

Using the Prover I: Basic Commands & Propositional Logic

Ben Di Vito
with prior contributions from Lee Pike

NASA Langley Formal Methods Group

`b.divito@nasa.gov`

November 21, 2007

Sequents 101

Using The Prover

Basics

Basic Proving Commands

Some Proving Examples

Additional Commands

Sequents

Sequent semantics: The conjunction of the *antecedents* above the *turnstile* implies the disjunction of *consequents*.

{-1}	(p => q)	←	<i>antecedent</i>
{-2}	p	←	<i>antecedent</i>
	-----	←	<i>turnstile</i>
{1}	q	←	<i>consequent</i>
{2}	r	←	<i>consequent</i>

Thus, $p \Rightarrow q$ and p entail either q or r .

Why Sequents?

There are many ways to represent proof information. Sequents are attractive because

- ▶ They ease the representation of subproofs.
- ▶ They ease the mechanization of proofs, where possible.
- ▶ They maintain a global picture of the proof at each step. That is, many of the formulas will be irrelevant in a given proof step, but they may be used later.

Terminal Sequents

A PVS proof is a sequence of commands to manipulate sequents.

- ▶ The commands transform sequents into new sequents in correctness-preserving ways.
- ▶ Goal: transform the sequent into a *terminal sequent* – one PVS obviously recognizes as being valid.
 - ▶ An antecedent is false.
 - ▶ A consequent is true.
 - ▶ The same formula is both an antecedent and a consequent.

Sanity check: Why are these “obviously valid?”

On the Prover's Lisp-Based Notation

Proof commands take the form of Lisp S-expressions.

- ▶ Examples: `(flatten)`, `(split -1)`, `(expand "fib")`
- ▶ Commands invoke prover *rules* or *strategies*.
- ▶ Arguments are typically numbers or strings.

Prover Command Documentation

Documentation for each proof command describes its syntax

Syntax	Possible invocations
<code>(copy fnum)</code>	<code>(copy 2)</code> <code>(copy -3)</code>
<code>(skosimp &optional (fnum *) preds?)</code>	<code>(skosimp)</code> <code>(skosimp -3)</code> <code>(skosimp + t)</code>
<code>(induct var &optional (fnum 1) name)</code>	<code>(induct "n")</code> <code>(induct "n" 2)</code> <code>(induct "n" :name "NAT_induction")</code>
<code>(hide &rest fnums)</code>	<code>(hide)</code> <code>(hide 2)</code> <code>(hide -)</code> <code>(hide -3 -4 1 2)</code> <code>(hide -2 +)</code>

Help Commands

Prover has a single help command:

- ▶ Syntax: `(help &optional name)`
- ▶ Provides a short description of each prover command
- ▶ Also a GUI based interface: `M-x x-prover-commands`
- ▶ Example:

Rule? `(help flatten)`

`(flatten &rest fnums):`

Disjunctively simplifies chosen formulas. It eliminates any top-level antecedent conjunctions, equivalences, and negations, and succedent disjunctions, implications, and negations from the sequent.

Control Commands

The prover provides several commands for control flow.

- ▶ Leaving the prover and terminating current proof:
 - ▶ Syntax: `(quit)`
- ▶ Undoing one or more proof steps:
 - ▶ Syntax: `(undo &optional to)`
 - ▶ Undoes effects of recent proof steps and restores an earlier state
 - ▶ Can undo a specified number of steps or to a specific label in the proof tree.
 - ▶ Example: `(undo 3)` undoes previous 3 steps.
 - ▶ Prunes the proof tree (deletes parallel branches).
 - ▶ Limited redo capability: `(undo undo)` undoes last `undo`.

Changing Branches in a Proof

It is possible to defer work on one branch and pursue another.

- ▶ Postponing the current proof branch:
 - ▶ Syntax: `(postpone &optional print?)`
 - ▶ Places current goal on parent's list of pending subgoals
 - ▶ Brings up next unproved subgoal as the current goal
 - ▶ The Emacs command `M-x siblings` shows the sibling subgoals of the current goal in a separate emacs buffer.

Sample proof tree:

```
(""  
  (split)  
  (("1" (flatten) (skosimp*) (inst?))  
   ("2" (flatten) (skosimp*) (inst?))))
```

Propositional Rules

Several commands are available to manipulate the current sequent.

- ▶ Sequent flattening is the most basic operation:
 - ▶ Syntax: `(flatten &rest fnums)`
 - ▶ Normally applied to entire sequent (no `fnums` given)
- ▶ Sequent splitting is the dual operation:
 - ▶ Syntax: `(split &optional (fnum *) depth)`
 - ▶ Splits the current goal into two or more subgoals for each specified formula
 - ▶ Causes branching in the proof tree
 - ▶ It's generally preferable to postpone splitting to reduce proof size

Where to Apply the Rules

Both the logical operator and the location of the formula in the sequent determine the appropriate rule to apply.

Location	Top-level logical connective	
	OR, \Rightarrow	AND, IFF
Antecedent	use (split)	use (flatten)
Consequent	use (flatten)	use (split)

Disjunctive vs. Conjunctive Form

Formulas involving NOT and IF are handled the same way regardless of which part of the sequent they appear:

Location	NOT	IF... THEN... ELSE
Any	use (flatten)	use (split)

Prover normally flattens negated formulas automatically.

PVS Theory for Examples

We will be using a simple PVS theory to illustrate basic prover commands:

```
%%%      Examples and exercises for basic prover commands

prover_basic: THEORY
BEGIN

p,q,r: bool                                % Propositional constants

      :

prop_0: LEMMA  ((p => q) AND p) => q

prop_1: LEMMA  NOT (p OR q) IFF (NOT p AND NOT q)

prop_2: LEMMA      ((p => q) => (p AND q))
      IFF ((NOT p => q) AND (q => p))

      :

fools_lemma: FORMULA  ((p OR q) AND r) => (p AND (q AND r))
```

Completing a Simple Proof

prop_0 :

|-----
{1} ((p => q) AND p) => q

Rule? (flatten)

Applying disjunctive simplification to flatten sequent,
this simplifies to:

prop_0 :

{-1} (p => q)
{-2} p
|-----
{1} q

Note that there is still only one goal.

- ▶ Proof tree is still linear
- ▶ (undo n) will undo n steps

Completing a Simple Proof (Cont'd)

Now we cause the proof tree to branch:

Rule? (split)
Splitting conjunctions,
this yields 2 subgoals:
prop_0.1 :

$$\begin{array}{l} \{-1\} \quad q \\ [-2] \quad p \\ | \text{-----} \\ [1] \quad q \end{array}$$

which is trivially true.

This completes the proof of prop_0.1.

Proof branched, another goal remains.

- ▶ Prover automatically moves to the next remaining goal.
- ▶ (undo n) will undo n steps along path to root.

Completing a Simple Proof (Cont'd)

prop_0.2 :

```
[-1]    p
  |-----
{1}     p
[2]     q
```

which is trivially true.

This completes the proof of prop_0.2.

Q.E.D.

Complete proof tree, showing two subgoals after splitting:

```
(" (flatten) (split) (("1" (propax))
                       ("2" (propax))))
```

A Second Proof

Half of associativity of AND:

prop_1 :

```
|-----  
{1} ((p AND q) AND r) => (p AND (q AND r))
```

Rule? (flatten)

Applying disjunctive simplification to flatten sequent,
this simplifies to:

prop_1 :

```
{-1} p  
{-2} q  
{-3} r  
|-----  
{1} (p AND (q AND r))
```

Again, splitting is required.

Second Proof (Cont'd)

Rule? (split)

Splitting conjunctions,
this yields 3 subgoals:

prop_1.1 :

[-1] p

[-2] q

[-3] r

|-----

{1} p

which is trivially true.

This completes the proof of prop_1.1.

prop_1.2 :

[-1] p

[-2] q

[-3] r

|-----

{1} q

which is trivially true.

Second Proof (Cont'd)

This completes the proof of prop_1.2.

prop_1.3 :

```
[-1]    p
[-2]    q
[-3]    r
  |-----
{1}     r
```

which is trivially true.

This completes the proof of prop_1.3.

Q.E.D.

Proof tree:

```
(" (flatten) (split) (("1" (propax))
                        ("2" (propax))
                        ("3" (propax))))
```

A Slightly Longer Proof (De Morgan's Law)

prop_2 :

|-----
{1} NOT (p OR q) IFF (NOT p AND NOT q)

Rule? (flatten)

No change on: (FLATTEN)

prop_2 :

|-----
{1} NOT (p OR q) IFF (NOT p AND NOT q)

A Slightly Longer Proof (De Morgan's Law)

Now, `flatten` doesn't work here, need `split`!

Rule? (`split`)

Splitting conjunctions,
this yields 2 subgoals:

`prop_2.1 :`

 |-----
{1} NOT (p OR q) IMPLIES (NOT p AND NOT q)

Longer Proof (Cont'd)

Rule? (flatten)

Applying disjunctive simplification to flatten sequent,
this simplifies to:

prop_2.1 :

```
|-----  
{1}    p  
{2}    q  
{3}    (NOT p AND NOT q)
```

Rule? (split)

Splitting conjunctions,
this yields 2 subgoals:

prop_2.1.1 :

```
{-1}    p  
|-----  
[1]    p  
[2]    q
```

which is trivially true.

This completes the proof of prop_2.1.1.

Longer Proof (Cont'd)

prop_2.1.2 :

```
{-1}    q
 |-----
[1]     p
[2]     q
```

which is trivially true.

This completes ... prop_2.1.2, ... prop_2.1.

Longer Proof (Cont'd)

prop_2.2 :

|-----
{1} (NOT p AND NOT q) IMPLIES NOT (p OR q)

Rule? (flatten)

Applying disjunctive simplification to flatten sequent,
this simplifies to:

prop_2.2 :

{-1} (p OR q)
|-----
{1} p
{2} q

Longer Proof (Cont'd)

Rule? (split)

Splitting conjunctions,
this yields 2 subgoals:

prop_2.2.1 :

$$\begin{array}{l} \{-1\} \quad p \\ |----- \\ [1] \quad p \\ [2] \quad q \end{array}$$

which is trivially true.

This completes the proof of prop_2.2.1.

prop_2.2.2 :

$$\begin{array}{l} \{-1\} \quad q \\ |----- \\ [1] \quad p \\ [2] \quad q \end{array}$$

which is trivially true.

This completes ... prop_2.2.2, ... prop_2.2.

Q.E.D.

What Happens When the Formula is not a Theorem?

fools_lemma :

|-----
{1} ((p OR q) AND r) => (p AND (q AND r))

Rule? (flatten)

Applying disjunctive simplification to flatten sequent,
this simplifies to:

fools_lemma :

{-1} (p OR q)
{-2} r
|-----
{1} (p AND (q AND r))

It's starting to look suspicious.

Impossible Proof (Cont'd)

Rule? (split)

Splitting conjunctions,
this yields 2 subgoals:
fools_lemma.1 :

```
{-1}    p
[-2]    r
  |-----
[1]     (p AND (q AND r))
```

Rule? (postpone)

Postponing fools_lemma.1.

fools_lemma.2 :

```
{-1}    q
[-2]    r
  |-----
[1]     (p AND (q AND r))
```

Impossible Proof (Cont'd)

Rule? (split)
Splitting conjunctions,
this yields 3 subgoals:

fools_lemma.2.1 :

```
[-1]    q
[-2]    r
  |-----
{1}    p
```

Rule? (quit)
Do you really want to quit? (Y or N): y

No hope. Give it up!

- ▶ Prover won't give up — you decide

Propositional Simplification

A “black-box” rule for propositional simplification:

- ▶ Syntax: $(prop)$
- ▶ Invokes several lower level propositional rules to carry out a proof without showing intermediate steps
- ▶ Can generally complete a proof if only propositional reasoning is required

Equality Conversion

A rule to convert boolean equalities to IFF:

- ▶ Syntax: `(iff &rest fnums)`
- ▶ Converts equalities on boolean terms so that propositional reasoning can be applied to the two sides
- ▶ Example: convert $(a < b) = (c < d)$ to $(a < b) \text{ IFF } (c < d)$

Replacing Equalities

Antecedent equalities can be used for replacement/rewriting:

- ▶ Syntax: `(replace fnum &optional (fnums *) ...)`
- ▶ Replaces term on LHS with RHS in target formulas
- ▶ Example: if formula -2 is $x = 3 * \text{PI} / 2$

`(replace -2)`

Causes replacement for x throughout the entire sequent

User-Directed Splitting

A rule to force splitting based on user-supplied cases:

- ▶ Syntax: `(case &rest formulas)`
- ▶ Given n formulas A_1, \dots, A_n `case` will split the current goal into $n + 1$ cases.
- ▶ Allows user-directed paths through the proof to be taken so branching can occur on conditions not apparent from the sequent itself
- ▶ Example: `(case "n < 0" "n = 0")` causes three cases to be examined corresponding to whether n is negative, zero, or positive (not negative and not zero).

Embedded IF-expressions

Embedded IF-expressions must be “lifted” to the top (outermost operator) to enable splitting.

▶ Command to lift IF-expressions:

- ▶ Syntax: `(lift-if &optional fnums (updates? t))`.
- ▶ When several IFs are in the sequent, may need to be selective about which to choose.
- ▶ After lifting, `split` may be used.

Effect of `lift-if`:

```
. . . f(IF a THEN b ELSE c ENDIF) . . .
```

becomes:

```
. . . IF a THEN f(b) ELSE f(c) ENDIF . . .
```

Repeated applications bring the IF to the top

Lemma Rules

The prover can be directed to import lemmas and other formulas. Lemmas can come from the containing theory, other user theories, PVS libraries, or the PVS prelude.

- ▶ Syntax: `(lemma name &optional subst)`
- ▶ Example: `(lemma "div_cancel2")`
- ▶ Introduces a new antecedent.
- ▶ Free variables are bound by `FORALL`.
- ▶ Also: `use` and `forward-chain`

Lemma Rules

The prover can be directed to import lemmas and other formulas. Rewriting is a specialized way to use external formulas.

- ▶ Can (conditionally) rewrite terms in the sequent with equivalent terms.
- ▶ Commands: `(rewrite name &optional (fnums *) ...)`,
`(rewrite-lemma lemma subst &optional (fnums *) ...)`, and others

Function applications can be expanded in place (a form of rewriting).

- ▶ Syntax: `(expand name &optional (fnum *) ...)`

Graphical Proof Display

- ▶ Current proof tree may be displayed during a proof.
 - ▶ Command: `M-x x-show-current-proof`
 - ▶ Tree is updated on each command
 - ▶ Clicking on a node shows its sequent.
 - ▶ Helpful for navigating during multiway or multilevel splits.
- ▶ Finished proof may also be displayed.
 - ▶ Command: `M-x x-show-proof`
 - ▶ Invoked outside of prover
- ▶ PostScript can be generated.

Summary

- ▶ Prover commands are S-expressions.
- ▶ Help is on the way:
`help` and `M-x x-prover-commands`
- ▶ Do-over! `undo`
- ▶ No longer just professional wrestling moves:
`split` and `flatten`
- ▶ Other propositional commands covered:
`prop`, `iff`, `replace`, `case`, `lift-if`, etc.
- ▶ A little help from my friends:
`lemma`
- ▶ A picture is worth a thousand proof commands:
`M-x x-show-current-proof`, and `M-x x-show-proof`