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)
• 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 philosophers 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
The PVS specification language consists of

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

  \[ \text{Eg. } \text{reverse} : [\text{List} \to \text{List}] \]

- a typed higher-order logic on top of this

  \[ \text{Eg. } (\forall x: \text{List} : \text{rev} (\text{rev} (x)) = x) \]

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**
  
  \[
  \text{TRUE, FALSE: bool} \\
  0, 1, -23, 23+5, 24\times5 : \text{int}
  \]

- **function abstraction and application**
  
  \[
  (\text{LAMBDA}(i,j: \text{nat}): i+j) : [\text{nat}, \text{nat} \rightarrow \text{nat}]
  \]

  \[f(i,j)\]

- **tuples and projection**
  
  \[
  (1, \text{true}) : [\text{int}, \text{bool}]
  \]

  \[\text{tup} \, \text{\textbackslash 2}, \text{proj} \, \text{\textbackslash 2}(\text{tup})\]

- **records and projection**
  
  \[
  (\# \ x := 1, \ y := 4 \ #) : [\# \ x : \text{int}, \ y : \text{int} \ #]
  \]

  \[\text{point} \, \text{\textbackslash x}\]
More expressions

- **let-expressions**
  
  $$\text{LET name} = e_1 \text{ IN } e_2$$

- **conditionals**
  
  $$\text{IF } c \text{ THEN } e_1 \text{ ELSE } e_2 \text{ ENDIF}$$
  
  $$\text{COND } c_1 \rightarrow e_1, \ldots, c_n \rightarrow E_2 \text{ ENDCOND}$$
  
  $$\text{COND } c_1 \rightarrow e_1, \ldots, \text{ELSE } \rightarrow E_2 \text{ ENDCOND}$$

- **record and function updates**
  
  $$\text{point WITH [ } \text{`x:=24} ]$$
  
  $$\text{f WITH [ (0):=1]}$$
Declarations and definitions

- **declarations**
  
  \[i : \text{int}\]
  
  \[A : \text{TYPE}\]

- **definitions**
  
  \[\text{twentyeight} : \text{int} = 25+3\]
  
  \[\text{Point: TYPE} = (\# x:\text{int}, y:\text{int} \#)\]
  
  \[p: \text{Point} = (\# x:=1, y:=4 \#)\]
  
  \[\text{square} : [\text{int} \to \text{int}] = (\lambda n: \text{nat}: n \times n)\]
  
  \[\text{pred}(n:\text{int}) : \text{int} = n-1\]
A typed higher-order logic, with

- conjunction, disjunction, negation, implication
  
  \text{AND OR NOT IMPLIES IFF}

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

- (in)equality
  
  $=$ / $\neq$

- (typed) universal/extensional quantification
  
  \text{FORALL EXISTS}

Eg. (FORALL ($i, j$: int): $i > 0$ AND $j > 0$ $\Rightarrow$ $i \times j \neq 0$)
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
...

PVS – p.16/54
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/=0 => n-1<n
- measure remains non-negative, ie. n/=0 => n-1>=0

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

Here PVS differs from typical functional programming languages!
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

\[
\text{nat} : \text{TYPE} = \{ i : \text{int} \mid i \geq 0 \} \\
\text{subrange}(n, m : \text{int}) : \text{TYPE} \\
\quad = \{ i : \text{int} \mid n \leq i \& i \leq m \}
\]

are useful for partial operations, e.g. division

\[
/ : [\text{int}, \{ n : \text{int} \mid n \neq 0 \} \rightarrow \text{int}]
\]

and also give rise to type checking conditions (TCCs)

Eg

\[
\text{average} = \text{sum} / \text{numbers} : \text{int}
\]

is only well-typed if \text{numbers} \neq 0.
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 of more new, hopefully simpler, proof obligations.
Sequents

PVS proof obligations are *sequents* of the form

\[
[-1] \quad P
\]
\[
[-2] \quad Q
\]
\[
[-3] \quad R
\]

\[
-----------
\]
\[
\{1\} \quad S
\]
\[
\{2\} \quad T
\]

Intuitive meaning: \((P \ \text{AND} \ Q \ \text{AND} \ R) \Rightarrow (S \ \text{OR} \ T)\)

- negatively numbered *ancedents/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 (\textit{flatten} \ -1) becomes

\([-1]\) \ P1
\[-2\] \ P2
-------------
...

Propositional logic

Similarly,

\[ \{1\} \quad Q_1 \ \text{OR} \quad Q_2 \]

after \((\text{flatten} \ 1)\) becomes

\[ \{1\} \quad Q_1 \quad \{2\} \quad Q_2 \]
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,

\[
\begin{align*}
\{1\} & \text{ P1 AND P2} \\
\text{after (split 1) results in two proof obligations} & \\
\[1\] \text{ P1} & \[1\] \text{ P2}
\end{align*}
\]

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 antedents \((P1 \ AND \ P2)\)
  and consequents \((Q1 \ OR \ Q2)\) and \((Q1 \ => \ Q2)\)

- **(split \[fnum\])**
  split based on consequent \((P1 \ AND \ P2)\)
  or antecedent \((Q1 \ OR \ Q2)\) or \((IF \ ... \))

- **(case "formula")**
  case distinction on formula, eg \((case "x>0")\)

- **(lift-if \[fnum\])**
  replace \(f(IF \ b \ THEN \ e1 \ ELSE \ e2 \ ENDIF)\)
  by \(IF \ b \ THEN \ f(e1) \ ELSE \ f(e2) \ IF\)
  typically as precursor to splitting

- **(prop)** automatic strategy for propositional logic

The argument \[fnum\] is optional; if you omit it PVS chooses one.
After \texttt{(skolem! 1)} becomes
\begin{verbatim}
{1} P(x!1)
\end{verbatim}

\texttt{(skolem 1 "name")} uses name instead of x!1

\texttt{(skosimp)} does \texttt{(skolem!)} and \texttt{(flatten)}

\texttt{(skosimp*)} does this repeatedly
[1] \( \exists (x) \, P(x) \)

-------------

...

After \( \text{skolem! } -1 \) becomes

\([-1] \, P(x!1) \)

-------------

...

Here \( x!1 \) is the so-called \textbf{witness}

\( \text{(skolem } 1 \, "name") \) calls the witness \textbf{name}
Predicate logic

Proof obligations

\[ \text{[fnum]} \ \text{FORALL} \ (x) \ P(x) \quad \ldots \quad \text{----------------------} \quad \text{---------------------} \]

\[ \text{----------------------} \quad \text{---------------------} \quad \{\text{fnum}\} \ \text{EXISTS} \ (x) \ P(x) \]

After \(\text{inst} \ [\text{fnum}] \ "\text{expr}"\) become

\[ \text{[fnum]} \ P(\text{expr}) \quad \ldots \quad \text{---------------------} \quad \text{---------------------} \]

\[ \{\text{fnum}\} \ P(\text{expr}) \]

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

\(\text{inst-cp} \ \ldots\) 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")` lets 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 antedent 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

```plaintext
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 [nat->I]
```

PVS – p.40/54
Trace refinement

Let $m_1: \text{machine}[S_1, I, O]$ and $m_2: \text{machine}[S_2, I, O]$.

$\rho: [S_1 \rightarrow S_2]$ gives a trace-refinement from $m_1$ to $m_2$ iff

$$\forall (i: I, s_1: S_1) \quad \rho(m_1 \: \delta(s_1, i)) = m_2 \: \delta(\rho(s_1), i)$$

ie.

commutes
Output refinement

\( \rho: [S_1 \rightarrow S_2] \) gives an output-refinement from \( m_1 \) to \( m_2 \)
iff

\[
\text{FORALL (s1:S1)} \\
\quad m_1 \text{`output(s1)} = m_2 \text{`output(\rho(s1))}
\]

ie.

\[
\begin{array}{ccc}
S_1 & \xrightarrow{\rho} & S_2 \\
\downarrow & & \downarrow \\
\rightarrow & & \rightarrow \\
\text{m1 `output} & \rightarrow & m_2 \text{`output}
\end{array}
\]

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

for certain $0 \leq D$, $\text{LO} < \text{HI}$, $\text{LO} + D \leq \text{HI} - D$
A formal model $tm_1$

Capture state as two booleans

```plaintext
state_1: 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 $tm2$

Implementation $tm2$ differs only slightly from the model $tm1$: capture state using 3-element enumeration type

$$state2: \text{TYPE} = \{Ht, Lt, Ut\}$$

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 $tm1$?
Another formal model $\text{tm}_3$

Capture state as a list of reals

\[ \text{state}_3: \text{TYPE} = \text{list}[\text{real}] \]

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 $\text{tm}_2$?*
Another formal model tm3

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

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

OR

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

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

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

What is the relation with tm2?
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

- \( (\text{prop}) \)
  decision procedure for propositional logic
- \( (\text{assert}) \)
  decision procedure for equational logic

To find out how a tactic work: append a $, eg. \( (\text{prop}$) instead of \text{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 \texttt{(prop)} and \texttt{(assert)}, eg.

- \texttt{(grind)} - the general purpose workhorse

But beware that many automated tactics

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

- \texttt{(apply-extensionality [fnum])}
  replace goal \( f = g \) by \( \forall x : f(x) = g(x) \)
  Also replaces \((#x:=e1, y:=e2#) = (#x:=e3, y:=e4#)\) by \( e1 = e3 \) and \( e2 = e4 \)

- \texttt{(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 fnun n) rather than ("expand "expr")

- Which case distinctions are useful?
  Many useful case-distinctions can be made by expanding definition, lift-ifing and splitting

- 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]