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?

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,**
  simlar 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
```

Logic

A typed higher-order logic, with

- conjunction, disjunction, negation, implication

  `AND OR NOT IMPLIES IFF`

  Alternative syntax: `&`, `=>` for `AND`, `IMPLIES`

- (in)equality

  `= /=`

- (typed) universal/extensional quantification

  `FORALL EXISTS`

Eg. `(FORALL (i,j:int): i>0 AND j>0 => i*j/=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/=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!

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 of 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 *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 `(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 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.

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 antecendent `(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 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