Logical Proving in PVS

Aaron Dutle

NASA Langley Research Center

aaron.m.dutle@nasa.gov

Outline

```
p,q,r : bool

prove | status-proofchain | show-prooflite
ex1: LEMMA
((p => q) and p ) => (q or r)
```

Basics of the prover

Propositional logic

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

Predicate logic

```
prove | status-proofchain | show-prooflite
pred_ex1: LEMMA
FORALL (s,t,u: bool):
(s AND t) OR u <=> (s OR u) AND (t OR u)
```

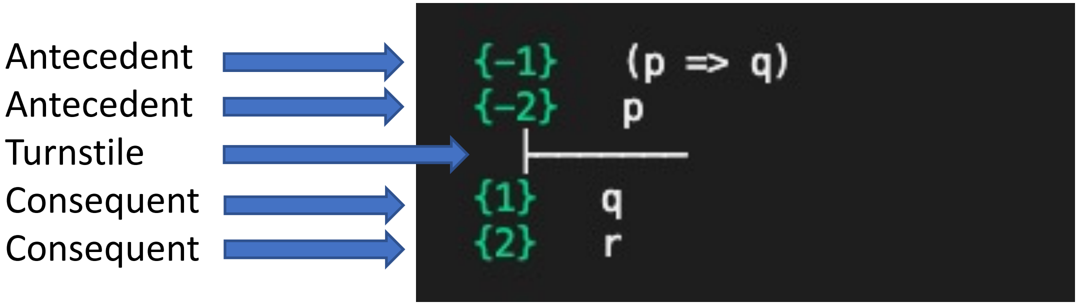
PVS prover structure

PVS uses *sequents* to represent proof goals. A sequent is composed of (numbered) *formulas*.

Read a sequent as “the conjunction (*and*) of the antecedents implies the disjunction (or) of the consequents”

The goal in the prover is to manipulate sequents using (logically sound) commands into something that is *obviously true* to PVS.

* FALSE in the antecedent
* TRUE in the consequent
* Same formula in antecedent and consequent



“p => q and p implies either q or r”

Trees of sequents

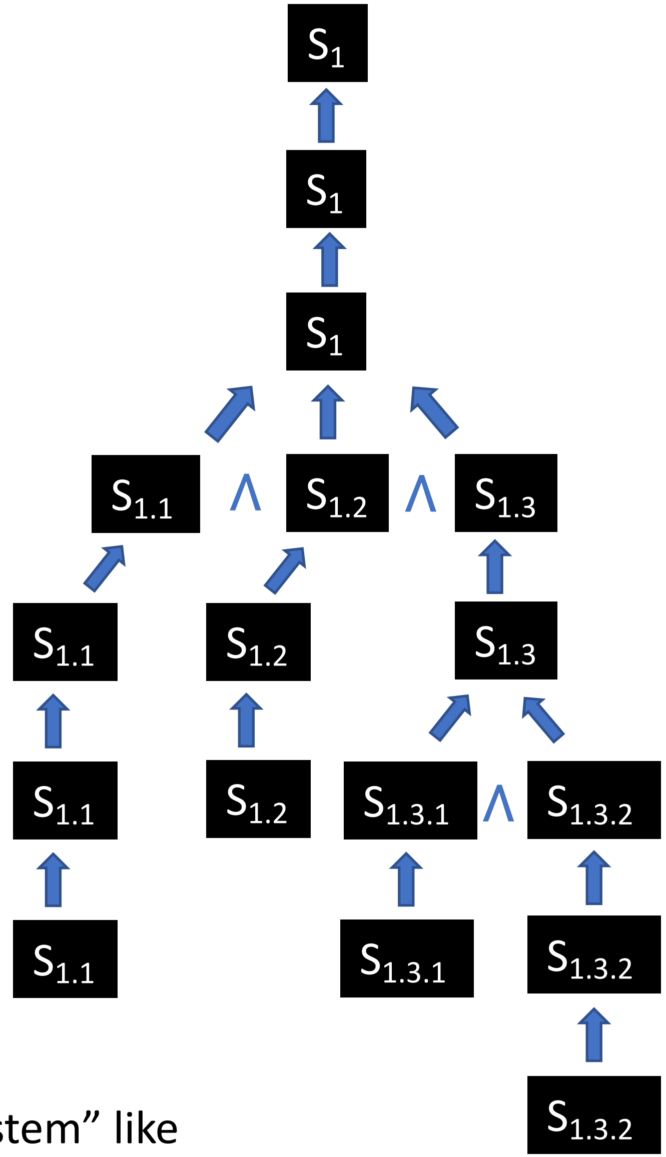
The proof process generates sequences or (usually) trees of sequents.

Non-branching case:

- Generates a sequence S_0, S_1,…, S_n
- Proof rules ensure that S_{i+1} => S_i
- Implication is transitive, so S_n => S_0

Branching case:

- Splits a sequent S_i into $S_{i+1,1}$, $S_{i+1,2}$,…, $S_{i+1,k}$
- The branches conjunctively prove the previous step, i.e. $S_{i+1,1}$, $S_{i+1,2}$,…, $S_{i+1,k}$ => S_i
- If each leaf is valid, then the original sequent is also



Notes: PVS only adds numbering to branching steps, as on the right. A "file-system" like tree can be viewed in the proof-explorer, or a more graphical version is shown using the button  in the menu bar.

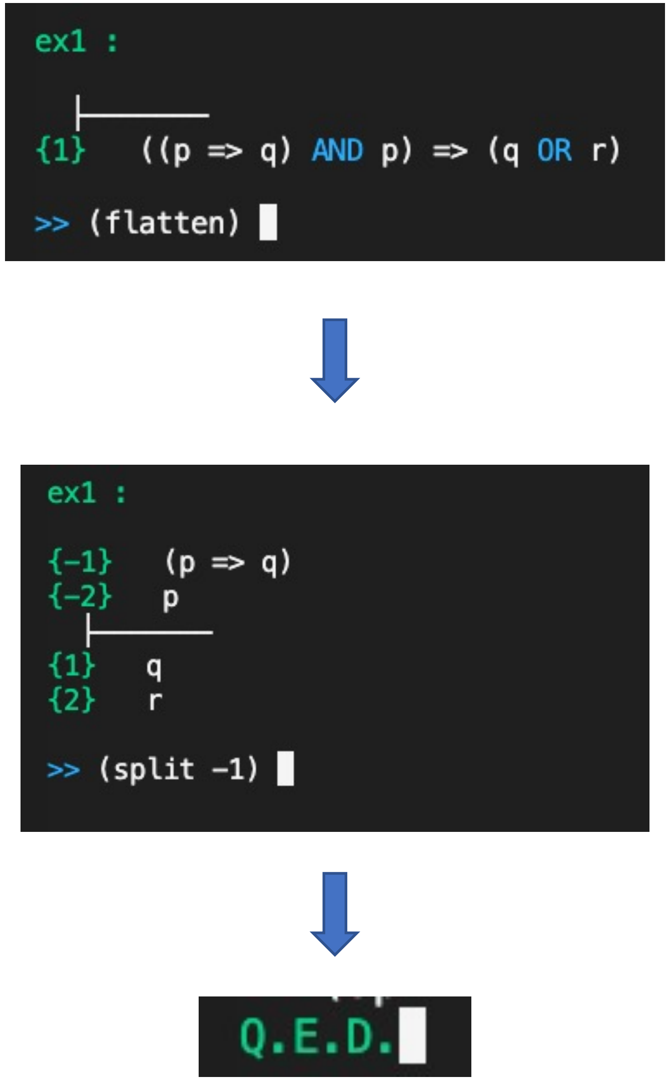
Manipulating Sequents: Basics

Proof commands are entered as Lisp S-expressions:

- Examples: `(flatten)`, `(split -1)`, `(expand “factorial”)`
- Commands are *proof rules*, *control rules*, or *strategies*.
- Arguments to the rules are generally numbers or strings
- Parentheses can be omitted for single line commands

Formulas are referred to by number (or label, coming soon):

- Positive numbers in the consequent
- Negative numbers in the antecedent
- Sometimes multiple formulas: (-2 -1 3 4)
- Special ones: + (entire consequent), - (entire antecedent), * (all formulas)



Manipulating Sequents: Help

```
>> help split

(split &optional (fnum *) depth):
   Conjunctively splits formula FNUM.  If FNUM is -, + or *, then
the first conjunctive sequent formula is chosen from the antecedent,
succedent, or the entire sequent.  Splitting eliminates any
top-level conjunction, i.e., positive AND, IFF, or IF-THEN-ELSE, and
negative OR, IMPLIES, or IF-THEN-ELSE.
```

Help with commands:

- Begin typing a command, and VSCode shows abbreviated help below the prover
- From the prompt, type `(help command_name)`
- Provides the syntax of the command, and a description

Command syntax	Some instances
`(copy fnum)`	`(copy 2)` `(copy -3)`
`(skeep &optional (fnum + -) preds?)`	`(skeep)` `(skeep -3)` `(skeep + t)`

`(induct var &optional (fnum 1) name)`	`(induct “n”)`
	`(induct “n” 2)`
	`(induct “n” :name “NAT_induction”)`

`(hide &rest fnums)`	`(hide 2)` `(hide -)`
	`(hide -3 -4 1 2)`
	`(hide -2 +)`

Reading the syntax:

- Shows the command, required, and optional inputs
- Optional arguments have the forms `(<arg> <dflt>)` or just `<arg>` with `nil` as default

Manipulating Sequents: Navigating

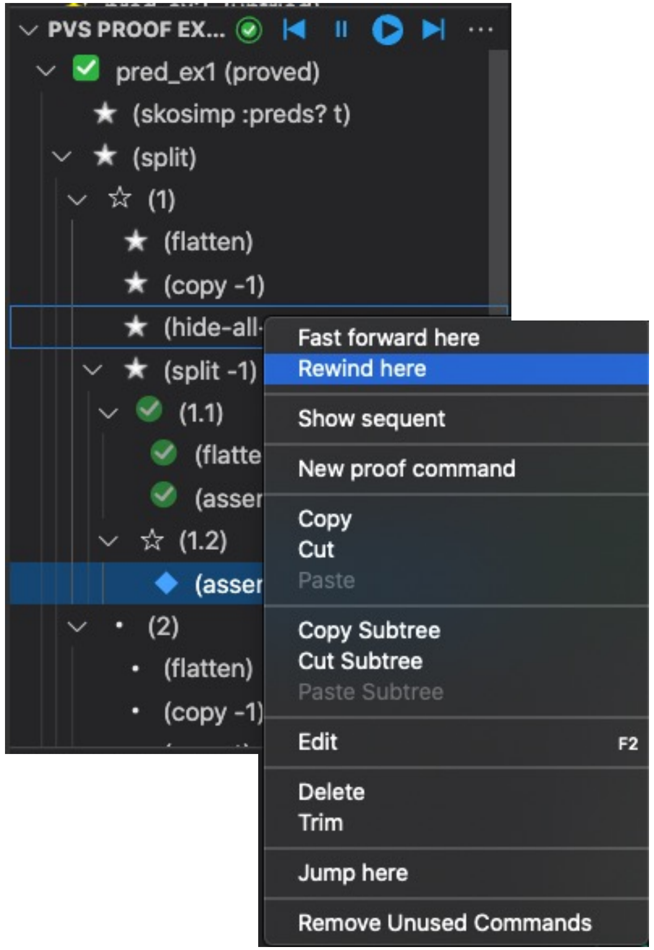
There are commands to control the place in the proof.

- Exiting the prover: `(quit)` brings a Save Proof prompt. Note: Yes saves and quits, No discards and quits, Cancel returns to the proof

- Switching Branches: `(postpone)` moves to the next open branch

- Undo/Redo: In Proof Explorer, right-click to fast-forward or rewind steps. Alternative: `(undo)` move you backward through proof steps, `(undo n)` moves back n steps `(undo undo)` cancels ONE undo step.

- Whether using `(undo)` or rewind, undoing a branch step undoes ALL of the siblings to the head (but Proof Explorer can replay them)



Navigate a proof with the buttons at top, or right-click to get to rewind or fast-forward to a chosen step.

Manipulating Sequents: Two Propositional Rules

Sequent flattening:

- Syntax: `(flatten &rest fnums)`
- Usually applied to the whole sequent, although formula numbers can be specified

Sequent splitting:

- Syntax: `(split &optional (fnum *) depth)`
- Splits the goal into two (or more) subgoals
- These goals become branches in the proof tree
- Note: complete steps common to all branches prior to splitting
- Related Commands: `(case “branch”)` `(splash)`

What should I use?

Location	Logical Connective	
	OR, IMPLIES	AND, IFF
Antecedent	`(split)`	`(flatten)`
Consequent	`(flatten)`	`(split)`

Remember: a sequent is the AND of the antecedents implies the OR of the consequent

- If the connective matches the side, use flatten
- If the connective opposes the side, use split

From logic class:

- P => Q is also (NOT P) OR Q
- P <=> Q is also (P => Q) AND (Q => P)

A Short Proof

From this basic theory, prove prop_0 with just split and flatten

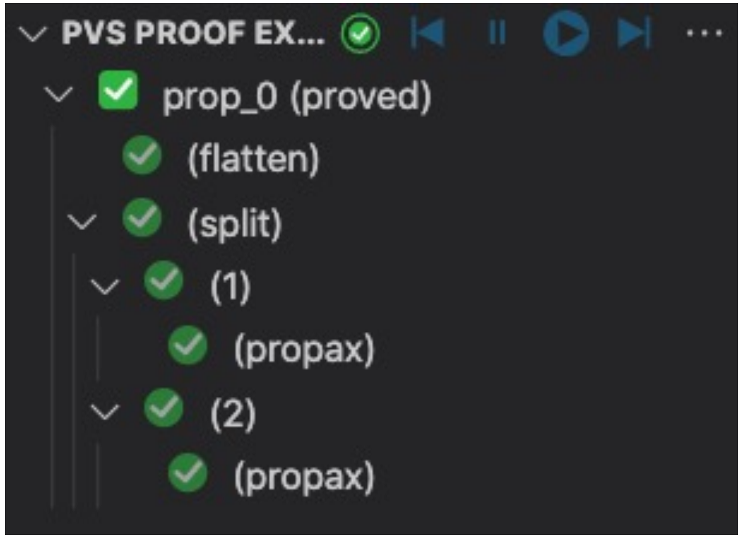
```
prop_basic: THEORY
BEGIN

p,q,r: bool          % Propositional constants

prove | status-proofchain | show-prooflite
prop_0: LEMMA  ((p => q) AND p) => q

prove | status-proofchain | show-prooflite
prop_1: LEMMA  ((p AND q) AND r) => (p AND (q AND r))

prove | status-proofchain | show-prooflite
prop_2: LEMMA  NOT (p OR q) IFF (NOT p AND NOT q)

%
%
%

prove | status-proofchain | show-prooflite
fools_lemma: FORMULA  ((p OR q) AND r) => (p AND (q AND r))


END prop_basic
```

A Short Proof

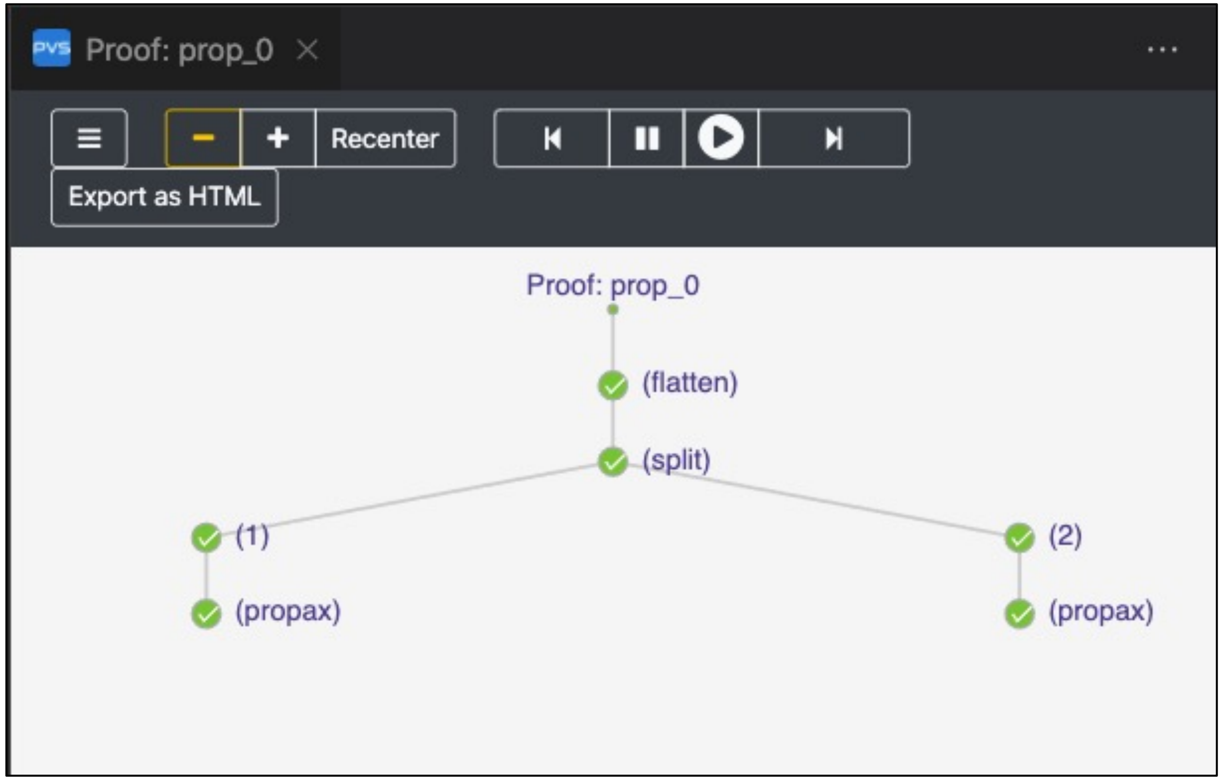
```
Starting prover session for prop_0


prop_0 :

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

>> flatten

Applying disjunctive simplification to flatten sequent

prop_0 :

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

>> split
Q.E.D.
```

- IMPLIES (=>) is the outermost connective, and in the consequent
- `(flatten)` transforms the original sequent to the second
- `(split)` then creates 2 (obviously true) branches to finish the proof

Two views of “A Short Proof”



The completed proof in “Proof Explorer”



The completed proof from “Show Proof Tree”

Other important commands

`(prop)`
- "Black-box" rule for propositional logic
- Will complete most propositional-only proofs in one step

`(iff &rest fnums)`
- Example Syntax: `(iff 2)`
- Converts equalities on Booleans to IFF so that propositional reasoning applies
- Example: (a<b) = (c=>d) becomes (a<b) IFF (c=>d)

`(expand name &optional (fnum *))`
- Example Syntax: `(expand "factorial" 1)`
- Rewrites a defined function or constant using the definition

`(lemma name &optional subst)`
- Example Syntax: `(lemma "floor_plus_int")`
- Adds an antecedent with the lemma
- Free variables bound with FORALL
- Related Commands: `use` and `forward-chain`

`(rewrite name &opt (fnums *) (target-fnums *) (dir lr))`

Where to get the inputs to the lemma

Where to apply the actual rewrite

Which direction to rewrite in

- Example Syntax:

`(rewrite "floor_plus_int" -2 :target-fnums 3 :dir rl)`

- Matches constants from formula -2
- Puts them in "floor_plus_int"
- Rewrites things in formula 3
- Using the equality reading left-to-right

Three more commands

`(replace fnum &optional (fnums *) (dir lr) …)`

- Example Syntax: `(replace -1 3)`
- Replaces using an equality formula inside target formulas, with the direction specified

`(case &rest formulas)`

- Example Syntax: `(case “n<0”)`
- Separates the proof into two cases: “formula” is true in the first, and ”formula” is false in the second.
- Allows for the user to decide where a split should occur.
- Multiple formulas be input for more branching

`(lift-if &optional fnums)`

- Example Syntax: `(lift-if -2)`
- IF – THEN – ELSE expressions must be on the outermost part of a formula to use `(split)`
- This command lifts such expressions one level
- Example:
  `… f(IF a THEN b ELSE c ENDIF) …`
  becomes
  `… IF a THEN f(b) ELSE f(c) ENDIF …`
- Alternative: Use `(case “a”)`

Put them to work

Try the commands out on some

Exercises!

Quantified Formulas

Formulas are often declared that use quantifiers over free variables

- Examples:

```
pred_ex1: LEMMA
FORALL (s,t,u: bool):
(s AND t) OR u <=> (s OR u) AND (t OR u)
```

```
x,y,z: VAR real
prove | status-proofchain | show-prooflite
pred_ex3: LEMMA EXISTS z: x+z = 0
```

- Note that free (previously declared) variables in formulas are treated as universally quantified, so

```
pred_ex2: LEMMA x*y = y*x
```

↓

```
  |-------
{1}   FORALL (x, y: real): x * y = y * x
```

inside the prover

- **Skolemization** and **Instantiation** are used to eliminate quantifiers

Skolemization

Suppose you have a property **P**, and you want to show **all** real numbers possess it.

- In the PVS prover, this looks like

```
  |-------
{1}   FORALL (x: real): P(x)
```

- In math, a proof would start with “Let x be an arbitrary real number…”
- In the PVS logic, this is called **Skolemization**

```
sko_1 :

  |-------
{1}   FORALL (x: real): P(x)

>> (skolem 1 "x")

For the top quantifier in 1, we introduce Skolem constants: x

sko_1 :

  |-------
{1}   P(x)
```

Skolemization

Similarly, suppose you have a property **Q**, and you know **some** real number possesses it.

- In the PVS prover, you would see

```
[-1]    EXISTS (x: real): Q(x)
  |-------
```

- In math, a proof would start with “Let x be an arbitrary real number with property **Q**…”
- This is still **Skolemization!!!**

```
sko_2 :

[-1]    EXISTS (x: real): Q(x)
  |-------

>> (skolem -1 "x")

For the top quantifier in -1, we introduce Skolem constants: x

sko_2 :

{-1}    Q(x)
  |-------
```

Skolemization

Skolemize:
- Universal quantifiers in the consequent
- Existential quantifiers in the antecedent
- For example: both formulas here

```
{-1}   EXISTS (x: real): Q(x)
  |-------
[1]   FORALL (x: real): P(x)
```

Skolemization introduces a fresh (not previously used in the proof) constant, called a **skolem constant**, representing a fixed but arbitrary representative.

Thoralf Skolem (1887-1963), Norwegian mathematician who worked in mathematical logic and set theory.

Skolem image from http://www.oslobilder.no/OMU/OB.F06426c, in public domain.

Instantiation

Instantiation is the dual process to skolemization

Suppose you have a property **P**, and you know that **all** real numbers possess it.

- In the PVS prover, this looks like

```
{-1}   FORALL (x: real): P(x)
  |-------
```

- Since it's true for all real numbers, you can choose your favorite one
- This is **Instantiation**

```
sko_2.1 :

{-1}   FORALL (x: real): P(x)
  |-------

>> (inst -1 "6.022 * 10^23")

Instantiating the top quantifier in -1 with the terms:
 6.022 * 10^23

sko_2.1 :

{-1}   P(6.022 * 10 ^ 23)
  |-------
```

Instantiation

Similarly, suppose you have a to prove the existence of a real number with property **Q,** and somehow, you've discovered one.

- In the PVS prover, this looks like

```
{-1}   Q(3.14159)
  |-------
[1]    EXISTS (x: real): Q(x)
```

- To finish this proof, you simply need to supply the witness to formula 1.
- Again, **Instantiation** does the trick.

```
sko_3 :

{-1}   Q(3.14159)
  |-------
{1}    EXISTS (x: real): Q(x)

>> (inst 1 "3.14159")
Q.E.D.
```

Instantiation

Instantiate:

- Existential quantifiers in the consequent
- Universal quantifiers in the antecedent
- For example: both formulas here

```
{-1}   FORALL (x: real): P(x)
  |-------
[1]   EXISTS (x: real): Q(x)
```

Instantiation replaces a quantified variable with some previously declared constant.

```
{-1}   FORALL (x: real): P(x) => Q(2 * x)
[-2]   EXISTS (x: real): P(x)
  |-------
[1]   EXISTS (x: real): Q(x)
```

Note: Instantiation doesn't have to involve numerical or externally declared constants, skolem constants are great.

In the example above, three commands:

```
(skolem -2 "x")
(inst -1 "x")
(inst 1 "2 * x")
```

will complete the proof.

Skolemization and Instantiation Commands

`(skeep)`
- Example Syntax: `(skeep -1)`
- Skolemize and "keep" variable names (when possible)
- Applies (flatten) after skolemizing, usually helpful

`(skolem fnum names)`
- The basic skolemization command
- Uses constants "names" in the quantified formula "fnum"

`(skolem! &opt fnum)`
- Skolemizes a formula, optionally specified
- A variable `x` becomes `x!1` or `x!2`

`(skosimp*)`
- Applies `(skolem!)` then `(flatten)`

Note: When specifying names, use `"_"` to leave a variable uninstantiated (useful when only some values change).

`(inst fnum &rest terms)`
- Example Syntax: `(inst -1 "pi/2")`
- The basic instantiation command

`(inst? &opt fnum)`
- If fnum is given, PVS tries to choose an appropriate instantiation for it
- If no fnum, PVS chooses a formula and an instantiation

`(inst-cp fnum &rest terms)`
- Works like `(inst)`, but keeps a copy of the quantified formula

Note: Be careful when instantiating. PVS will typecheck any instantiations, and may stop instantiation, or produce TCC branches.
- Example: If you have `FORALL (n: nat): P(n)` and instantiate it with "0.5" you'll get an (unprovable) TCC branch asking to prove that 0.5 is a nat.

Commands to make the sequent look good

`(hide &rest fnums)`

- Example Syntax: `(hide -1 -2 +)`
- Removes formulas from the sequent
- Removed formulas are NOT used for deduction, or affected by commands
- Useful if the sequent is complicated
- Alternate: `(hide-all-but &opt (fnums *))`

`(reveal &rest fnums)`

- Example Syntax: `(reveal 2)`
- Brings hidden formulas back to the current sequent
- Need to know the right number (or label)! Get it with `(show-hidden-formulas)`

`(label name fnums)`

- Example Syntax: `(label “ind_hyp” -3)`
- Allows labelling of formulas with strings
- Hide a labeled formula early in a proof, and reveal it at the end when you need it
- Note: `hide` and `reveal` both accept labels!

Commands to make life easier

The prover has a collection of (increasingly aggressive) simplification commands.

`(prop)`
- Repeated `flatten` and `split`

`(bddsimp)`
- Propositional simplification with Binary Decision Diagrams (BDDs)

`(assert)`
- Applies type-specific decision procedures and auto-rewrites

`(ground)`
- Propositional simplification plus decision procedures

`(smash)`
- Repeatedly tries `bddsimp`, `assert`, and `lift-if`

`(grind)`
- All of the above, plus definition expansion and `inst?`

Note: `(grind)` can take a long time, get stuck in a loop, or leave the sequent unfamiliar. Sometimes it needs to be interrupted or undone to get back to normal.

What's your type?

The prover can be asked to reveal information about the TYPE of an expression.

`(typepred &rest exprs)`

- Example Syntax: `(typepred "a")`
- Causes type constraints for expressions to be added to the sequent
- Subtype predicates are often recalled this way
- Alternate: When skolemizing, use the `:preds? T` option at the end of `(skeep)`

```
st :

  |-------
{1}   FORALL (a: {x: real | abs(x) < 1}): a ^ 2 < 1

>> (skeep)

Skolemizing and keeping names of the universal formula in (+ -)

st :

  |-------
{1}   a ^ 2 < 1

>> (typepred "a")

Adding type constraints for  a

st :

{-1}   abs(a) < 1
  |-------
[1]   a ^ 2 < 1
```

An example using `typepred`

Put them to work

Try the commands out on some

Exercises!

Getting more help

- PVS website: https://pvs.csl.sri.com/
- PVS prover guide: https://pvs.csl.sri.com/doc/pvs-prover-guide.pdf (locally at <pvs_folder>/doc/prover/prover.pdf)
- PVS google group: https://groups.google.com/g/pvs-group

Further help

Try the commands out on some

Exercises!