Basic Commands & Propositional Logic

Ben Di Vito
with prior contributions from Lee Pike

NASA Langley Formal Methods Group
b.divito@nasa.gov

9–12 October 2012

Sequents

PVS uses *sequents* to represent proof goals. Each contains one or more (sub)formulas.

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

```
{-1}    (p => q)   ⟵ antecedent
{-2}    p          ⟵ antecedent
  |-------         ⟵ turnstile
{1}     q          ⟵ consequent
{2}     r          ⟵ consequent
```

Thus, `p => q` and `p` entail either `q` or `r`.

When entering the prover, the initial sequent will contain a single consequent and no antecedents.

Terminal Sequents

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

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

We will return to the question of why these are "obviously valid."

From Sequents To Proofs

The proof process generates a sequence or tree of sequents.

- Non-branching case: $S_0, S_1, \ldots, S_n$
- Each proof rule ensures that backward implication holds: $S_{i+1} \Rightarrow S_i$
- Since implication is transitive, it follows that $S_n \Rightarrow S_0$.
- If S_n is valid, then so is S_0, i.e., S_0 has been proved.
- When there are branching proof steps, this generalizes to trees of sequents.
  - Every non-branching path in the tree has the backward implication property.
  - The branching steps maintain the property conjunctively: $S_{i+1,1} \wedge \ldots \wedge S_{i+1,k} \Rightarrow S_i$
  - If every leaf L is a valid formula, then so is S_0.

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.

Formulas can be referred to by number:

- Positive numbers for consequents.
- Negative numbers for antecedents.
- Sometimes a list of numbers can be used: `(-2 -1 3 4)`
- Special symbols: `+` (all consequents), `-` (all antecedents), $*$ (all formulas)

Prover Command Documentation

Documentation for each proof command describes its syntax.

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

Optional arguments are specified using two forms:

- (<arg> <dflt>) : default value is <dflt>
- <arg> : default value is nil

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)`, which can be abbreviated `q`
- 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.
  - Limited redo capability: `(undo undo)` undoes last `undo`.
  - Caution: `undo` prunes the proof tree (deletes parallel branches).

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).
  - Performs disjunctive simplification repeatedly.
- 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 helps to carry out steps common to all branches before splitting.

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, =>`	`AND, IFF`
Antecedent	use `(split)`	use `(flatten)`
Consequent	use `(flatten)`	use `(split)`

Recall logical equivalences:

- `P => Q` is equivalent to `(NOT P) OR Q`
- `P IFF Q` is equivalent to `(P => Q) AND (Q => P)`

PVS Theory for Examples

A simple PVS theory to illustrate 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))

END prover_basic
```

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)` here would retract the flatten command.

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 :

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

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))))
```

The Logical Basis of Sequents

A sequent represents a logical formula in disjunctive form.

$$A_1, \ldots, A_m \;\vdash\; C_1, \ldots, C_n$$

$$(A_1 \wedge \ldots \wedge A_m) \Rightarrow (C_1 \vee \ldots \vee C_n)$$

$$\neg(A_1 \wedge \ldots \wedge A_m) \vee (C_1 \vee \ldots \vee C_n)$$

$$\neg A_1 \vee \ldots \vee \neg A_m \vee C_1 \vee \ldots \vee C_n$$

Terminal sequents are special cases that make the disjunction trivially true.

- $C_i = \mathit{True}$
- $A_i = \mathit{False}$
- $A_i = C_j$ (an instance of $P \vee \neg P$)

Basis of Sequents (Cont'd)

Negations are automatically flattened (eliminated) by moving negated formulas to the other side of the turnstile.

- If $C_i = \neg P$, drop C_i and add antecedent P.
- If $A_i = \neg P$, drop A_i and add consequent P.

Contradictory antecedents and contradictory consequents are recognized.

- If P and $\neg P$ both appear as antecedents or as consequents, the $\neg P$ formula will migrate to the other side as P.
- Then a terminal sequent results.
- No need for explicit proof by contradiction.

Disjunctive Simplification

$$A_1, \ldots, A_m, P \wedge Q \ \vdash \ C_1, \ldots, C_n, R \vee S$$

$$\neg A_1 \vee \ldots \vee \neg A_m \vee \neg(P \wedge Q) \vee C_1 \vee \ldots \vee C_n \vee R \vee S$$

$$\neg A_1 \vee \ldots \vee \neg A_m \vee \neg P \vee \neg Q \vee C_1 \vee \ldots \vee C_n \vee R \vee S$$

Sequents can be flattened when formulas are disjunctive.

- If C_i is $P \vee Q$, drop C_i and add consequents P and Q.
- If A_i is $P \wedge Q$, drop A_i and add antecedents P and Q.

There are other disjunctive cases.

- If C_i is $P \Rightarrow Q$, drop C_i and add antecedent P and consequent Q.
- If A_i is $P \Leftrightarrow Q$, drop A_i and add antecedents $P \Rightarrow Q$ and $Q \Rightarrow P$.

Conjunctive Splitting

$$A_1, \ldots, A_m, P \lor Q \;\vdash\; C_1, \ldots, C_n, R \land S$$

$$\neg A_1 \lor \ldots \lor \neg A_m \lor \neg(P \lor Q) \lor C_1 \lor \ldots \lor C_n \lor (R \land S)$$

$$\neg A_1 \lor \ldots \lor \neg A_m \lor (\neg P \land \neg Q) \lor C_1 \lor \ldots \lor C_n \lor (R \land S)$$

When formulas are conjunctive, sequents can be split into two or more cases.

- Follows from the equivalence of $P \lor (Q \land R)$ and $(P \lor Q) \land (P \lor R)$.
- If C_i is $P \land Q$, create two sequents, replacing C_i by P then Q.
- If A_i is $P \lor Q$, create two sequents, replacing A_i by P then Q.

Conjunctive Splitting (Cont'd)

$$A_1, \ldots, A_m, P \Rightarrow Q \;\vdash\; C_1, \ldots, C_n, R \Leftrightarrow S$$

$$\neg A_1 \vee \ldots \vee \neg A_m \vee \neg(\neg P \vee Q) \vee C_1 \vee \ldots \vee C_n \vee ((R \Rightarrow S) \wedge (S \Rightarrow R))$$

$$\neg A_1 \vee \ldots \vee \neg A_m \vee (P \wedge \neg Q) \vee C_1 \vee \ldots \vee C_n \vee ((R \Rightarrow S) \wedge (S \Rightarrow R))$$

The other conjunctive cases are similar.

- If C_i is $P \Leftrightarrow Q$, create two sequents with C_i replaced by $P \Rightarrow Q$ then $Q \Rightarrow P$.
- If A_i is $P \Rightarrow Q$, create two sequents as follows.
  - Create a sequent with A_i replaced by Q.
  - Create a sequent by dropping A_i and adding consequent P.

Splitting can also be used for top-level IF-expressions.

Implication Handling

During flattening and splitting, the two sides of an implication go to opposide sides of the turnstile.

```
(flatten)                    New sequent

                             {-1}   p
{-1}   r                     [-2]   r
  |-------                     |-------
{1}    p => q                [1]    q


(split -1)                    Branch 1                Branch 2

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

Due to negation elimination, the contrapositive ($\neg Q \Rightarrow \neg P$) of the implication ($P \Rightarrow Q$) will give the same results.

Example: A Longer Proof (De Morgan's Law)

```
prop_2 :

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

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

prop_2.1 :

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

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:

prop_2.1 :
  |-------
{1}    p
{2}    q
{3}    (NOT p AND NOT q)
```

Longer Proof (Cont'd)

```
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.

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 :

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

which is trivially true.  This completes the proof of prop_2.2.1.

prop_2.2.2 :

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

which is trivially true.  This completes ... prop_2.2.2, ... prop_2.2.

Q.E.D.
```

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.

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) IFF (c < d)`

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 (Cont'd)

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 *) ...)`
- Also works for constants.

Example: Propositional Weather Model

```
landing_weather: THEORY
BEGIN

clear:  bool    % Minimal cloudiness
cloudy: bool    % Mostly cloudy skies
rainy:  bool    % Steady rainfall
snowy:  bool    % Includes sleet, freezing rain, etc.
windy:  bool    % Moderate wind speed

cond_ax1: AXIOM  rainy => cloudy
cond_ax2: AXIOM  snowy => cloudy
cond_ax3: AXIOM  clear IFF NOT cloudy

ideal:     bool = clear AND NOT windy
favorable: bool = NOT rainy AND NOT snowy
adverse:   bool = rainy OR snowy

weath_1: LEMMA  rainy => NOT clear
weath_2: LEMMA  snowy => NOT clear
weath_3: LEMMA  clear => favorable
           ⋮
END landing_weather
```

A Proof About the Weather

```
weath_1 :

  |-------
{1}   rainy => NOT clear

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
weath_1 :

{-1}  rainy
{-2}  clear
  |-------

Rule? (lemma "cond_ax1")
Applying cond_ax1
this simplifies to:
weath_1 :

{-1}  rainy => cloudy
[-2]  rainy
[-3]  clear
  |-------
```

A Weather Proof (Cont'd)

```
Rule? (split)
Splitting conjunctions,
this yields  2 subgoals:
weath_1.1 :

{-1}  cloudy
[-2]  rainy
[-3]  clear
  |-------

Rule? (lemma "cond_ax3")
Applying cond_ax3
this simplifies to:
weath_1.1 :

{-1}  clear IFF NOT cloudy
[-2]  cloudy
[-3]  rainy
[-4]  clear
  |-------
```

A Weather Proof (Cont'd)

```
Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
weath_1.1 :

{-1}  clear IMPLIES NOT cloudy
{-2}  NOT cloudy IMPLIES clear
[-3]  cloudy
[-4]  rainy
[-5]  clear
  |-------

Rule? (split -1)
Splitting conjunctions, this yields  2 subgoals:
weath_1.1.1 :

[-1]  NOT cloudy IMPLIES clear
[-2]  cloudy
[-3]  rainy
[-4]  clear
  |-------
{1}   cloudy

which is trivially true.  This completes the proof of weath_1.1.1.
```

A Weather Proof (Cont'd)

```
weath_1.1.2 :

[-1]  NOT cloudy IMPLIES clear
[-2]  cloudy
[-3]  rainy
[-4]  clear
  |-------
{1}   clear

which is trivially true.
This completes ... weath_1.1.2. This completes ... weath_1.1.

weath_1.2 :

[-1]  rainy
[-2]  clear
  |-------
{1}   rainy

which is trivially true.
This completes the proof of weath_1.2.

Q.E.D.
```

A Second Weather Proof

```
weath_3 :

  |-------
{1}   clear => favorable

Rule? (expand "favorable")
Expanding the definition of favorable,
this simplifies to:
weath_3 :

  |-------
{1}   clear => NOT rainy AND NOT snowy

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
weath_3 :

{-1}  clear
  |-------
{1}   NOT rainy AND NOT snowy
```

A Second Weather Proof (Cont'd)

```
Rule? (split)
Splitting conjunctions,
this yields  2 subgoals:
weath_3.1 :

{-1}  rainy
[-2]  clear
  |-------

Rule? (lemma "weath_1")
Applying weath_1
this simplifies to:
weath_3.1 :

{-1}  rainy => NOT clear
[-2]  rainy
[-3]  clear
  |-------
```

A Second Weather Proof (Cont'd)

```
Rule? (split)
Splitting conjunctions, this yields  2 subgoals:
weath_3.1.1 :

[-1]  rainy
[-2]  clear
  |-------
{1}   clear

which is trivially true.  This completes the proof of weath_3.1.1.

weath_3.1.2 :

[-1]  rainy
[-2]  clear
  |-------
{1}   rainy

which is trivially true.  This completes the proof of weath_3.1.2.
This completes the proof of weath_3.1.
```

A Second Weather Proof (Cont'd)

```
weath_3.2 :

{-1}  snowy
[-2]  clear
  |-------

Rule? (lemma "weath_2")
Applying weath_2
this simplifies to:
weath_3.2 :

{-1}  snowy => NOT clear
[-2]  snowy
[-3]  clear
  |-------
```

A Second Weather Proof (Cont'd)

```
Rule? (split)
Splitting conjunctions, this yields  2 subgoals:
weath_3.2.1 :

[-1]  snowy
[-2]  clear
  |-------
{1}   clear

which is trivially true.  This completes the proof of weath_3.2.1.

weath_3.2.2 :

[-1]  snowy
[-2]  clear
  |-------
{1}   snowy

which is trivially true.  This completes ... weath_3.2.2.
This completes ... weath_3.2.

Q.E.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 * 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 $A_1, \ldots, A_n$ `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

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`
- Core propositional reasoning commands:

  `split` and `flatten`
- Other propositional commands covered:

  `prop`, `iff`, `replace`, `case`, `lift-if`, etc.
- A little help from my friends:

  `lemma`, `expand`
- A picture is worth a thousand proof commands:

  `M-x x-show-current-proof`, and `M-x x-show-proof`