PVS Survival Hints

by

Ricky W. Butler

Mail Stop 130
NASA Langley Research Center
Hampton, Virginia 23681-2199

email: R.W.Butler@nasa.gov
phone: (757) 864-6198
fax: (757) 864-4234
web: http://shemesh.larc.nasa.gov/fm/

October 12, 2012

Stopping PVS Prover and Restore

If you have a run-away prover command (e.g. GRIND), you can type `C-c C-c` (while in the `*pvs*` buffer), which will drop you into Lisp:

```
Error: Received signal number 2 (Keyboard interrupt)
  [condition type: INTERRUPT-SIGNAL]

Restart actions (select using :continue):
 0: continue computation
[1c] <rcl>
```

At the `<rcl>` prompt, type `(restore)` to get back to the previous sequent.

If PVS has lost its mind, type `M-x reset-pvs`. Note. During garbage collection, this command will not engage immediately.

Managing Proofs

- **Corresponding to each .pvs file, there is a .prf file that holds the proofs.**
- **Put cursor on a particular lemma and type** `M-x edit-proof` **to see the stored proof. A buffer entitled** `Proof` **will be displayed. The buffer will contain something like:**

```
("" (SKOSIMP*)
    (EXPAND "add_flight")
    (LIFT-IF)
    (EXPAND "del_flight")
    (PROPAX))
```

- **You can change the proof and save it by issuing** `C-C C-i` **to the** `Proof` **buffer.**

Managing Proofs (cont.)

- **WARNING: If you rename a lemma, the corresponding proof will disappear.**
- **The system will save the proof in a file called** `orphaned-proofs.prf`
- **The command** `M-x show-orphaned-proofs` **will bring up a buffer entitled** `Proofs` **which looks something like**

```
File                Theory              Formula
----                ------              -------
flight_sched3       flight_sched3       u0_TCC1
extra               extra               dumb_but_instructive
```

  If you put the cursor on a particular formula name in the buffer and type `s`**, the proof will be displayed in a buffer named** `Proof`**.**

If you manually edit a .prf file, the PVS system does NOT automatically load it for you. You have to issue a `M-x install-pvs-proof-file` **to get these changes loaded by PVS.**

To display all of the proofs in the current file, use `M-x show-proofs-pvs-file`

To display ALL proofs including those in IMPORTED theories,
use `M-x show-proofs-importchain`

Changing Variable Names

The best way to change variable names in PVS is to

1. **Create a "top.pvs" file that imports directly or indirectly all of the theories you are working on.**
2. **Issue the `M-x dump-pvs-files` command in your `top.pvs` theory.**
3. **Make changes in the dump file and save it.**
4. **Exit PVS.**
5. **Issue Unix command `rm pvsbin/*.bin` to your directory. (This shouldn't be necessary, but there have been bugs with .bin in the past, and I still do this just in case.)**
6. **Restart PVS.**
7. **Issue `M-x undump-pvs-files`.**
8. **Go to `top.pvs` and retypecheck.**

Note. The `M-x dump-pvs-files` command can be used to prepare bug reports to send to SRI.

Note. The `top.pvs` theory is a good place to issue `M-x pri` (or `M-x prove-importchain`) to reprove all of your stuff.

Stepping Through a Proof

You can step through a proof one command at a time, by issuing `M-x step-proof` on a lemma.

The window is split into two parts, one containing the proof (named `Proof`) and the other containing the `*pvs*` proof buffer.

The command `TAB 1` executes one proof step.

The command `ESC n TAB 1` executes `n` proof steps.

Repositioning the cursor in the `Proof` buffer changes which command will execute next.

Therefore, use `C-x o` to switch between windows rather than clicking.

Command Abbreviations

TAB * (skosimp*)

TAB a (assert)

TAB c (case) – prompts for expression

TAB e (expand) – expands identifier at cursor

TAB E (apply-extensionality)

TAB f (flatten)

TAB g (ground)

TAB G (grind)

TAB i (inst) – prompts for expression

TAB ? (inst?)

TAB l (lift-if)

TAB s (split)

TAB p (prop)

TAB r (replace)

TAB u (undo)y

Overcoming Some Common Proof Problems

- **If you find yourself proving the same result over and over:**
  - **introduce a case statement early:** `(case "<same-result>")`
  - **add a lemma before this theorem**
- **If you are seeing a plethora of TCCs generated during your proofs:**
  - **Look to see if there are some JUDGEMENTS you can add to your specification**
  - **Consider adding a new lemma that you can serve as an auto-rewrite. This can be automatically turned on by adding** `AUTO-REWRITE "<lemma>"` **to your theory.**
- **If you lost in the details (evidence: massive sequents, dozens of unproved subgoals).**
  - **Go get a cup of coffee**
  - **Walk around the building and think about the big picture**
  - **Come back and scratch out a new informal proof on paper**

Adding Declarations Without Having To Re-Typecheck

If you would like to add a declaration without causing the system to retypecheck everthing, use `M-x add-declaration`

This is useful when you are in a proof and you would like to add a lemma.

This is also useful when you have a large collection of theories and many of them import some fundamental theories.

You can add some stuff to the fundamental theories without having to re-typecheck everything that uses it.

This command opens up a buffer. You type in the new stuff and then issue `C-c C-c`.

The command `M-x modify-declaration` works similarly.

Proof Information Commands

You are baffled because

```
{-1}   lovely_predicate(x!1 - y!1, x!1 + y!1)
  |-------
{1}    lovely_predicate(x!1 - y!1, x!1 + y!1)
```

is not discharged by (ASSERT).

Issue command `M-x show-expanded-sequent` **and you see**

```
{-1}   thry1[nat].lovely_predicate(x!1 - y!1, x!1 + y!1)
  |-------
{1}    thry2[nat].lovely_predicate(x!1 - y!1, x!1 + y!1)
```

Because of the extensive overloading that is supported in PVS, it is sometimes necessary to see the fully expanded names to help you disambiguate formulas in a sequent.

Hidden Formulas

- The PVS prover command `(hide -1)` removes a formula from a sequent.
- This helps reduce clutter.
- To see all of the hidden formulas, type `M-x show-hidden-formulas`. The formulas will be displayed in a buffer named `Hidden`.
- The prover command `(reveal x)` will return a formula to the sequent. Note. `x` is the number of the formula in the `Hidden` buffer and not its original number.

The new `LABEL` command can be used prior to HIDING to facilitate the subsequent `REVEAL` command.:

```
(LABEL EASY  -3)
(HIDE EASY)
  .
  .
(REVEAL EASY)
```

View Commands

- **The command `M-x show-tccs` dispays the TCCs associated with a theory.**
- **The command `M-x ppe` (or `M-x pretty-print-expanded`) can be used to see all of the TCCs as well as specially generated axioms. For example, `M-x ppe` on**

```
ppefun: THEORY
BEGIN
   letters: TYPE = {a,b,c}
END ppefun
```

will produce

```
ppefun: THEORY
BEGIN
letters: TYPE = {a, b, c}

letters: TYPE
a?, b?, c?: [letters -> boolean]
a: (a?)
b: (b?)
c: (c?)
ord(x: letters): upto(2) = CASES x OF a: 0, b: 1, c: 2 ENDCASES
```

M-x ppe continued

```
letters_inclusive: AXIOM
      (FORALL (letters_var: letters):
         a?(letters_var) OR b?(letters_var) OR c?(letters_var));

letters_induction: AXIOM
      (FORALL (p: [letters -> boolean]):
         p(a) AND p(b) AND p(c)
             IMPLIES (FORALL (letters_var: letters): p(letters_var)));

END ppefun
```

- **the**`<type>_inclusive` **axiom must be used in proofs via a** `LEMMA` **command.**
- **the decision procedures automatically handle the disjointness property.**
- **the** `<type>_induction` **axiom isn't real useful.**

Proof Status Commands

The command `M-x spt` (abbreviation for `M-x status-proof-theory`):

```
Proof summary for theory flight_sched3
   u0_TCC1................................................unfinished
   putative2..............................................proved - incomplete
   SchedAdd...............................................proved - incomplete
   DelAdd.................................................proved - incomplete
   AddChange..............................................proved - incomplete
   Theory totals: 5 formulas, 5 attempted, 4 succeeded.
```

untried	**Proof not yet attempted.**
unfinished	**Proof attempted but not finished.**
proved-incomplete	**The proof is done but depends upon other unproven lemmas.**
proved-complete	**The proof is done and all lemmas it depends upon are also proved.**

Proof Status Commands (cont.)

The command `M-x spc` **(abbreviation for** `M-x status-proofchain`**):**

```
flight_sched3.AddChange has been PROVED.

  The proof chain for AddChange is INCOMPLETE.
  It depends on the following unproved conjectures:
    flight_sched3.u0_TCC1

  AddChange depends on the following proved theorems:
    if_def.IF_TCC1

  AddChange depends on the following definitions:
    flight_sched3.change_flight
    flight_sched3.add_flight
    flight_sched3.scheduled?
    notequal./=
```

Save Context

The command `M-x sc` saves the context in the file `.pvscontext`.

Proofs are not saved by this, they are automatically saved everytime you perform a `M-x pr`.

This saves the status of proofs and `.bin` files, which speeds up typechecking.

This command is issued for you when you exit PVS.

Issuing this command periodically, protects you when PVS crashes.

This is usually only important when you are working on large projects.

Change Context

If you want to work in a different directory without exiting PVS:

```
M-x cc
```

If you want to save the status of proofs in case PVS crashes:

```
M-x sc
```

But periodically one should

```
rm pvsbin/*.bin
rm .pvscontext
```

and `M-x spi` **from top theory.**

When PVS Is Acting Very Wierd 1

- Sometimes the incremental typechecker loses its mind.
- This is manifested by strange error messages or crashing into Lisp during typechecking.
- Try:

```
C-u M-x tc
```

- This forces a complete re-typecheck (i.e. it does not use any info from previous typechecks).

When PVS Is Acting Very Wierd 2

When PVS is acting very wierd, try

1. **Exit PVS**
2. **rm pvsbin/*.bin**
3. **rm .pvscontext**
4. **Restart PVS**

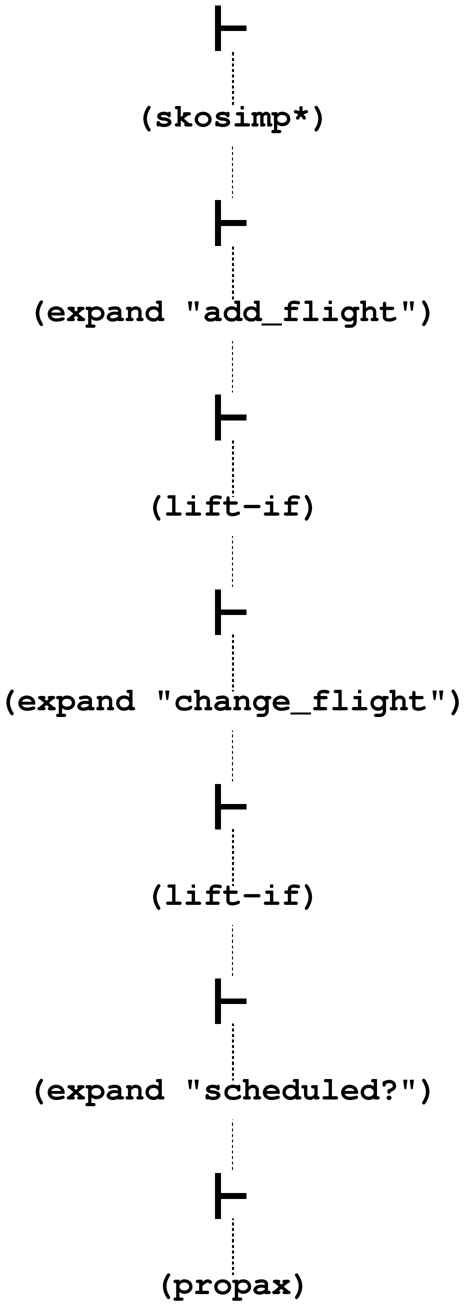
There are still some latent bugs that are associated with restoring typecheck info from `.bin` files after you make changes deep down in a specification. This works around that problem.

Sending Bug Reports to SRI:

Send pvs-dump to `pvs-bugs@csl.sri.com`
Check existing bugs at:
`http://pvs.csl.sri.com/cgi-bin/pvs/pvs-bug-list/`

X Windows Commands

The commands `M-x x-show-proof` can be used to display a proof in a graphical tree structure:



```
⊢
(skosimp*)
⊢
(expand "add_flight")
⊢
(lift-if)
⊢
(expand "change_flight")
⊢
(lift-if)
⊢
(expand "scheduled?")
⊢
(propax)
```

Prelude Commands

- **The command `M-x vpf` (or `M-x view-prelude-library` can be used to examine the PVS prelude.**
- **The command `M-x load-prelude-library` can be used to extend the PVS prelude with the contents of another directory.**

Help Commands

The command `help-pvs` or `C-c h` yields

```
PVS System Command List

--------------------------------Exiting PVS-----------------------------------

suspend-pvs (C-x C-z)       Suspend PVS
exit-pvs (C-x C-c)          Terminate the PVS session

------------------------------Getting Help------------------------------------

help-pvs, pvs-help (C-c h)  - Display help for the PVS system commands
help-pvs-language, pvs-help-language (C-c C-h l)
    - Display help for the PVS language


------------------------Parsing and Typechecking------------------------------
With an argument, these will force reparsing/retypechecking.

typecheck, tc (C-c t)       Typecheck the PVS file in the current buffer
typecheck-prove, tcp        Typecheck the PVS file in the current buffer,
    and attempts to prove the TCCs

-----------------------Prover Invocation Commands-----------------------------

prove,            pr  (C-c p)      Prove formula pointed to by cursor
```

...

Help Commands (cont).

The command `help-pvs-prover` or `C-c C-h p` yields:

```
 ----------------------------------------------------------------------
The rules (in alphabetical order) are:

(APPLY STRATEGY &OPTIONAL COMMENT):
    Applies STRATEGY as if it were a rule, and prints COMMENT string.
This is the basic way of converting a glass-box strategy into an
atomic step so that internal execution of the strategy is hidden
and only the resulting subgoals are returned.  E.g.,
 (apply (then (skosimp*)(flatten)(inst?))
      "Skolemizing, flattening, and instantiating").

(AUTO-REWRITE &REST NAMES):
    Installs automatic rewrite rules with given NAMES.
The rewrites are applied by the ASSERT and DO-REWRITE commands.
Each entry in the list NAMES is either an antecedent FNUM or
names a definition, assumption or
lemma/theorem/formula/proposition/conjecture, or a list of such
to indicate always-rewrites (as explained below).
In (AUTO-REWRITE A (B C) D (E F)), B, C, E, and F are always-rewrites.
Rewrites only take effect if relevant conditions and TCCs simplify to T.
If there is an IF or CASES at the outermost part of the
righthand-side, then it is treated as a condition except in the
case of an always-rewrite.  E.g.,
  (auto-rewrite "assoc" ("delete" "union") -3 (-4 -5))

...
```

Help Commands (cont.)

The command `help-pvs-prover-emacs` or `C-c C-h e` yields

```
PVS prover command     Key      Comments
------------------     ---      --------
Any Command            TAB      Prompts for command name
apply-extensionality   E        Prompts for formula number
assert                 a
auto-rewrite           A        Uses formula at point, or prompts
auto-rewrite-theory    C-a      Prompts for theory
bddsimp                B
beta                   b        Prompts for formula number
case                   c        Prompts for expression
case-replace           C        Prompts for expression
do-rewrite             D        Prompts for formula number
expand                 e        Expand definition at point
flatten                f
grind                  G
ground                 g
hide                   C-h      Prompts for formula number
iff                    F        Prompts for formula number
induct                 I        Prompts for variable name
inst                   i        Prompts for formula number and expressions
inst?                  ?        Prompts for formula number, additional arguments
lemma                  L        Prompts for lemma name
lift-if                l        Prompts for formula number
```

...

Retrieving Previous Prover Commands

- **When cursor is next to `Rule?`, you can cycle back through previously entered proof commands by repeatedly typing `M-s`**

```
Rule? (HIDE -2 -4 -6 -7 -8)
Rule? (ASSERT)
Rule? (LEMMA "lub_int")
Rule? (LEMMA "axiom_of_archimedes")
```

- **If you type in the first few letters of a command before typing the `M-s` you will cycle back through the commands that match these letters. For example**

```
Rule? (lem
```

The type `M-s` repeatedly you will get

```
Rule? (LEMMA "lub_int")
Rule? (LEMMA "axiom_of_archimedes")
Rule? (LEMMA "mother_of_all_theorems")
Rule? (LEMMA "existence_of_Atlantis")
```

A Little Example

```
Agh: THEORY
BEGIN

    x: VAR posreal
    y: VAR real

    sqrt(x: posreal): real

    sqrt_ax: AXIOM sqrt(x) = y IFF y*y = x

    goodness_me: THEOREM FALSE


END Agh
```

A Little Example(cont)

```
goodness_me :

  |-------
{1}   FALSE

Rule? (LEMMA "sqrt_ax")

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

Rule? (INST-CP -1 "4" "2")

[-1]  FORALL (x, y: real): sqrt(x) = y IFF y * y = x
{-2}  sqrt(4) = 2 IFF 2 * 2 = 4
  |-------

Rule? (INST -1 "4" "-2")

{-1}  sqrt(4) = -2 IFF -2 * -2 = 4
[-2]  sqrt(4) = 2 IFF 2 * 2 = 4
  |-------

Rule? (GROUND)
Q.E.D.
```

A Little Example(cont)

`M-x spc` **reveals**

```
Agh.goodness_me has been PROVED.

  The proof chain for goodness_me is COMPLETE.

  goodness_me depends on the following axioms:
    Agh.sqrt_ax
```

`M-x spt` **reveals**

```
 Proof summary for theory Agh
    goodness_me.................................proved - complete   [0](0.05 s)
    Theory totals: 1 formulas, 1 attempted, 1 succeeded (0.05 s)
```

Agh!!!! What Happened?

Predicate Subtype Abbreviation

```
T: TYPE
x: VAR T
P(x): bool    % or P: pred[T]

ST: TYPE = {t: T | P(t)}
```

The following is equivalent:

```
T: TYPE
x: VAR T
P(x): bool    % or P: pred[T]

ST: TYPE = (P)
```

One can use this wherever a type is required:

```
P(x): bool

f: VAR [(P) -> real]
```

Some Useful GRIND options

`GRIND` **performs auto-rewrite-defs/theories, auto-rewrite then applies skolem!, inst?, lift-if, bddsimp, and assert, until nothing works.**

`(GRIND :if-match NIL)` —–- **turns off instantiation**

`(GRIND :exclude "foo" )` —–- **don’t rewrite ”foo”**

`(GRIND :defs NIL)` —–- **defns not installed as rewrites**

Online Help

`http://shemesh.larc.nasa.gov/fm/`
`http://pvs.csl.sri.com/`
`pvs-help@csl.sri.com`
`pvs-bugs@csl.sri.com`

Moderated list for topics of general interest to all PVS users:

`pvs@csl.sri.com`

This list is low-traffic, and contains calls for papers, announcements of new PVS related papers, and notifications of new releases of PVS. To subscribe, send message with 'subscribe' to pvs-request@csl.sri.com.

At

`http://pvs.csl.sri.com/documentation.shtml`

PVS Manuals PVS Tutorials PVS Papers PVS Bibliography

Logic Refresher

Given

`X = the domain of all animals`

and two predicates

`cat(x) <--> x is a cat`

`is finicky(x) <--> the animal x is finicky`

translate the following English statements into logic:

`All animals that are cats are finicky`

`There is a cat that is finicky`

Logic Refresher (cont.)

`All animals that are cats are finicky`

$$\forall x : \textbf{cat}(x) \supset \textbf{finicky}(x)$$

`There is a cat that is finicky`

$$\exists x : \textbf{cat}(x) \wedge \textbf{finicky}(x)$$

MORAL: Existential quantification almost never involves $\supset$!

Logic Refresher (cont.)

This problem can be avoided in PVS by using predicate subtypes:

```
X: TYPE

cat(x: X): bool

cats: TYPE = {x: X | cat(x)}

finicky(x: X): bool

l1: LEMMA FORALL (c: cats): finicky(c)

l2: LEMMA EXISTS (c: cats): finicky(c)
```

But we still need to be aware of what is really going on!

Rewrite “congruence”

```
{-1}  P(x1!1, y1!1, y2!1)
{-2}  P(x2!1, y2!1, y2!1)
  |-------
{1}   f( x1!1 * 200,  y1!1 * sqrt(x1!1),  sq(x1!1) / y1!1) =
       f( x2!1 * 300,  y2!1 * exp(y2!1) / x2!1,  z2!1)

Rule?  (rewrite "congruence")
Found matching substitution:
x2: D gets (x2!1 * 300, y2!1 * exp(y2!1) / x2!1, z2!1),g: [D -> R] g
x1 gets (x1!1 * 200, y1!1 * sqrt(x1!1), sq(x1!1) / y1!1),f gets f,
Rewriting using congruence, matching in *, this simplifies to:

[-1]  P(x1!1, y1!1, y2!1)
[-2]  P(x2!1, y2!1, y2!1)
  |-------
{1}   200 * x1!1 = 300 * x2!1 AND
        y1!1 * sqrt(x1!1) = y2!1 * exp(y2!1) / x2!1 AND
        sq(x1!1) / y1!1 = z2!1
{2}   f(200 * x1!1, y1!1 * sqrt(x1!1), sq(x1!1) / y1!1) =
       f(300 * x2!1, y2!1 * exp(y2!1) / x2!1, z2!1)
```

Decompose-equality

If you have

```
{-1}  P(x!1, y!1, z!1)
  |-------
{1}   g = h
```

You can use `TAB E` **or** `(apply-extensionality  :hide? t)` **to get**
`[1]  FORALL (x: real): g(x) = h(x).`

But what if you have `{-1}  g = h`**, then you need something else:**

```
{-1}  g = h
  |-------
{1}   P(x!1, y!1, z!1)

Rule? (decompose-equality -1)
Applying decompose-equality,
this simplifies to:
T2 :

{-1}  FORALL (x: real): g(x) = h(x)
  |-------
[1]   P(x!1, y!1, z!1)
```

TAB Y

You have an old proof that used to work, but now is broken. You run the proof and end up with

```
atanh_taylors_prep6.1.1.1.1.1.1.1.1.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1

[-1]  FS11 + FS12 + FS1 + FS23 + FS33 = atanhF(1 + n!1)
[-2]  polynomial(FS11 + FS12 + FS1 + FS23, 2 + 2 * n!1) +
       polynomial(FS33, 2 + 2 * n!1)
       = polynomial(FS11 + FS12 + FS1 + FS23 + FS33, 2 + 2 * n!1)
[-3]  polynomial(FS11 + FS12 + FS1, 2 + 2 * n!1) +
       polynomial(FS23, 2 + 2 * n!1)
       = polynomial(FS11 + FS12 + FS1 + FS23, 2 + 2 * n!1)
  |-------
[1]   n!1 = 0
{2}   polynomial(FS23, 2 + 2 * n!1)(x!1) +
       polynomial(FS33, 2 + 2 * n!1)(x!1)   +
       polynomial(LAMBDA (x_1: nat): FS11(x_1) + FS12(x_1) + FS1(x_1
                  2 + 2 * n!1)(x!1)
       = polynomial(atanhF(1 + n!1), 2 + 2 * n!1)(x!1)

Rule?
```

But where in the proof file is this thing?

TAB Y (cont.)

So you open up the proof buffer with M-x edit-proof.
Return to the *pvs* buffer, and put the cursor at the beginning of the line:

```
atanh_taylors_prep6.1.1.1.1.1.1.1.1.1.2.1.1.1.1.1.1.1.1.1.1.1.1.1.1
```

```
 "extensionality"
 ("f"
  "FS11 + FS12 + FS1
  "g"
  "atanhF(1+n!1)"))
(split -1)
(("1"
  (replace -1 -2)

  (replace -3 -2 rl)
  (replace -2 2 rl)
  (expand "+" 2)
  (propax))
 ("2"
  (hide-all-but (1 2)
  (skosimp*)
  (expand*
```

TAB C-u

Suppose you are currently located in the proof buffer at `uglyproof.1` after issuing the following proof steps:

```
(assert)
(expand "sqrt")
(expand "discr")
(assert)
(split -1)
(("1"
  (expand "root")
  (expand "discr")
  (expand "sq")
  (bddsimp)
```

The command

```
TAB <cntrl>-u
```

will return you to the sequent at the beginning of the branch: at the `(expand "root")` command.

Some Final Hints

- **Show all proofs of theory:** `M-x show-proofs-theory`
- **What uses this lemma?:** `M-x usedby-proofs`
- **Save older proofs too:** `M-x set-proof-backup-number`
- **Where is this symbol defined?:** `M-.` **or** `M-,`
- **You convert a lemma to an axiom and can't get rid of the "unfinished" status. Use** `M-x remove-proof`
- **See visual graph of your theory structure:** `M-x x-theory-hierarchy`