1] P1

.....

[1] P2

Note: many tactics can often be used on dual constructs – eg AND and OR – on different sides of the line.

Tactics for propositional logic

- `(flatten [fnum])`
flatten antecedents ($P1 \text{ AND } P2$)
and consequents ($Q1 \text{ OR } Q2$) and ($Q1 \Rightarrow Q2$)
- `(split [fnum])`
split based on consequent ($P1 \text{ AND } P2$)
or antecedent ($Q1 \text{ OR } Q2$) or ($\text{IF } \dots$)
- `(case "formula")`
case distinction on formula, eg (`case "x>0"`)
- `(lift-if [fnum])`
replace $f(\text{IF } b \text{ THEN } e1 \text{ ELSE } e2 \text{ ENDIF})$
by $\text{IF } b \text{ THEN } f(e1) \text{ ELSE } f(e2) \text{ IF}$
typically as precursor to splitting
- `(prop)` automatic strategy for propositional logic

The argument [fnum] is optional; if you omit it PVS chooses one.

Predicate logic

...

{1} FORALL (x:A) P(x)

after (skolem! 1) becomes

...

{1} P(x!1)

(skolem 1 "name") uses name instead of x!1

(skosimp) does (skolem!) and (flatten)

(skosimp*) does this repeatedly

Predicate logic

```
[1] EXIST (x) P(x)
```

...

after (`skolem! -1`) becomes

```
[-1] P(x!1)
```

...

Here `x!1` is the so-called **witness**

(`skolem 1 "name"`) calls the witness name

Predicate logic

Proof obligations

[fnum] FORALL (x) P(x)	...
-----	-----

...	{fnum} EXISTS (x) P(x)
-----	------------------------

after (inst [fnum] "expr") become

[fnum] P(expr)	...
-----	-----

...	{fnum} P(expr)
-----	----------------

(inst? [fnum]) lets PVS guess expr; only works in simple cases!!

(inst-cp ...) leaves copy of the quantification

Tactics for predicate logic

- `(skolem! [fnum])`
introduces skolem constants for consequent
(`FORALL(x) P`) or antecedent (`EXIST(x) P`)
- `(skolem [fnum] "name1" ... "namen")`
let's you choose name of these constants.
- `(inst [fnum] "expr1" ... "exprn")`
instantiates antecedent (`FORALL(x) P`) or provides
witness for consequent (`EXIST(x) P`)
- `(inst? [fnum])`
lets PVS guess the expression; only works in simple
cases!

Tactics for equational reasoning

- `(expand "name" [fnum] [n])`
expand n th occurrence of `name` by its definition in `fnum`; the default is all occurrences
Shorthand: put cursor on `name` and type `TAB r`
- `(replace fnum [fnums] LR)`
use antecedent `fnum` of the form $l = r$, to replace occurrences of `l` by `r` in `fnums`.
Shorthand: `TAB r`, which interactively ask for all the options.
- `(replace fnum [fnums] RL)`
idem, but in other direction
- `(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 lemma instead of antecedent

Tactics for induction

- `(induct "n")`
for goal of the form `(FORALL (... ,n:nat, ...) P)`
- `(induct-and-simplify "n")`
Idem, but combined with simplification

Exercise 1

Load the file `pvs_intro.pvs` into PVS and start proving!

This involves basic propositional, equational, and equational logic.

Instructions & hints are included in file.

Background for exercise 2

**Something with a more computer science flavour:
deterministic state machines and refinements**

Deterministic state machines

```
machine[S, I, O: TYPE+]: THEORY
BEGIN
  machine: TYPE =
    [# delta : [S, I -> S], output: [S -> O] #]

  m: VAR machine

  trace(m)(s:S, inp:sequence[I])(n:nat): RECURSIVE S =
    IF n = 0 THEN s
      ELSE trace(m)(delta(m)(s, inp(n)), inp)(n - 1)
    ENDIF
  MEASURE n
```

Sequences over type I are elements of type $[\text{nat} \rightarrow I]$

Trace refinement

Let $m1 : machine[S1, I, O]$ and $m2 : machine[S2, I, O]$

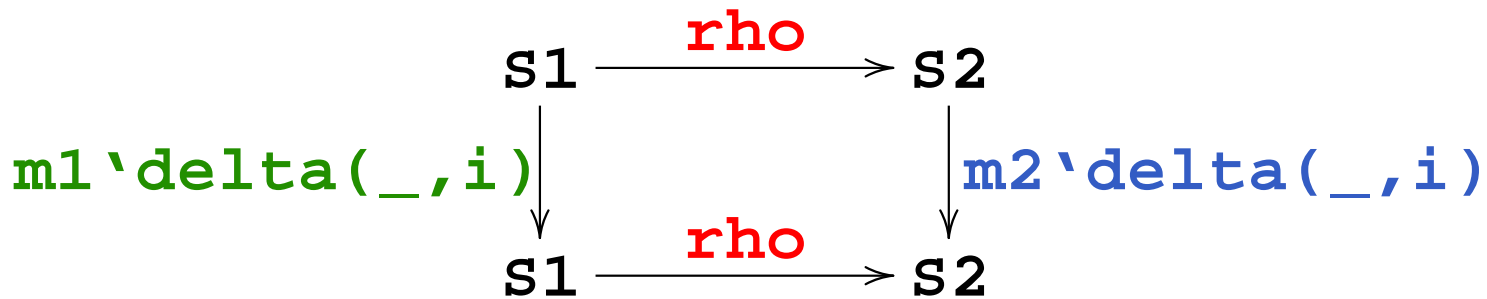
$\rho : [S1 \rightarrow S2]$ gives a trace-refinement from $m1$ to $m2$

iff

FORALL ($i : I, s1 : S1$)

$$\rho(m1 \backslash \text{delta}(s1, i)) = m2 \backslash \text{delta}(\rho(s1), i)$$

ie.



commutes

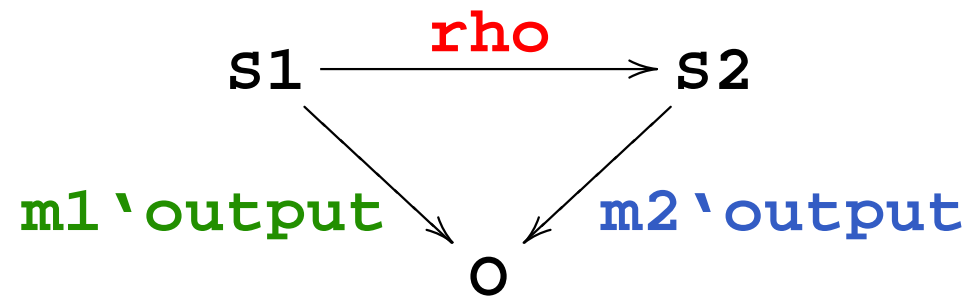
Output refinement

rho: [S1 -> S2] gives an output-refinement from **m1** to **m2**
iff

FORALL (s1 : S1)

$$\mathbf{m1} \text{ `output}(s1) = \mathbf{m2} \text{ `output}(\mathbf{rho}(s1))$$

ie.



commutes

Tripmeter: informal spec

A tripmeter, with input type `real` and output type `boolean`, monitors if input stays between thresholds `HI` and `LO`.

The meter is *tripped* – ie. *outputs true* – iff

1. `input > HI`
2. `input < LO`
3. `input > HI - D` but meter was tripped in previous state, because of conditions 1 or 3.
4. `input < LO + D` but meter was tripped in previous state, because of conditions 2 or 4.

for certain $0 \leq D, LO < HI, LO+D \leq HI-D$

A formal model τ_{m1}

Capture state as two booleans

```
state1: TYPE = [# LT, HT: bool #]
```

where

- HT is true iff meter tripped due to high input (1 & 3)
- LT is true iff meter tripped due to low input (2 & 4)

Could we just use one boolean, to record if meter is tripped?

Can both booleans become true at the same time?

An implementation t_{m2}

Implementation t_{m2} differs only slightly from the model

t_{m1} :

capture state using 3-element enumeration type

```
state2: TYPE = {Ht, Lt, Ut}
```

where

- Ht means meter tripped due to high input (1 & 3)
- Lt means meter tripped due to low input (2 & 4)
- Ut means meter untripped

What is the relation with t_{m1} ?

Another formal model t_{m3}

Capture state as a list of reals

```
state3: TYPE = list[real]
```

Intuition: the state records the history of all inputs.
So transition function just adds input to list.

What should the output function be?

What is the relation with t_{m2} ?

Another formal model τ_{m3}

Output TRUE iff for state $x: \text{list}[\text{real}]$

$(\text{EXISTS}(i): i < \text{length}(x) \ \& \ \text{nth}(x,i) > \text{HI} \ \& \\ \text{FORALL}(j): j < i \Rightarrow \text{HI}-D < \text{nth}(x,j) \ \& \ \text{nth}(x,j) \leq \text{HI})$

OR

$(\text{EXISTS}(i): i < \text{length}(x) \ \& \ \text{nth}(x,i) < \text{LO} \ \& \\ \text{FORALL}(j): j < i \Rightarrow \text{LO} < \text{nth}(x,j) \ \& \ \text{nth}(x,j) \leq \text{LO}+D)$

Here $\text{nth}(x, j)$ is the j -th element of x .
The first element of x is the latest input.

What does $(\text{EXISTS}(i): i < \text{length}(x) \ \& \ \text{nth}(x,i) > \text{HI} \ \& \\ (\text{FORALL}(j): j < i \Rightarrow \text{HI}-D < \text{nth}(x,j) \ \& \ \text{nth}(x,j) \leq \text{HI}))$
mean?

What is the relation with τ_{m2} ?

Exercise 2

Load files `machine.pvs` and `tripmeter.pvs` into PVS.

Complete definitions and prove lemmas in `tripmeter.pvs`

Automation

PVS philosophy: automate everything that can be automated

- **(prop)**
decision procedure for propositional logic
- **(assert)**
decision procedure for equational logic

**To find out how a tactic work: append a \$, eg. (prop\$)
instead of prop.**

**This is a useful way to discover more tactics that may be useful in
a particular situation**

Some more PVS tactics

There are more powerful tactics than `(prop)` and `(assert)`, eg.

- `(grind)` - the general purpose workhorse

But beware that many automated tactics

- may produce far too many subgoals; then `(undo)`
Moral: if `(grind)` fails to prove a goal, always `(undo)` as it probably did something silly.
- may take too long, or fail to terminate; then interrupt prover with `CTRL-c CTRL-c`, followed by `(restore)`

More about equality

- `(apply-extensionality [fnum])`
replace goal $f=g$ by $\text{FORALL}(x) : f(x)=g(x)$

Also replaces $(\#x:=e1, y:=e2\#) = (\#x:=e3, y:=e4\#)$ by
 $e1=e3$ and $e2=e4$
- `(decompose-equality [fnum])`
idem, but also works on assumptions

Hints for complicated proofs

- **Try to understand what the assumptions/goals mean**
This is often the bottleneck in verifications; ugly PVS syntax can be hard to read
- **Which instantiations of assumptions are useful ?**
- **Which lemmas might be useful?**
- **Carefully expand definitions**
Too much expansion makes things unreadable, so use ("expand
"expr" fnum n) rather than ("expand "expr")
- **Which case distinctions are useful ?**
Many useful case-distinctions can be made by expanding definition,
`lift-ifing` and `spliting`
- **exercise: if automated tactics (eg. induct-and-simplify, grind) succeed, try to do the proof "by hand" without using such automated techniques**
Not just masochism: the improved insight can be crucial once automated tactics fail.

Concluding remarks

- PVS is a very general tool
- Still a *BIG* step from being formal with pencil & paper to being formal in theorem prover
- Experience: specification is easy, verification is difficult,
...relatively speaking
- But, errors often exposed during specification, *not* verification
- Mainly for experts on critical applications and academics

Our work with PVS

Use of PVS in Digital Security group includes

- formalisation of single-threaded Java in LOOP project, incl. denotational semantics and Hoare logic for program verification for specification language JML
- formalisation of semantics of C++ in ROBIN project
- formalisation of scheduling protocol [FMICS'2007] and C++ implementation of ReadersWritersLock [TBA]