Adapting Proof General

Proof General — Organize your proofs!

Adapting Proof General 4.5 to new provers
July 2022
proofgeneral.github.io

David Aspinall with T. Kleymann

This manual and the program Proof General are Copyright © 2000-2011 by members of the Proof General team, LFCS Edinburgh.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

This manual documents Proof General, Version 4.5, for use GNU Emacs 24.3 or (as far as possible) later versions. Proof General is distributed under the terms of the GNU General Public License (GPL), version 3 or later; please check the accompanying file `COPYING` for more details.

Visit Proof General on the web at `https://proofgeneral.github.io`

Introduction

Welcome to Proof General!

Proof General a generic Emacs-based interface for proof assistants.

This manual contains information for adapting Proof General to new proof assistants, and some sketches of the internal implementation. It is not intended for most ordinary users of the system. For full details about how to use Proof General, and information on its availability and installation, please see the main Proof General manual which should accompany this one.

We positively encourage the support of new systems. Proof General has grown more flexible and useful as it has been adapted to more proof assistants.

Typically, adding support for a new prover improves support for others, both because the code becomes more robust, and because new ideas are brought into the generic setting. Notice that the Proof General framework has been built as a "product line architecture": generality has been introduced step-by-step in a demand-driven way, rather than at the outset as a grand design. Despite this strategy, the interface has a surprisingly clean structure. The approach means that we fully expect hiccups when adding support for new assistants, so the generic core may need extension or modification. To support this we have an open development method: if you require changes in the generic support, please contact us (or make adjustments yourself and send them to us).

Proof General has a home page at `https://proofgeneral.github.io`. Visit this page for the latest version of the manuals, other documentation, system downloads, etc.

Future

The aim of the Proof General project is to provide a powerful and configurable interfaces which help user-interaction with interactive proof assistants.

The strategy Proof General uses is to targets power users rather than novices; other interfaces have often neglected this class of users. But we do include general user interface niceties, such as toolbar and menus, which make use easier for all.

Proof General has been Emacs based so far, but plans are afoot to liberate it from the points and parentheses of Emacs Lisp. The successor project Proof General Kit proposes that proof assistants use a *standard* XML-based protocol for interactive proof, dubbed **PGIP**.

PGIP enables middleware for interactive proof tools and interface components. Rather than configuring Proof General for your proof assistant, you will need to configure your proof assistant to understand PGIP. There is a similarity however; the design of PGIP was based heavily on the Emacs Proof General framework. This means that effort on customizing Emacs Proof General to a new proof assistant is worthwhile even in the light of PGIP: it will help you to understand Proof General's model of interaction, and moreover, we hope to use the Emacs customizations to provide experimental filters which allow supported provers to communicate using PGIP.

At the time of writing, these ideas are in early stages. For latest details, or to become involved, see the Proof General Kit webpage (`http://proofgeneral.inf.ed.ac.uk/kit`).

Credits

David Aspinall put together and wrote most of this manual. Thomas Kleymann wrote some of the text in Chapter 8. Much of the content is generated automatically from Emacs docstrings, some of which have been written by other Proof General developers.

1 Beginning with a New Prover

Proof General has about 100 configuration variables which are set on a per-prover basis to configure the various features. It may sound like a lot but don't worry! Many of the variables occur in pairs (typically regular expressions matching the start and end of some text), and you can begin by setting just a fraction of the variables to get the basic features of script management working. The bare minimum for a working prototype is about 25 simple settings.

For more advanced features you may need (or want) to write some Emacs Lisp. If you're adding new functionality please consider making it generic for different proof assistants, if appropriate. When writing your modes, please follow the Emacs Lisp conventions, See Info file `Elisp`, node '`Tips`'.

The configuration variables are declared in the file `generic/proof-config.el`. The details in the central part of this manual are based on the contents of that file, beginning in Chapter 2 [Menus and Toolbar and User-level Commands], page 7, and continuing until Chapter 7 [Global Constants], page 33. Other chapters cover the details of configuring for multiple files and for supporting the other Emacs packages mentioned in the user manual (*Support for other Packages*). If you write additional Elisp code interfacing to Proof General, you can find out about some useful functions by reading Chapter 13 [Writing More Lisp Code], page 49. The last chapter of this manual describes some of the internals of Proof General, in case you are interested, maybe because you need to extend the generic core to do something new.

In the rest of this chapter we describe the general mechanisms for instantiating Proof General. We assume some knowledge of the content of the main Proof General manual.

1.1 Overview of adding a new prover

Each proof assistant supported has its own subdirectory under `proof-home-directory`, used to store a root elisp file and any other files needed to adapt the proof assistant for Proof General.

Here is how to go about adding support for a new prover.

1. Make a directory called `myassistant/` under the Proof General home directory `proof-home-directory`, to put the specific customization and associated files in.
2. Add a file `myassistant.el` to the new directory.
3. Edit `proof-site.el` to add a new entry to the `proof-assistants-table` variable. The new entry should look like this:

   ```
   (myassistant "My Proof Assistant" "\\.myasst$")
   ```

   The first item is used to form the name of the internal variables for the new mode as well as the directory and file where it loads from. The second is a string, naming the proof assistant. The third item is a regular expression to match names of proof script files for this assistant. See the documentation of `proof-assistant-table` for more details.
4. Define the new Proof General modes in `myassistant.el`, by setting configuration variables to customize the behaviour of the generic modes.

`proof-assistant-table` [Variable]

Proof General's table of supported proof assistants.

This is copied from '`proof-assistant-table-default`' at load time, removing any entries that do not have a corresponding directory under '`proof-home-directory`'.

Each entry is a list of the form

```
(symbol name file-extension [AUTOMODE-REGEXP] [IGNORED-EXTENSIONS-LIST])
```

The *name* is a string, naming the proof assistant. The *symbol* is used to form the name of the mode for the assistant, '`SYMBOL-mode`', run when files with *automode-regexp* (or with extension *file-extension*) are visited. If present, *ignored-extensions-list* is a list of file-name

extensions to be ignored when doing file-name completion (*ignored-extensions-list* is added to '`completion-ignored-extensions`').

symbol is also used to form the name of the directory and elisp file for the mode, which will be

proof-home-directory/symbol/symbol`.el`

where *proof-home-directory* is the value of the variable '`proof-home-directory`'.

The final step of the description above is where the work lies. There are two basic methods. You can write some Emacs lisp functions and define the modes using the macro `define-derived-mode`. Or you can use the new easy configuration mechanism of Proof General 3.0 described in the next section, which calls `define-derived-mode` for you. You still need to know which configuration variables should be set, and how to set them.

The documentation below (and inside Emacs) should help with that, but the best way to begin might be to use an existing Proof General instance as an example.

1.2 Demonstration instance and easy configuration

Proof General is supplied with a demonstration instance for Isabelle which configures the basic features. This is a whittled down version of Isabelle Proof General, which you can use as a template to get support for a new assistant going. Check the directory `demoisa` for the two files `demoisa.el` and `demoisa-easy.el`.

The file `demoisa.el` follows the scheme described in Section 1.3 [Major modes used by Proof General], page 5. It uses the Emacs Lisp macro `define-derived-mode` to define the four modes for a Proof General instance, by inheriting from the generic code. Settings which configure Proof General are made by functions called from within each mode, as appropriate.

The file `demoisa-easy.el` uses a new simplified mechanism to achieve (virtually) the same result. It uses the macro `proof-easy-config` defined in `proof-easy-configl.el` to make all of the settings for the Proof General instance in one go, defining the derived modes automatically using a regular naming scheme. No lisp code is used in this file except the call to this macro. The minor difference in the end result is that all the variables are set at once, rather than inside each mode. But since the configuration variables are all global variables anyway, this makes no real difference.

The macro `proof-easy-config` is called like this:

```
(proof-easy-config myprover "MyProver"
     config_1 val_1
     ...
     config_n val_n)
```

The main body of the macro call is like the body of a `setq`. It contains pairs of variables and value settings. The first argument to the macro is a symbol defining the mode root, the second argument is a string defining the mode name. These should be the same as the first part of the entry in `proof-assistant-table` for your prover. See Section 1.1 [Overview of adding a new prover], page 3. After the call to `proof-easy-config`, the new modes *myprover*`-mode`, *myprover*`-shell-mode`, *myprover*`-response-mode`, and *myprover*`-goals-mode` will be defined. The configuration variables in the body will be set immediately.

This mechanism is in fact recommended for new instantiations of Proof General since it follows a regular pattern, and we can more easily adapt it in the future to new versions of Proof General.

Even Emacs Lisp experts should prefer the simplified mechanism. If you want to set some buffer-local variables in your Proof General modes, or invoke supporting lisp code, this can easily be done by adding functions to the appropriate mode hooks after the `proof-easy-config` call. For example, to add extra settings for the shell mode for `demoisa`, we could do this:

```
(defun demoisa-shell-extra-config ()
```

```
    extra configuration ...
  )
(add-hook 'demoisa-shell-mode-hook 'demoisa-shell-extra-config)
```

The function to do extra configuration `demoisa-shell-extra-config` is then called as the final step when `demoisa-shell-mode` is entered (be wary, this will be after the generic `proof-shell-config-done` is called, so it will be too late to set normal configuration variables which may be examined by `proof-shell-config-done`).

1.3 Major modes used by Proof General

There are four major modes used by Proof General, one for each type of buffer it handles. The buffer types are: script, shell, response and goals. Each of these has a generic mode, respectively: `proof-mode`, `proof-shell-mode`, `proof-response-mode`, and `proof-goals-mode`.

The pattern for defining the major mode for an instance of Proof General is to use `define-derived-mode` to define a specific mode to inherit from each generic one, like this:

```
(define-derived-mode myass-shell-mode proof-shell-mode
   "MyAss shell" nil
   (myass-shell-config)
   (proof-shell-config-done))
```

Where `myass-shell-config` is a function which sets the configuration variables for the shell (see Chapter 4 [Proof Shell Settings], page 19).

It's important that each of your modes invokes one of the functions `proof-config-done`, `proof-shell-config-done`, `proof-response-config-done`, or `proof-goals-config-done` once it has set its configuration variables. These functions finalize the configuration of the mode.

The modes must be named standardly, replacing `proof-` with the prover's symbol name, *`PA-`*. In other words, you must define *`PA-`*`mode`, *`PA-`*`shell-mode`, etc.

See the file `demoisa.el` for an example of the four calls to `define-derived-mode`.

Aside: notice that the modes are selected using stub functions inside `proof-site.el`, which set the variables `proof-mode-for-script`, `proof-mode-for-shell`, etc, that actually select the right mode. These variables are declared in `pg-vars.el`.

2 Menus, toolbar, and user-level commands

The variables described in this chapter configure the menus, toolbar, and user-level commands. They should be set in the script mode before `proof-config-done` is called. (Toolbar configuration must be made before `proof-toolbar.el` is loaded, which usually is triggered automatically by an attempt to display the toolbar).

2.1 Settings for generic user-level commands

`proof-assistant-home-page` [Variable]
Web address for information on proof assistant.
Used for Proof General's help menu.

`proof-context-command` [Variable]
Command to display the context in proof assistant.

`proof-info-command` [Variable]
Command to ask for help or information in the proof assistant.
String or fn. If a string, the command to use. If a function, it should return the command string to insert.

`proof-showproof-command` [Variable]
Command to display proof state in proof assistant.

`proof-goal-command` [Variable]
Command to set a goal in the proof assistant. String or fn.
If a string, the format character '`%s`' will be replaced by the goal string. If a function, it should return the command string to insert.

`proof-save-command` [Variable]
Command to save a proved theorem in the proof assistant. String or fn.
If a string, the format character '`%s`' will be replaced by the theorem name. If a function, it should return the command string to insert.

`proof-find-theorems-command` [Variable]
Command to search for a theorem containing a given term. String or fn.
If a string, the format character '`%s`' will be replaced by the term. If a function, it should return the command string to send.

2.2 Menu configuration

As well as the generic Proof General menu, each proof assistant is provided with a specific menu which can have prover-specific commands. Proof General puts some default things on this menu, including the commands to start/stop the prover, and the user-extensible "Favourites" menu.

`PA-menu-entries` [Variable]
Extra entries for proof assistant specific menu.
A list of menu items [*name callback enabler* ...]. See the documentation of '`easy-menu-define`' for more details.

`PA-help-menu-entries` [Variable]
Extra entries for help submenu for proof assistant specific help menu.
A list of menu items [*name callback enabler* ...]. See the documentation of '`easy-menu-define`' for more details.

2.3 Toolbar configuration

Unlike the menus, Proof General has only one toolbar. For the "generic" aspect of Proof General to work well, we shouldn't change (the meaning of) the existing toolbar buttons too far. This would discourage people from experimenting with different proof assistants when they don't really know them, which is one of the advantages that Proof General brings.

But in case it is hard to map some of the generic buttons onto functions in particular provers, and to allow extra buttons, there is a mechanism for adjustment.

I used The Gimp to create the buttons for Proof General. The development distribution includes a button blank and some notes in `etc/notes.txt` about making new buttons.

`proof-toolbar-entries-default` [Variable]

Example value for proof-toolbar-entries. Also used to define scripting menu.
This gives a bare toolbar that works for any prover, providing the appropriate configuration variables are set. To add/remove prover specific buttons, adjust the '`<PA>-toolbar-entries`' variable, and follow the pattern in '`proof-toolbar.el`' for defining functions, images.

`PA-toolbar-entries` [Variable]

List of entries for Proof General toolbar and Scripting menu.
Format of each entry is (*token menuname tooltip toolbar-p* [VISIBLE-P]).

For each *token*, we expect an icon with base filename *token*, a function proof-toolbar-<TOKEN>, and (optionally) a dynamic enabler proof-toolbar-<TOKEN>-enable-p.

If *visible-p* is absent, or evaluates to non-nil, the item will appear on the toolbar or menu. If it evaluates to nil, the item is not shown.

If *menuname* is nil, item will not appear on the scripting menu.

If *toolbar-p* is nil, item will not appear on the toolbar.

The default value is '`proof-toolbar-entries-default`' which contains the standard Proof General buttons.

Here's an example of how to remove a button, from `af2.el`:

```
(setq af2-toolbar-entries
      (assq-delete-all 'state af2-toolbar-entries))
```

3 Proof Script Settings

The variables described in this chapter should be set in the script mode before `proof-config-done` is called. These variables configure recognition of commands in the proof script, and also control some of the behaviour of script management.

3.1 Recognizing commands and comments

The first four settings configure the generic parsing strategy for commands in the proof script. Usually only one of these three needs to be set. If the generic parsing functions are not flexible for your needs, you can supply a value for `proof-script-parse-function`.

Note that for the generic functions to work properly, it is **essential** that you set the syntax table for your mode properly, so that comments and strings are recognized. See the Emacs documentation to discover how to do this (particularly for the function `modify-syntax-entry`, (See Info file `Elisp`, node '`Syntax Tables`').

See Section 14.5 [Proof script mode], page 56, for more details of the parsing functions.

`proof-terminal-string` [Variable]
String that terminates commands sent to prover; nil if none.

To configure command recognition properly, you must set at least one of these: '`proof-script-sexp-commands`', '`proof-script-command-end-regexp`', '`proof-script-command-start-regexp`', '`proof-terminal-string`', or '`proof-script-parse-function`'.

`proof-electric-terminator-noterminator` [Variable]
If non-nil, electric terminator does not actually insert a terminator.

`proof-script-sexp-commands` [Variable]
Non-nil if script has Lisp-like syntax: commands are `top-level` sexps.
You should set this variable in script mode configuration.

To configure command recognition properly, you must set at least one of these: '`proof-script-sexp-commands`', '`proof-script-command-end-regexp`', '`proof-script-command-start-regexp`', '`proof-terminal-string`', or '`proof-script-parse-function`'.

`proof-script-command-start-regexp` [Variable]
Regular expression which matches start of commands in proof script.
You should set this variable in script mode configuration.

To configure command recognition properly, you must set at least one of these: '`proof-script-sexp-commands`', '`proof-script-command-end-regexp`', '`proof-script-command-start-regexp`', '`proof-terminal-string`', or '`proof-script-parse-function`'.

`proof-script-command-end-regexp` [Variable]
Regular expression which matches end of commands in proof script.
You should set this variable in script mode configuration.

The end of the command is considered to be the end of the match of this regexp. The regexp may include a nested group, which can be used to recognize the start of the following command (or white space). If there is a nested group, the end of the command is considered to be the start of the nested group, i.e. (`match-beginning` 1), rather than (`match-end` 0).

To configure command recognition properly, you must set at least one of these: '`proof-script-sexp-commands`', '`proof-script-command-end-regexp`', '`proof-script-command-start-regexp`', '`proof-terminal-string`', or '`proof-script-parse-function`'.

The next four settings configure the comment syntax. Notice that to get reliable behaviour of the parsing functions, you may need to modify the syntax table for your prover's mode. Read the Elisp manual (See Info file `Elisp`, node '`Syntax Tables`') for details about that.

`proof-script-comment-start` [Variable]
String which starts a comment in the proof assistant command language.
The script buffer's '`comment-start`' is set to this string plus a space. Moreover, comments are usually ignored during script management, and not sent to the proof process.

You should set this variable for reliable working of Proof General, as well as '`proof-script-comment-end`'.

`proof-script-comment-start-regexp` [Variable]
Regexp which matches a comment start in the proof command language.

The default value for this is set as (`regexp-quote` '`proof-script-comment-start`') but you can set this variable to something else more precise if necessary.

`proof-script-comment-end` [Variable]
String which ends a comment in the proof assistant command language.
Should be an empty string if comments are terminated by '`end-of-line`' The script buffer's '`comment-end`' is set to a space plus this string, if it is non-empty.

See also '`proof-script-comment-start`'.

You should set this variable for reliable working of Proof General.

`proof-script-comment-end-regexp` [Variable]
Regexp which matches a comment end in the proof command language.

The default value for this is set as (`regexp-quote` '`proof-script-comment-end`') but you can set this variable to something else more precise if necessary.

`proof-case-fold-search` [Variable]
Value for '`case-fold-search`' when recognizing portions of proof scripts.
Also used for completion, via '`proof-script-complete`'. The default value is nil. If your prover has a case **insensitive** input syntax, '`proof-case-fold-search`' should be set to t instead. NB: This setting is not used for matching output from the prover.

Finally, the function `proof-looking-at-syntactic-context` is used internally to help determine the syntactic structure of the buffer. You can test it to check the settings above. If necessary, you can override this with a system-specific function.

`proof-looking-at-syntactic-context` [Function]
Determine if current point is at beginning or within comment/string context.
If so, return a symbol indicating this ('comment or '`'string`). This function invokes <PA-syntactic-context> if that is defined, otherwise it calls '`proof-looking-at-syntactic-context`'.

3.2 Recognizing proofs

Several settings each may be supplied for recognizing goal-like and save-like commands. The `-with-hole-` settings are used to make a record of the name of the theorem proved.

The `-p` subsidiary predicates were added to allow more discriminating behaviour for particular proof assistants. (This is a typical example of where the core framework needs some additional generalization, to simplify matters, and allow for a smooth handling of nested proofs; the present state is only part of the way there).

`proof-goal-command-regexp` [Variable]
Matches a goal command in the proof script.
This is used to make the default value for '`proof-goal-command-p`', used as an important part of script management to find the start of an atomic undo block.

`proof-goal-command-p` [Variable]
A function to test: is this really a goal command span?

This is added as a more refined addition to '`proof-goal-command-regexp`', to solve the problem that Coq and some other provers can have goals which look like definitions, etc. (In the future we may generalize '`proof-goal-command-regexp`' instead).

`proof-goal-with-hole-regexp` [Variable]
Regexp which matches a command used to issue and name a goal.
The name of the theorem is built from the variable '`proof-goal-with-hole-result`' using the same convention as for '`query-replace-regexp`'. Used for setting names of goal..save regions and for default configuration of other modes (function menu, imenu).

It's safe to leave this setting as nil.

`proof-goal-with-hole-result` [Variable]
How to get theorem name after '`proof-goal-with-hole-regexp`' match.
String or Int. If an int N, use '`match-string`' to get the value of the Nth parenthesis matched. If a string, use '`replace-match`'. In this case, '`proof-goal-with-hole-regexp`' should match the entire command.

`proof-save-command-regexp` [Variable]
Matches a save command.

`proof-save-with-hole-regexp` [Variable]
Regexp which matches a command to save a named theorem.
The name of the theorem is built from the variable '`proof-save-with-hole-result`' using the same convention as '`query-replace-regexp`'. Used for setting names of goal..save and proof regions.

It's safe to leave this setting as nil.

`proof-completed-proof-behaviour` [Variable]
Indicates how Proof General treats commands beyond the end of a proof.
Normally goal...save regions are "closed", i.e. made atomic for undo. But once a proof has been completed, there may be a delay before the "save" command appears — or it may not appear at all. Unless nested proofs are supported, this can spoil the undo-behaviour in script management since once a new goal arrives the old undo history may be lost in the prover. So we allow Proof General to close off the goal..[save] region in more flexible ways. The possibilities are:

```
           nil  -  nothing special; close only when a save arrives
     'closeany  -  close as soon as the next command arrives, save or not
    'closegoal  -  close when the next "goal" command arrives
      'extend   -  keep extending the closed region until a save or goal.
```

If your proof assistant allows nested goals, it will be wrong to close off the portion of proof so far, so this variable should be set to nil.

NB: `'extend` behaviour is not currently compatible with appearance of save commands, so don't use that if your prover has save commands.

`proof-really-save-command-p` [Variable]
Is this really a save command?

This is a more refined addition to '`proof-save-command-regexp`'. It should be a function taking a span and command as argument, and can be used to track nested proofs.

3.3 Recognizing other elements

To configure *Imenu* (which in turn configures *Speedbar*), you may use the following setting. If this is unset, a generic setting based on `proof-goal-with-hole-regexp` is configured.

`proof-script-imenu-generic-expression` [Variable]
Regular expressions to help find definitions and proofs in a script.
Value for '`imenu-generic-expression`', see documentation of Imenu and that variable for details.

`imenu-generic-expression` [Variable]
The regex pattern to use for creating a buffer index.

If non-nil this pattern is passed to '`imenu--generic-function`' to create a buffer index.

The value should be an alist with elements that look like this:

```
(menu-title regexp index)
```

or like this:

```
(menu-title regexp index function ARGUMENTS...)
```

with zero or more ARGUMENTS. The former format creates a simple element in the index alist when it matches; the latter creates a special element of the form (*name function position-marker* ARGUMENTS...) with *function* and *arguments* beiong copied from '`imenu-generic-expression`'.

menu-title is a string used as the title for the submenu or nil if the entries are not nested.

regexp is a regexp that should match a construct in the buffer that is to be displayed in the menu; i.e., function or variable definitions, etc. It contains a substring which is the name to appear in the menu. See the info section on Regexps for more information.

index points to the substring in *regexp* that contains the name (of the function, variable or type) that is to appear in the menu.

The variable is buffer-local.

The variable '`imenu-case-fold-search`' determines whether or not the regexp matches are case sensitive. and '`imenu-syntax-alist`' can be used to alter the syntax table for the search.

For example, see the value of '`lisp-imenu-generic-expression`' used by '`lisp-mode`' and '`emacs-lisp-mode`' with '`imenu-syntax-alist`' set locally to give the characters which normally have \"punctuation\" syntax \"word\" syntax during matching."

3.4 Configuring undo behaviour

The settings here are used to configure the way "undo" commands are calculated.

`proof-non-undoables-regexp` [Variable]
Regular expression matching commands which are **not** undoable.
These are commands which should not appear in proof scripts, for example, undo commands themselves (if the proof assistant cannot "redo" an "undo"). Used in default functions '`proof-generic-state-preserving-p`' and '`proof-generic-count-undos`'. If you don't use those, may be left as nil.

`proof-undo-n-times-cmd` [Variable]
Command to undo n steps of the currently open goal.
String or function. If this is set to a string, '`%s`' will be replaced by the number of undo steps to issue. If this is set to a function, it should return a list of the appropriate commands (given the number of undo steps).

This setting is used for the default '`proof-generic-count-undos`'. If you set '`proof-count-undos-fn`' to some other function, there is no need to set this variable.

`proof-ignore-for-undo-count` [Variable]
Matcher for script commands to be ignored in undo count.
May be left as nil, in which case it will be set to '`proof-non-undoables-regexp`'. Used in default function '`proof-generic-count-undos`'.

`proof-count-undos-fn` [Variable]
Function to calculate a list of commands to undo to reach a target span.
The function takes a span as an argument, and should return a string which is the command to undo to the target span. The target is guaranteed to be within the current (open) proof. This is an important function for script management. The default setting '`proof-generic-count-undos`' is based on the settings '`proof-non-undoables-regexp`' and '`proof-non-undoables-regexp`'.

`proof-generic-count-undos` *span* [Function]
Count number of undos in *span*, return commands needed to undo that far.
Command is set using '`proof-undo-n-times-cmd`'.

A default value for '`proof-count-undos-fn`'.

For this function to work properly, you must configure '`proof-undo-n-times-cmd`' and '`proof-ignore-for-undo-count`'.

`proof-find-and-forget-fn` [Variable]
Function to return list of commands to forget to before its argument span.
This setting is used to for retraction (undoing) in proof scripts.

It should undo the effect of all settings between its target span up to (`proof-unprocessed-begin`). This may involve forgetting a number of definitions, declarations, or whatever.

If return value is nil, it means there is nothing to do.

This is an important function for script management. Study one of the existing instantiations for examples of how to write it, or leave it set to the default function '`proof-generic-find-and-forget`' (which see).

`proof-generic-find-and-forget` *span* [Function]
Calculate a forget/undo command to forget back to *span*.
This is a long-range forget: we know that there is no open goal at the moment, so forgetting involves unbinding declarations, etc, rather than undoing proof steps.

This generic implementation assumes it is enough to find the nearest following span with a '`name`' property, and retract that using '`proof-forget-id-command`' with the given name.

If this behaviour is not correct, you must customize the function with something different.

`proof-forget-id-command` [Variable]
Command to forget back to a given named span.
A string; '`%s`' will be replaced by the name of the span.

This is only used in the implementation of '`proof-generic-find-and-forget`', you only need to set if you use that function (by not customizing '`proof-find-and-forget-fn`'.

`pg-topterm-goalhyplit-fn` [Variable]
Function to return cons if point is at a goal/hypothesis/literal.
This is used to parse the proofstate output to mark it up for proof-by-pointing or literal command insertion. It should return a cons or nil. First element of the cons is a symbol, '`goal`', '`hyp`' or '`lit`'. The second element is a string: the goal, hypothesis, or literal command itself.

If you leave this variable unset, no proof-by-pointing markup will be attempted.

`proof-kill-goal-command` [Variable]

Command to kill the currently open goal.

If this is set to nil, PG will expect '`proof-find-and-forget-fn`' to do all the work of retracting to an arbitrary point in a file. Otherwise, the generic split-phase mechanism will be used:

1. If inside an unclosed proof, use '`proof-count-undos`'. 2. If retracting to before an unclosed proof, use '`proof-kill-goal-command`', followed by '`proof-find-and-forget-fn`' if necessary.

3.5 Nested proofs

Proof General allows configuration for provers which have particular notions of nested proofs. The right thing may happen automatically, or you may need to adjust some of the following settings.

First, you should alter the next setting if the prover retains history for nested proofs.

`proof-nested-goals-history-p` [Variable]

Whether the prover supports recovery of history for nested proofs.
If it does (non-nil), Proof General will retain history inside nested proofs. If it does not, Proof General will amalgamate nested proofs into single steps within the outer proof.

Second, it may happen (i.e. it does for Coq) that the prover has a history mechanism which necessitates keeping track of the number of nested "undoable" commands, even if the history of the proof itself is lost.

`proof-nested-undo-regexp` [Variable]

Regexp for commands that must be counted in nested goal-save regions.

Used for provers which allow nested atomic goal-saves, but with some nested history that must be undone specially.

At the moment, the behaviour is that a goal-save span has a `'nestedundos` property which is set to the number of commands within it which match this regexp. The idea is that the prover-specific code can create a customized undo command to retract the goal-save region, based on the `'nestedundos` setting. Coq uses this to forget declarations, since declarations in Coq reside in a separate context with its own (flat) history.

3.6 Omitting proofs for speed

In normal operation, the commands in an asserted region are sent successively to the proof assistant. When the proof assistant reports an error, processing stops. This ensures the consistency of the development. Proof General supports omitting portions of the asserted region to speed processing up at the cost of consistency. Portions that can be potentially omitted are called *opaque proofs* in Proof General, because usually only opaque proofs (in the sense of Coq) can be omitted without risking to break the following code. This feature is also described in the Proof General manual, See Info file `ProofGeneral`, node '`Script processing commands`' and See Info file `ProofGeneral`, node '`Omitting proofs for speed`'.

The omit proofs feature works in a simple, straightforward way: After parsing the asserted region, Proof General uses regular expressions to search for commands that start (`proof-script-proof-start-regexp`) and end (`proof-script-proof-end-regexp`) an opaque proof. If one is found, the opaque proof is replaced with a cheating command (`proof-script-proof-admit-command`). From this description it is immediate, that the omit proof feature does only work if proofs are not nested. If a nested proof is found, a warning is displayed and omitting proofs stops at that location for the currently asserted region.

To enable the omit proofs feature, the following settings must be configured.

`proof-omit-proofs-configured` [Variable]
t if the omit proofs feature has been configured by the proof assitant.
See also '`proof-omit-proofs-option`' or the Proof General manual for a description of the feature. This option can only be set, if all of '`proof-script-proof-start-regexp`', '`proof-script-proof-end-regexp`', '`proof-script-definition-end-regexp`' and '`proof-script-proof-admit-command`' have been configured.

The omit proofs feature skips over opaque proofs in the source code, admitting the theorems, to speed up processing.

If '`proof-omit-proofs-option`' is set by the user, all proof commands in the source following a match of '`proof-script-proof-start-regexp`' up to and including the next match of '`proof-script-proof-end-regexp`', are omitted (not send to the proof assistant) and replaced by '`proof-script-proof-admit-command`'. If a match for '`proof-script-definition-end-regexp`' is found while searching forward for the proof end, the current proof (up to and including the match of '`proof-script-definition-end-regexp`') is considered to be not opaque and not omitted, thus all these proof commands _are_ sent to the proof assistant.

The feature does not work for nested proofs. If a match for '`proof-script-proof-start-regexp`' is found before the next match for '`proof-script-proof-end-regexp`' or '`proof-script-definition-end-regexp`', the search for opaque proofs immediately stops and all commands following the previous match of '`proof-script-proof-start-regexp`' are sent verbatim to the proof assistant.

All the regular expressions for this feature are matched against the commands inside proof action items, that is as strings, without surrounding space.

`proof-script-proof-start-regexp` [Variable]
Regular expression for the start of a proof for the omit proofs feature.
See '`proof-omit-proofs-configured`'.

`proof-script-proof-end-regexp` [Variable]
Regular expression for the end of an opaque proof for the omit proofs feature.
See '`proof-omit-proofs-configured`'.

`proof-script-definition-end-regexp` [Variable]
Regexp for the end of a non-opaque proof for the omit proofs feature.
See '`proof-omit-proofs-configured`'.

`proof-script-proof-admit-command` [Variable]
Proof command to be inserted instead of omitted proofs.

3.7 Safe (state-preserving) commands

A proof command is "safe" if it can be issued away from the proof script. For this to work it should be state-preserving in the proof assistant (with respect to an on-going proof).

`proof-state-preserving-p` [Variable]
A predicate, non-nil if its argument (a command) preserves the proof state.
This is a safety-test used by '`proof-minibuffer-cmd`' to filter out scripting commands which should be entered directly into the script itself.

The default setting for this function, '`proof-generic-state-preserving-p`' tests by negating the match on '`proof-non-undoables-regexp`'.

`proof-generic-state-preserving-p` *cmd* [Function]
Is *cmd* state preserving? Match on '`proof-non-undoables-regexp`'.

3.8 Activate scripting hook

`proof-activate-scripting-hook` [Variable]

Hook run when a buffer is switched into scripting mode.
The current buffer will be the newly active scripting buffer.

This hook may be useful for synchronizing with the proof assistant, for example, to switch to a new theory (in case that isn't already done by commands in the proof script).

When functions in this hook are called, the variable '`activated-interactively`' will be non-nil if '`proof-activate-scripting`' was called interactively (rather than as a side-effect of some other action). If a hook function sends commands to the proof process, it should wait for them to complete (so the queue is cleared for scripting commands), unless activated-interactively is set.

3.9 Automatic multiple files

See Chapter 8 [Handling Multiple Files], page 35, for more details about this setting.

`proof-auto-multiple-files` [Variable]

Whether to use automatic multiple file management.
If non-nil, Proof General will automatically retract a script file whenever another one is retracted which it depends on. It assumes a simple linear dependency between files in the order which they were processed.

If your proof assistant has no management of file dependencies, or one which depends on a simple linear context, you may be able to use this setting to good effect. If the proof assistant has more complex file dependencies then you should configure it to communicate with Proof General about the dependencies rather than using this setting.

3.10 Completely asserted buffers

When switching scripting from buffer A to buffer B Proof General normally offers the choice of either completely retracting or completely asserting buffer A. The option to completely assert buffer A is offered, because the material in B may depend on A. Even if B does not depend on A, it does no harm if one keeps the development of A loaded in the proof assistant. This observation is true for many proof assistants.

One exception is Coq. Assume file B depends on file A. When Coq processes B it does not read the sources of A. Instead it loads a compiled object representation of A. Therefore, when switching from A to B, it does make no sense to keep the material of A loaded in the proof assistant. For Coq, the material of A may even provoke errors on correct input. Therefore, for coq, the right behaviour is to completely retract buffer A before switching to B.

`proof-no-fully-processed-buffer` [Variable]

Set to t if buffers should always retract before scripting elsewhere.
Leave at nil if fully processed buffers make sense for the current proof assistant. If nil the user can choose to fully assert a buffer when starting scripting in a different buffer. If t there is only the choice to fully retract the active buffer before starting scripting in a different buffer. This last behavior is needed for Coq.

3.11 Completions

Proof General allows provers to create a *completion table* to help writing keywords and identifiers in proof scripts. This is documented in the main *Proof General* user manual but summarized here for (a different kind of) completion.

Completions are filled in according to what has been recently typed, from a database of symbols. The database is automatically saved at the end of a session. Completion is usually a hand-wavy thing, so we don't make any attempt to maintain a precise completion table or anything.

The completion table maintained by `complete.el` is initialized from `PA-completion-table` when `proof-script.el` is loaded. This is done with the function `proof-add-completions` which you may want to call at other times.

`PA-completion-table` [Variable]

List of identifiers to use for completion for this proof assistant.
Completion is activated with M-x complete.

If this table is empty or needs adjusting, please make changes using '`customize-variable`' and post suggestions at https://github.com/ProofGeneral/PG/issues

`proof-add-completions` [Command]

Add completions from <PA>-completion-table to completion database.
Uses '`add-completion`' with a negative number of uses and ancient last use time, to discourage saving these into the users database.

4 Proof Shell Settings

The variables in this chapter concern the proof shell mode, and are the largest group. They are split into several subgroups. The first subgroup are commands invoked at various points. The second subgroup of variables are concerned with matching the output from the proof assistant. The final subgroup contains various hooks which you can set to add lisp customization to Proof General in various points (some of them are also used internally for behaviour you may wish to adjust).

Variables for configuring the proof shell are put into the customize group `proof-shell`.

These should be set in the shell mode configuration, before `proof-shell-config-done` is called.

To understand the way the proof assistant runs inside Emacs, you may want to refer to the `comint.el` (Command interpreter) package distributed with Emacs. This package controls several shell-like modes available in Emacs, including the `proof-shell-mode` and all specific shell modes derived from it.

4.1 Commands

Settings in this section configure Proof General with commands to send to the prover to activate certain actions.

`proof-prog-name` [Variable]

System command to run the proof assistant in the proof shell.

May contain arguments separated by spaces, but see also the prover specific settings '`<PA>-prog-args`' and '`<PA>-prog-env`'.

Remark: if '`<PA>-prog-args`' is non-nil, then '`proof-prog-name`' is considered strictly: it must contain **only** the program name with no option, spaces are interpreted literally as part of the program name.

`PA-prog-args` [Variable]

Arguments to be passed to '`proof-prog-name`' to run the proof assistant.

If non-nil, will be treated as a list of arguments for '`proof-prog-name`'. Otherwise '`proof-prog-name`' will be split on spaces to form arguments.

Remark: Arguments are interpreted strictly: each one must contain only one word, with no space (unless it is the same word). For example if the arguments are -x foo -y bar, then the list should be '("-x" "foo" "-y" "bar"), notice that '("-x foo" "-y bar") is **wrong**.

`PA-prog-env` [Variable]

Modifications to '`process-environment`' made before running '`proof-prog-name`'.

Each element should be a string of the form ENVVARNAME=*value*. They will be added to the environment before launching the prover (but not pervasively). For example for coq on Windows you might need something like: (setq `coq-prog-env` '("HOME=C:\Program Files\Coq\"))

`proof-shell-auto-terminate-commands` [Variable]

Non-nil if Proof General should try to add terminator to every command.

If non-nil, whenever a command is sent to the prover using '`proof-shell-invisible-command`', Proof General will check to see if it ends with '`proof-terminal-string`', and add it if not. If '`proof-terminal-string`' is nil, this has no effect.

`proof-shell-pre-sync-init-cmd` [Variable]

The command for configuring the proof process to gain synchronization.

This command is sent before Proof General's synchronization mechanism is engaged, to allow customization inside the process to help gain syncrhonization (e.g. engaging special markup).

It is better to configure the proof assistant for this purpose via command line options if possible, in which case this variable does not need to be set.

See also '`proof-shell-init-cmd`'.

`proof-shell-init-cmd` [Variable]
The command(s) for initially configuring the proof process.
This command is sent to the process as soon as synchronization is gained (when an annotated prompt is first recognized). It can be used to configure the proof assistant in some way, or print a welcome message (since output before the first prompt is discarded).

See also '`proof-shell-pre-sync-init-cmd`'.

`proof-shell-restart-cmd` [Variable]
A command for re-initialising the proof process.

`proof-shell-quit-cmd` [Variable]
A command to quit the proof process. If nil, send EOF instead.

`proof-shell-cd-cmd` [Variable]
Command to the proof assistant to change the working directory.
The format character '`%s`' is replaced with the directory, and the escape sequences in '`proof-shell-filename-escapes`' are applied to the filename.

This setting is used to define the function '`proof-cd`' which changes to the value of (`default-directory`) for script buffers. For files, the value of (`default-directory`) is simply the directory the file resides in.

NB: By default, '`proof-cd`' is called from '`proof-activate-scripting-hook`', so that the prover switches to the directory of a proof script every time scripting begins.

`proof-shell-start-silent-cmd` [Variable]
Command to turn prover goals output off when sending many script commands.
If non-nil, Proof General will automatically issue this command to help speed up processing of long proof scripts. See also '`proof-shell-stop-silent-cmd`'. NB: terminator not added to command.

`proof-shell-stop-silent-cmd` [Variable]
Command to turn prover output on.
If non-nil, Proof General will automatically issue this command to help speed up processing of long proof scripts. See also '`proof-shell-start-silent-cmd`'. NB: Terminator not added to command.

`proof-shell-silent-threshold` [Variable]
Number of waiting commands in the proof queue needed to trigger silent mode.
Default is 2, but you can raise this in case switching silent mode on or off is particularly expensive (or make it ridiculously large to disable silent mode altogether).

See Chapter 8 [Handling Multiple Files], page 35, for more details about the final two settings in this group,

`proof-shell-inform-file-processed-cmd` [Variable]
Command to the proof assistant to tell it that a file has been processed.
The format character '`%s`' is replaced by a complete filename for a script file which has been fully processed interactively with Proof General. See '`proof-format-filename`' for other possibilities to process the filename.

This setting used to interface with the proof assistant's internal management of multiple files, so the proof assistant is kept aware of which files have been processed. Specifically, when

scripting is deactivated in a completed buffer, it is added to Proof General's list of processed files, and the prover is told about it by issuing this command.

If this is set to nil, no command is issued.

See also: '`proof-shell-inform-file-retracted-cmd`', '`proof-shell-process-file`', '`proof-shell-compute-new-files-list`'.

`proof-shell-inform-file-retracted-cmd` [Variable]

Command to the proof assistant to tell it that a file has been retracted.
The format character '`%s`' is replaced by a complete filename for a script file which Proof General wants the prover to consider as not completely processed. See '`proof-format-filename`' for other possibilities to process the filename.

This is used to interface with the proof assistant's internal management of multiple files, so the proof assistant is kept aware of which files have been processed. Specifically, when scripting is activated, the file is removed from Proof General's list of processed files, and the prover is told about it by issuing this command. The action may cause the prover in turn to suggest to Proof General that files depending on this one are also unlocked.

If this is set to nil, no command is issued.

It is also possible to set this value to a function which will be invoked on the name of the retracted file, and should remove the ancestor files from '`proof-included-files-list`' by some other calculation.

See also: '`proof-shell-inform-file-processed-cmd`', '`proof-shell-process-file`', '`proof-shell-compute-new-files-list`'.

4.2 Script input to the shell

Generally, commands from the proof script are sent verbatim to the proof process running in the proof shell. For historical reasons, carriage returns are stripped by default. You can set `proof-shell-strip-crs-from-input` to adjust that. For more sophisticated pre-processing of the sent string, you may like to set `proof-shell-insert-hook`.

`proof-shell-strip-crs-from-input` [Variable]

If non-nil, replace carriage returns in every input with spaces.
This is enabled by default: it is appropriate for many systems based on human input, because several CR's can result in several prompts, which may mess up the display (or even worse, the synchronization).

If the prover can be set to output only one prompt for every chunk of input, then newlines can be retained in the input.

`proof-shell-insert-hook` [Variable]

Hook run by '`proof-shell-insert`' before inserting a command.
Can be used to configure the proof assistant to the interface in various ways – for example, to observe or alter the commands sent to the prover, or to sneak in extra commands to configure the prover.

This hook is called inside a '`save-excursion`' with the '`proof-shell-buffer`' current, just before inserting and sending the text in the variable '`string`'. The hook can massage '`string`' or insert additional text directly into the '`proof-shell-buffer`'. Before sending '`string`', it will be stripped of carriage returns.

Additionally, the hook can examine the variable '`action`'. It will be a symbol, set to the callback command which is executed in the proof shell filter once '`string`' has been processed. The '`action`' variable suggests what class of command is about to be inserted, the first two are normally the ones of interest:

```
'proof-done-advancing       A "forward" scripting command
```

```
'proof-done-retracting      A "backward" scripting command
'proof-done-invisible       A non-scripting command
'proof-shell-set-silent     Indicates prover output has been surpressed
'proof-shell-clear-silent   Indicates prover output has been restored
'init-cmd                   Early initialization command sent to prover
```

Caveats: You should be very careful about setting this hook. Proof General relies on a careful synchronization with the process between inputs and outputs. It expects to see a prompt for each input it sends from the queue. If you add extra input here and it causes more prompts than expected, things will break! Extending the variable '`string`' may be safer than inserting text directly, since it is stripped of carriage returns before being sent.

Example uses: Lego used this hook for setting the pretty printer width if the window width has changed; Plastic used it to remove literate-style markup from '`string`'.

See also '`proof-script-preprocess`' which can munge text when it is added to the queue of commands.

4.3 Settings for matching various output from proof process

These settings control the way Proof General reacts to process output. The single most important setting is `proof-shell-annotated-prompt-regexp`, which **must** be set as part of the prover configuraton. This is used to configure the communication with the prover process.

`pg-subterm-first-special-char` [Variable]

First special character.
Codes above this character can have special meaning to Proof General, and are stripped from the prover's output strings. Leave unset if no special characters are being used.

`proof-shell-annotated-prompt-regexp` [Variable]

Regexp matching a (possibly annotated) prompt pattern.

this IS THE *most important setting* TO *configure*!!

Output is grabbed between pairs of lines matching this regexp, and the appearance of this regexp is used by Proof General to recognize when the prover has finished processing a command.

To help speed up matching you may be able to annotate the proof assistant prompt with a special character not appearing in ordinary output, which should appear in this regexp.

`proof-shell-error-regexp` [Variable]

Regexp matching an error report from the proof assistant.

We assume that an error message corresponds to a failure in the last proof command executed. So don't match mere warning messages with this regexp. Moreover, an error message should **not** be matched as an eager annotation (see '`proof-shell-eager-annotation-start`') otherwise it will be lost.

Error messages are considered to begin from '`proof-shell-error-regexp`' and continue until the next prompt. The variable '`proof-shell-truncate-before-error`' controls whether text before the error message is displayed.

The engine matches interrupts before errors, see '`proof-shell-interrupt-regexp`'.

It is safe to leave this variable unset (as nil).

`proof-shell-interrupt-regexp` [Variable]

Regexp matching output indicating the assistant was interrupted.
We assume that an interrupt message corresponds to a failure in the last proof command executed. So don't match mere warning messages with this regexp. Moreover, an interrupt message

should not be matched as an eager annotation (see ‘`proof-shell-eager-annotation-start`’) otherwise it will be lost.

The engine matches interrupts before errors, see ‘`proof-shell-error-regexp`’.

It is safe to leave this variable unset (as nil).

`proof-shell-truncate-before-error` [Variable]

Non-nil means truncate output that appears before error messages.
If nil, the whole output that the prover generated before the last error message will be shown.

NB: the default setting for this is t to be compatible with behaviour in Proof General before version 3.4. The more obvious setting for new instances is probably nil.

Interrupt messages are treated in the same way. See ‘`proof-shell-error-regexp`’ and ‘`proof-shell-interrupt-regexp`’.

`proof-shell-proof-completed-regexp` [Variable]

Regexp matching output indicating a finished proof.

When output which matches this regexp is seen, we clear the goals buffer in case this is not also marked up as a ‘`goals`’ type of message.

We also enable the QED function (save a proof) and we may automatically close off the proof region if another goal appears before a save command, depending on whether the prover supports nested proofs or not.

`proof-shell-start-goals-regexp` [Variable]

Regexp matching the start of the proof state output.
This is an important setting. Output between ‘`proof-shell-start-goals-regexp`’ and ‘`proof-shell-end-goals-regexp`’ will be pasted into the goals buffer and possibly analysed further for proof-by-pointing markup. If it is left as nil, the goals buffer will not be used.

The goals display starts at the beginning of the match on this regexp, unless it has a match group, in which case it starts at (`match-end` 1).

`proof-shell-end-goals-regexp` [Variable]

Regexp matching the end of the proof state output, or nil.
This allows a shorter form of the proof state output to be displayed, in case several messages are combined in a command output.

The portion treated as the goals output will be that between the match on ‘`proof-shell-start-goals-regexp`’ (which see) and the start of the match on ‘`proof-shell-end-goals-regexp`’.

If nil, use the whole of the output from the match on ‘`proof-shell-start-goals-regexp`’ up to the next prompt.

`proof-shell-assumption-regexp` [Variable]

A regular expression matching the name of assumptions.

At the moment, this setting is not used in the generic Proof General.

Future use may provide a generic implementation for ‘`pg-topterm-goalhyplit-fn`’, used to help parse the goals buffer to annotate it for proof by pointing.

4.4 Settings for matching urgent messages from proof process

Among the various dialogue messages that the proof assistant outputs during proof, Proof General can consider certain messages to be "urgent". When processing many lines of a proof, Proof General will normally supress the output, waiting until the final message appears before displaying anything to the user. Urgent messages escape this: typically they include messages

that the prover wants the user to notice, for example, perhaps, file loading messages, timing statistics or dedicated tracing messages which can be sent to the `*trace*` buffer.

So that Proof General notices, these urgent messages should be marked-up with "eager" annotations.

`proof-shell-eager-annotation-start` [Variable]

Eager annotation field start. A regular expression or nil.
An "eager annotation indicates" to Proof General that some following output should be displayed (or processed) immediately and not accumulated for parsing later. Note that this affects processing of output which is ordinarily accumulated: output which appears before the eager annotation start will be discarded.

The start/end annotations can be used to hilight the output, but are stripped from display of the message in the minibuffer.

It is useful to recognize (starts of) warnings or file-reading messages with this regexp. You must also recognize any special messages from the prover to PG with this regexp (e.g. '`proof-shell-clear-goals-regexp`', '`proof-shell-retract-files-regexp`', etc.)

See also '`proof-shell-eager-annotation-start-length`', '`proof-shell-eager-annotation-end`'.

Set to nil to disable this feature.

`proof-shell-eager-annotation-start-length` [Variable]

Maximum length of an eager annotation start.
Must be set to the maximum length of the text that may match '`proof-shell-eager-annotation-start`' (at least 1). If this value is too low, eager annotations may be lost!

This value is used internally by Proof General to optimize the process filter to avoid unnecessary searching.

`proof-shell-eager-annotation-end` [Variable]

Eager annotation field end. A regular expression or nil.
An eager annotation indicates to Emacs that some following output should be displayed or processed immediately.

See also '`proof-shell-eager-annotation-start`'.

It is nice to recognize (ends of) warnings or file-reading messages with this regexp. You must also recognize (ends of) any special messages from the prover to PG with this regexp (e.g. '`proof-shell-clear-goals-regexp`', '`proof-shell-retract-files-regexp`', etc.)

The default value is `"\n"` to match up to the end of the line.

The default action for urgent messages is to display them in the response buffer, highlighted. But we also allow for some control messages, issued from the proof assistant to Proof General and not intended for the user to see. These are recognized in the same way as urgent messages (marked with eager annotations), so they will be acted on as soon as they are issued by the prover.

`proof-shell-clear-response-regexp` [Variable]

Regexp matching output telling Proof General to clear the response buffer.

More precisely, this should match a string which is bounded by matches on '`proof-shell-eager-annotation-start`' and '`proof-shell-eager-annotation-end`'.

This feature is useful to give the prover more control over what output is shown to the user. Set to nil to disable.

`proof-shell-clear-goals-regexp` [Variable]
Regexp matching output telling Proof General to clear the goals buffer.

More precisely, this should match a string which is bounded by matches on '`proof-shell-eager-annotation-start`' and '`proof-shell-eager-annotation-end`'.

This feature is useful to give the prover more control over what output is shown to the user. Set to nil to disable.

`proof-shell-interactive-prompt-regexp` [Variable]
Matches output from the prover which indicates an interactive prompt.
If we match this, we suppose that the prover has switched to an interactive diagnostic mode which requires direct interaction with the shell rather than via script management. In this case, the shell buffer will be displayed and the user left to their own devices.

Note: this should match a string which is bounded by matches on '`proof-shell-eager-annotation-start`' and '`proof-shell-eager-annotation-end`'.

`proof-shell-trace-output-regexp` [Variable]
Matches tracing output which should be displayed in trace buffer.
Each line which matches this regexp but would otherwise be treated as an ordinary response, is sent to the trace buffer instead of the response buffer.

This is intended for unusual debugging output from the prover, rather than ordinary output from final proofs.

This should match a string which is bounded by matches on '`proof-shell-eager-annotation-start`' and '`proof-shell-eager-annotation-end`'.

Set to nil to disable.

`proof-shell-theorem-dependency-list-regexp` [Variable]
Matches output telling Proof General about dependencies.
This is to allow navigation and display of dependency information. The output from the prover should be a message with the form

dependencies `OF X Y Z ARE A B C`

with X Y Z, A B C separated by whitespace or somehow else (see '`proof-shell-theorem-dependency-list-split`'. This variable should be set to a regexp to match the overall message (which should be an urgent message), with two sub-matches for X Y Z and A B C.

This is an experimental feature, currently work-in-progress.

Two important control messages are recognized by `proof-shell-process-file` and `proof-shell-retract-files-regexp`, used for synchronizing Proof General with a file loading mechanism built into the proof assistant. See Chapter 8 [Handling Multiple Files], page 35, for more details about how to use the final four settings described here.

`proof-shell-process-file` [Variable]
A pair (*regexp* . *function*) to match a processed file name.

If *regexp* matches output, then the function *function* is invoked. It must return the name of a script file (with complete path) that the system has successfully processed. In practice, *function* is likely to inspect the match data. If it returns the empty string, the file name of the scripting buffer is used instead. If it returns nil, no action is taken.

More precisely, *regexp* should match a string which is bounded by matches on '`proof-shell-eager-annotation-start`' and '`proof-shell-eager-annotation-end`'.

Care has to be taken in case the prover only reports on compiled versions of files it is processing. In this case, *function* needs to reconstruct the corresponding script file name. The new (true) file name is added to the front of '`proof-included-files-list`'.

`proof-shell-retract-files-regexp` [Variable]

Matches a message that the prover has retracted a file.

More precisely, this should match a string which is bounded by matches on '`proof-shell-eager-annotation-start`' and '`proof-shell-eager-annotation-end`'.

At this stage, Proof General's view of the processed files is out of date and needs to be updated with the help of the function '`proof-shell-compute-new-files-list`'.

`proof-shell-compute-new-files-list` [Variable]

Function to update '`proof-included-files list`'.

It needs to return an up-to-date list of all processed files. The result will be stored in '`proof-included-files-list`'.

This function is called when '`proof-shell-retract-files-regexp`' has been matched in the prover output.

In practice, this function is likely to inspect the previous (global) variable '`proof-included-files-list`' and the match data triggered by '`proof-shell-retract-files-regexp`'.

`proof-cannot-reopen-processed-files` [Variable]

Non-nil if the prover allows re-opening of already processed files.

If the user has used Proof General to process a file incrementally, then PG will retain the spans recording undo history in the buffer corresponding to that file (provided it remains visited in Emacs).

If the prover allows, it will be possible to undo to a position within this file. If the prover does **not** allow this, this variable should be set non-nil, so that when a completed file is activated for scripting (to do undo operations), the whole history is discarded.

4.5 Hooks and other settings

`proof-shell-filename-escapes` [Variable]

A list of escapes that are applied to %s for filenames.

A list of cons cells, car of which is string to be replaced by the cdr. For example, when directories are sent to Isabelle, HOL, and Coq, they appear inside ML strings and the backslash character and quote characters must be escaped. The setting

```
'(("\\" . "\\\\")
  ("\"" . "\\\""))
```

achieves this.

This setting is used inside the function '`proof-format-filename`'.

`proof-shell-process-connection-type` [Variable]

The value of '`process-connection-type`' for the proof shell.

Set non-nil for ptys, nil for pipes.

note: In Emacs >= 24 (checked for 24 and 25.0.50.1), t is not a good choice: input is cut after *4095* chars, which hangs pg.

`proof-shell-handle-error-or-interrupt-hook` [Variable]

Run after an error or interrupt has been reported in the response buffer.

Hook functions may inspect '`proof-shell-last-output-kind`' to determine whether the cause was an error or interrupt. Possible values for this hook include:

'`proof-goto-end-of-locked-on-error-if-pos-not-visible-in-window`'
'`proof-goto-end-of-locked-if-pos-not-visible-in-window`'

which move the cursor in the scripting buffer on an error or error/interrupt.

Remark: This hook is called from shell buffer. If you want to do something in scripting buffer, '`save-excursion`' and/or '`set-buffer`'.

`proof-shell-pre-interrupt-hook` [Variable]

Run immediately after '`comint-interrupt-subjob`' is called.
This hook is added to allow customization for systems that query the user before returning to the top level.

`proof-shell-handle-output-system-specific` [Variable]

Set this variable to handle system specific output.
Errors and interrupts are recognised in the function '`proof-shell-handle-immediate-output`'. Later output is handled by '`proof-shell-handle-delayed-output`', which displays messages to the user in **goals** and **response** buffers.

This hook can run between the two stages to take some effect.

It should be a function which is passed (cmd string) as arguments, where '`cmd`' is a string containing the currently processed command and '`string`' is the response from the proof system. If action is taken and goals/response display should be prevented, the function should update the variable '`proof-shell-last-output-kind`' to some non-nil symbol.

The symbol will be compared against standard ones, see documentation of '`proof-shell-last-output-kind`'. A suggested canonical non-standard symbol is '`'systemspecific`.

5 Goals Buffer Settings

The goals buffer settings allow configuration of Proof General for proof by pointing or similar features. See the Proof General documentation web page (`https://proofgeneral.github.io/doc`) for a link to the technical report ECS-LFCS-97-368 which hints at how to use these settings.

`pg-goals-change-goal` [Variable]
Command to change to the goal '`%s`'.

`pbp-goal-command` [Variable]
Command sent when '`pg-goals-button-action`' is requested on a goal.

`pbp-hyp-command` [Variable]
Command sent when '`pg-goals-button-action`' is requested on an assumption.

`pg-goals-error-regexp` [Variable]
Regexp indicating that the proof process has identified an error.

`proof-shell-result-start` [Variable]
Regexp matching start of an output from the prover after pbp commands.
In particular, after a '`pbp-goal-command`' or a '`pbp-hyp-command`'.

`proof-shell-result-end` [Variable]
Regexp matching end of output from the prover after pbp commands.
In particular, after a '`pbp-goal-command`' or a '`pbp-hyp-command`'.

`pg-subterm-start-char` [Variable]
Opening special character for subterm markup.
Subsequent special characters with values **below** '`pg-subterm-first-special-char`' are assumed to be subterm position indicators. Annotations should be finished with '`pg-subterm-sep-char`'; the end of the concrete syntax is indicated by '`pg-subterm-end-char`'.

If '`pg-subterm-start-char`' is nil, subterm markup is disabled.

`pg-subterm-sep-char` [Variable]
Finishing special for a subterm markup.
See doc of '`pg-subterm-start-char`'.

`pg-topterm-regexp` [Variable]
Annotation regexp that indicates the beginning of a `"top"` element.
A `"top"` element may be a sub-goal to be proved or a named hypothesis, for example. It could also be a literal command to insert and send back to the prover.

The function '`pg-topterm-goalhyplit-fn`' examines text following this special character, to determine what kind of top element it is.

This setting is also used to see if proof-by-pointing features are configured. If it is unset, some of the code for parsing the prover output is disabled.

`pg-subterm-end-char` [Variable]
Closing special character for subterm markup.
See '`pg-subterm-start-char`'.

6 Splash Screen Settings

The splash screen can be configured, in a rather limited way.

`proof-splash-time` [Variable]
Minimum number of seconds to display splash screen for.
The splash screen may be displayed for a wee while longer than this, depending on how long it takes the machine to initialise Proof General.

`proof-splash-contents` [Variable]
Evaluated to configure splash screen displayed when entering Proof General.
A list of the screen contents. If an element is a string or an image specifier, it is displayed centred on the window on its own line. If it is nil, a new line is inserted.

7 Global Constants

The settings here are internal constants used by Proof General. You don't need to configure these for your proof assistant unless you want to modify or extend the defaults.

`proof-general-name` [Variable]
Proof General name used internally and in menu titles.

`proof-general-home-page` [User Option]
Web address for Proof General.

The default value is `"https://proofgeneral.github.io"`.

`proof-universal-keys` [Variable]
List of key bindings made for all proof general buffers.
Elements of the list are tuples '`(k . f)`' where '`k`' is a key binding (vector) and '`f`' the designated function.

8 Handling Multiple Files

Large proof developments are typically spread across multiple files. Many provers support such developments by keeping track of dependencies and automatically processing scripts. Proof General supports this mechanism. The user's point of view is considered in the user manual. Here, we describe the more technical nitty gritty. This is what you need to know when you customise another proof assistant to work with Proof General.

Documentation for the configuration settings mentioned here appears in the previous sections, this section is intended to help explain the use of those settings.

Proof General maintains a list `proof-included-files-list` of files which it thinks have been processed by the proof assistant. When a file which is on this list is visited in Emacs, it will be coloured entirely blue to indicate that it has been processed. No editing of the file will be allowed (unless `proof-strict-read-only` allows it).

`proof-included-files-list` [Variable]

List of files currently included in proof process.
This list contains files in canonical truename format (see '`file-truename`').

Whenever a new file is being processed, it gets added to this list via the '`proof-shell-process-file`' configuration settings. When the prover retracts a file, this list is resynchronised via the '`proof-shell-retract-files-regexp`' and '`proof-shell-compute-new-files-list`' configuration settings.

Only files which have been **fully** processed should be included here. Proof General itself will automatically add the filenames of a script buffer which has been completely read when scripting is deactivated. It will automatically remove the filename of a script buffer which is completely unread when scripting is deactivated.

NB: Currently there is no generic provision for removing files which are only partly read-in due to an error, so ideally the proof assistant should only output a processed message when a file has been successfully read.

The way that `proof-included-files-list` is maintained is the key to multiple file management. Ideally you should not set this variable directly, but instead use (some of) the various configuration settings that enable functionality inside Proof General for managing `proof-included-files-list` (see below if the configuration setting do not suffice).

There is a range of strategies for managing multiple files. Ideally, file dependencies should be managed by the proof assistant. Proof General will use the prover's low-level commands to process a whole file and its requirements non-interactively, without going through script management. So that the user knows which files have been processed, the proof assistant should issue messages which Proof General can recognize ("file `foo` has been processed") — see `proof-shell-process-file`. When the user wants to edit a file which has been processed, the file must be retracted (unlocked). The proof assistant should provide a command corresponding to this action, which undoes a given file and all its dependencies. As each file is undone, a message should be issued which Proof General can recognize ("file `foo` has been undone") – see `proof-shell-retract-files-regexp`. (The function `proof-shell-compute-new-files-list` should be set to calculate the new value for `proof-included-files-list` after a retract message has been seen).

As well as this communication from the assistant to Proof General about processed or retracted files, Proof General can communicate the other way: it will tell the proof assistant when it has processed or retracted a file via script management. This is because during script management, the proof assistant may not be aware that it is actually dealing with a file of proof commands (rather than just terminal input).

Proof General will provide this information in two special instances. First, when scripting is turned off in a file that has been completely processed, Proof General will tell the proof assistant using `proof-shell-inform-file-processed-cmd`. Second, when scripting is turned on in a file which is completely processed, Proof General will tell the proof assistant to reconsider: the file should not be considered completely processed yet. This uses the setting `proof-shell-inform-file-retracted-cmd`. This second, retracting, case might lead to a series of messages from the prover telling Proof General to unlock files which depend on the present one, again via `proof-shell-retract-files-regexp`.

The special case for retracting is the primary file the user wishes to edit: this is automatically removed from `proof-included-files-list`, but it depends on the proof assistant whether or not it is possible to revert to a partially processed version of the file (or `"undo into"` it). This is the reason for the setting `proof-cannot-reopen-processed-files`. If this is non-nil, any attempt to undo a fully processed file will unlock the entire file (whether or not Proof General itself has history information for the file).

What we have described so far is the ideal case, but it may require some support from the proof assistant to set up (for example, if file-level undo is not normally supported, or the messages during file processing are not suitable). Moreover, some proof assistants may not have file handling with dependencies, or may have a particularly simple case of a linear context: each file depends on all the ones processed before it. Proof General allows you a shortcut to get automatic management of multiple files in these cases by setting the flag `proof-auto-multiple-files`. This setting is probably an approximation to the right thing for any proof assistant. More files than necessary will be retracted if the prover has a tree-like file dependency rather than a linear one.

Finally, we should mention how Proof General recognizes file processing messages from the proof assistant. Proof General considers *output* delimited by the the two regular expressions `proof-shell-eager-annotation-start` and `proof-shell-eager-annotation-end` as being important. It displays the *output* in the Response buffer and analyses the contents further. Among other important messages characterised by these regular expressions (warnings, errors, or information), the prover can tell the interface whenever it processes or retracts a file.

To summarize, the settings for multiple file management that may be customized are as follows. To recognize file-processing, `proof-shell-process-file`. To recognize messages about file undoing, `proof-shell-retract-files-regexp` and `proof-shell-compute-new-files-list`. See Section 4.4 [Settings for matching urgent messages from proof process], page 23. To tell the prover about files handled with script management, use `proof-shell-inform-file-processed-cmd` and `proof-shell-inform-file-retracted-cmd`. See Section 4.1 [Proof shell commands], page 19. If your prover does not allow re-opening of closed files, set `proof-cannot-reopen-processed-files` to `t`. Finally, set the flag `proof-auto-multiple-files` for a automatic approximation to multiple file handling. See Chapter 3 [Proof Script Settings], page 9.

Internally Proof General uses `proof-register-possibly-new-processed-file` to add a file to `proof-included-files-list` and to possibly inform the prover about this fact, See Section 14.5 [Proof script mode], page 56. The function `proof-shell-process-urgent-message-retract` is responsible for taking (possibly several) files off `proof-included-files-list`. It relies on `proof-shell-compute-new-files-list` (see Section 4.4 [Settings for matching urgent messages from proof process], page 23) to compute the new value of `proof-included-files-list` and then calls `proof-restart-buffers` on all those buffers that have been taken off from `proof-included-files-list`, See Section 14.5 [Proof script mode], page 56.

9 Configuring Editing Syntax

Emacs has some standard settings which configure the syntax of major modes. The main setting is the *syntax table*, which determines the syntax of programming elements such as strings, comments, and parentheses. To configure the syntax table, you can either write calls to `modify-syntax-entry` in your mode functions, or set the following variables to contain the tables for each mode. (The main mode to be concerned about is of course the proof script, where user editing takes place).

`proof-script-syntax-table-entries` [Variable]

List of syntax table entries for proof script mode.
A flat list of the form

```
(char syncode char syncode ...)
```

See doc of '`modify-syntax-entry`' for details of characters and syntax codes.

At present this is used only by the '`proof-easy-config`' macro.

`proof-shell-syntax-table-entries` [Variable]

List of syntax table entries for proof script mode.
A flat list of the form

```
(char syncode char syncode ...)
```

See doc of '`modify-syntax-entry`' for details of characters and syntax codes.

At present this is used only by the '`proof-easy-config`' macro.

Some additional useful settings are:

`comment-quote-nested` [Variable]

Non-nil if nested comments should be quoted. This should be locally set by each major mode if needed. The default setting is non-nil: modes which allow nested comments may set this to nil.

`outline-regexp` [Variable]

Regular expression to match the beginning of a heading. Any line whose beginning matches this regexp is considered to start a heading.

`outline-heading-end-regexp` [Variable]

Regular expression to match the beginning of a heading. Any line whose beginning matches this regexp is considered to start a heading.

10 Configuring Font Lock

Support for Font Lock in Proof General is described in the user manual (see the *Syntax highlighting* section). To configure Font Lock for a new proof assistant, you need to set the variable `font-lock-keywords` in each of the mode functions you want highlighting for. Proof General will automatically install these settings, and use font lock minor mode (for syntax highlighting as you type) in script buffers.

To understand its format, check the documentation of `font-lock-keywords` inside Emacs.

Instead of setting `font-lock-keywords` in each mode function, you can use the following four variables to make the settings in place. This is particularly useful if use the easy configuration mechanism for Proof General, see Section 1.2 [Demonstration instance and easy configuration], page 4.

`proof-script-font-lock-keywords` [Variable]
Value of '`font-lock-keywords`' used to fontify proof scripts.
The proof script mode should set this before calling '`proof-config-done`'. Used also by '`proof-easy-config`' mechanism. See also '`proof-goals-font-lock-keywords`' and '`proof-response-font-lock-keywords`'.

`proof-goals-font-lock-keywords` [Variable]
Value of '`font-lock-keywords`' used to fontify the goals output.
The goals shell mode should set this before calling '`proof-goals-config-done`'. Used also by '`proof-easy-config`' mechanism. See also '`proof-script-font-lock-keywords`' and '`proof-response-font-lock-keywords`'.

`proof-response-font-lock-keywords` [Variable]
Value of '`font-lock-keywords`' used to fontify the response output.
The response mode should set this before calling '`proof-response-config-done`'. Used also by '`proof-easy-config`' mechanism. See also '`proof-script-font-lock-keywords`' and '`proof-goals-font-lock-keywords`'.

Proof General provides a special function, `proof-zap-commas`, for tweaking the font lock behaviour of provers which have declarations of the form `x,y,z:Ty`. This function removes highlighting on the commas, and can be added as the last element of `font-lock-keywords`. Further manipulation of font lock behaviour can be achieved via two hook functions which are run before and after fontifying the output buffers.

`proof-zap-commas` *limit* [Function]
Remove the face of all '`,`' from point to *limit*.
Meant to be used from '`font-lock-keywords`' as a way to unfontify commas in declarations and definitions. Useful for provers which have declarations of the form x,y,z:Ty All that can be said for it is that the previous ways of doing this were even more bogus....

`pg-before-fontify-output-hook` [Variable]
This hook is called before fontifying a region in an output buffer.
A function on this hook can alter the region of the buffer within the current restriction, and must return the final value of (`point-max`). [This hook is presently only used by phox-symlock].

`pg-after-fontify-output-hook` [Variable]
This hook is called before fonfitying a region in an output buffer.
[This hook is presently only used by Isabelle].

11 Configuring Tokens

Unicode Tokens is basically an overly complicated way of configuring font-lock, along with some helpful menus. The font lock configuration makes use of recent Emacs features, particularly including `compose-region` which allows the presentation of the buffer be different from the underlying buffer contents. Compared with the X-Symbol package used previously by Proof General, this has the huge advantage of not requiring the underlying text to be changed to display symbols.

Usage of the Unicode Tokens package is described in the Proof General user manual, See Info file `ProofGeneral`, node '`Unicode support`'.

`proof-tokens-activate-command` [Variable]

Command to activate token input/output for prover.
If non-nil, this command is sent to the proof assistant when Unicode Tokens support is activated.

`proof-tokens-deactivate-command` [Variable]

Command to deactivate token input/output for prover.
If non-nil, this command is sent to the proof assistant when Unicode Tokens support is deactivated.

We expect tokens to be used uniformly, so that along with each script mode buffer, the response buffer and goals buffer also invoke Tokens to display special characters in the same token language. This happens automatically. If you want additional modes to use Tokens with the token language for your proof assistant, you can set `proof-tokens-extra-modes`.

`proof-tokens-extra-modes` [Variable]

List of additional mode names to use with Proof General tokens.
These modes will have Tokens enabled for the proof assistant token language, in addition to the four modes for Proof General (script, shell, response, pbp).

Set this variable if you want additional modes to also display tokens (for example, editing documentation or source code files).

12 Configuring Proof-Tree Visualization

The proof-tree visualization feature was written with the idea of supporting Coq as well as other proof assistants. Nevertheless, supporting proof-tree visualization for a second proof assistant will almost certainly require changes in the generic Elisp code in `generic/proof-tree.el` as well as in the Prooftree program.

12.1 A layered set of proof trees

Prooftree can actually display more than one proof tree per proof. This is necessary to support the `Grab Existential Variables` command in Coq. When the main goal has been proved, this command turns all open existential variables into new proof obligations. All these new proof obligations become root nodes for their own proof trees. When they all have been proved one can again grab the open existential variables...

For each proof, Prooftree can therefore display several layers, where each layer can contain several (graphically) independent proof trees. The first layer contains one tree for the original proof goal. The second layer contains proof trees for goals that have been added to the proof after the first proof tree was completed. And so on.

To organize the layers, Prooftree must identify those proof commands that add new goals to a proof.

`proof-tree-new-layer-command-regexp` [Variable]

Regexp to match proof commands that add new goals to a proof.

This regexp must match the command that turns the proof assistant into prover mode, which adds the initial goal to the proof. It must further match commands that add additional goals after all previous goals have been proved.

12.2 Prerequisites

Proof-tree visualization requires certain support from the proof assistant. Patching the proof assistant is therefore the first step of adding support for proof-tree visualization. The following features are needed.

Unique goal identification

: The proof assistant must assign and output a unique string for each goal. For Coq the internal `evar` index number is used, which is printed for each goal in the form `(ID XXX)` when Coq is started with the option `-emacs`.

  The unique goal identification is needed to distinguish newly spawned subgoals from older open subgoals and to mark the current goal in the proof-tree display.

Indication of newly generated subgoals

: A proof command that spawns additional subgoals must somehow indicate the goal ID's of these new subgoals. Otherwise the proof-tree display will not be able to reconstruct the proof-tree structure.

  For Coq the newly spawned subgoals appear always in the list of additional subgoals below the current goal. Note, that it is not required to mark the newly spawned subgoals. They may appear in a mixed list with older open subgoals. Note further, that it is not required that always the complete set of all open subgoals is printed (which is indeed not the case after of `Focus` command in Coq). It is only required that the goal ID's of all newly spawned subgoals is somehow printed.

State number for undo

: There must be a state number that is strictly increasing when asserting proof commands and that is reset to the appropriate number after retracting some proof commands.

For Coq the state number in the extended prompt (visible only with option `-emacs`) is used.

Information about existential variables
: Existential variables are placeholders that might or must be instantiated later in the proof. Prooftree supports existential variables with three features. Firstly, it can update goals when existential variables get instantiated. Secondly, it can mark the proof commands that introduced or instantiated existential variables and, thirdly, it can display and track dependencies between existential variables.

  For the first feature, the proof assistant must list the currently instantiated existential variables for every goal. For the second feature it must additionally list the not instantiated existential variables. Finally, for the third feature, it must display the dependencies for instantiated existential variables.

  For Coq, all necessary information is provided in the existential evar line, that is printed with the `-emacs` switch.

12.3 Proof-Tree Display Internals

This section gives some information about the inner structure of the code that realizes the proof-tree display. The idea here is that this section provides the background information to make the documentation of the customizable variables of the proof-tree Elisp code easy to understand.

12.3.1 Organization of the Code

The proof-tree display is realized by Proof General in cooperation with the external Prooftree program. The latter is a GTK application in OCaml. Both, the Elisp code in Proof General and the Prooftree OCaml code is divided into a generic and a proof assistant specific part.

The generic Elisp code lives in `generic/proof-tree.el`. As usual in Proof General, it contains various customizable variables, which the proof assistant specific code must set. Most of these variables contain regular expressions, but there are also some that hold functions. The Coq specific code for the proof-tree display is distributed in a few chunks over `coq/coq.el`.

The main task of the Elisp code is to extract goals, undo events and information about existential variables from the proof-assistant output and to send all this data to Prooftree. The Elisp code does also determine if additional output must be requested from the proof assistant. In that case it adds appropriate commands to `proof-action-list`, see Section 14.5 [Proof script mode], page 56. These additional commands are flagged with `proof-tree-show-subgoal`, `no-goals-display` and `no-response-display`. The flag `proof-tree-show-subgoal` ensures that a number of internal functions ignore these additional commands. The other two flags ensure that their output is neither displayed in the goals nor the response buffer.

For the decision about which goals must be sent to Prooftree, the Elisp code maintains the following two state variables.

`proof-tree-sequent-hash` [Variable]
: Hash table to remember sequent ID's.
  Needed because some proof assistants do not distinguish between new subgoals, which have been created by the last proof command, and older, currently unfocussed subgoals. If Proof General meets a goal, it is treated as new subgoal if it is not in this hash yet.

  The hash is mostly used as a set of sequent ID's. However, for undo operations it is necessary to delete all those sequents from the hash that have been created in a state later than the undo state. For this purpose this hash maps sequent ID's to the state number in which the sequent has been created.

  The hash table is initialized in '`proof-tree-start-process`'.

`proof-tree-existentials-alist` [Variable]
Alist mapping existential variables to sequent ID's.
Used to remember which goals need a refresh when an existential variable gets instantiated. To support undo commands the old contents of this list must be stored in '`proof-tree-existentials-alist-history`'. To ensure undo is properly working, this variable should only be changed by using '`proof-tree-delete-existential-assoc`', '`proof-tree-add-existential-assoc`' or '`proof-tree-clear-existentials`'.

When retracting these two variables must be set to their previous state. For `proof-tree-sequent-hash` this is done with the state numbers that are stored in the hash. For `proof-tree-existentials-alist` a separate alist stores previous states.

`proof-tree-existentials-alist-history` [Variable]
Alist mapping state numbers to old values of '`proof-tree-existentials-alist`'.
Needed for undo.

In Prooftree the separation between generic and proof-assistant specific code is less obvious. The Coq specific code is in the file `coq.ml`. All the remaining code is generic.

Prooftree opens for each proof a separate window. It reconstructs the proof tree and orders the existential variables in a dependency hierarchy. It stores a complete history of previous states to support arbitrary undo operations. Under normal circumstances one starts just one Prooftree process that keeps running for the remainder of the Proof General session, regardless of how many proof-tree windows are displayed.

A fair amount of the Prooftree code is documented with `ocamldoc` documentation comments. With `make doc` they can be converted into a set of html pages in the `doc` subdirectory.

12.3.2 Communication

Prooftree is a standard Emacs subprocess that reads goals and other proof status messages from its standard input. The communication between Proof General and Prooftree is almost one way only. Proof General sends proof status messages to Prooftree, from which Prooftree reconstructs the current proof status and the complete proof tree. Prooftree never requests additional information from Proof General.

There are only a few messages that Prooftree sends to Proof General. These messages communicate user requests to Proof General, for instance, when the user selects the undo menu item, or when he closes the Prooftree window.

The communication protocol is completely described in the `ocamldoc` documentation of `input.ml` in the Prooftree sources. All messages consist of UTF-8 encoded human-readable strings. The strings have either a fixed length or their byte-length is encoded in the message before the string itself.

For debugging purposes Prooftree can save all input in a file. This feature can be turned on in the `Debug` tab of the Prooftree configuration dialog or with option `-tee`. The text that Prooftree sends to Proof General can be found in buffer `*proof-tree*`.

12.3.3 Guards

The proof-tree display code inside Proof General uses two guard variables.

`proof-tree-configured` [Variable]
Whether external proof-tree display is configured.
This boolean enables the proof-tree menu entry and the function that starts external proof-tree display.

`proof-tree-external-display` [Variable]

Display proof trees in external prooftree windows if t.
Actually, if this variable is t then the user requested an external proof-tree display. If there was no unfinished proof when proof-tree display was requested and if no proof has been started since then, then there is obviously no proof-tree display. In this case, this variable stays t and the proof-tree display will be started for the next proof.

Controlled by '`proof-tree-external-display-toggle`'.

In Proof General, the code for the external proof-tree display is called from the proof-shell filter function in `proof-shell-exec-loop` and `proof-shell-filter-manage-output`, see Section 14.6 [Proof shell mode], page 59. The variable `proof-tree-external-display` is a guard for these calls, to ensure that the proof-tree specific code is only called if the user requested a proof-tree display.

The whole proof-tree package contains only one function that can be called interactively: `proof-tree-external-display-toggle`, which switches `proof-tree-external-display` on and off. When `proof-tree-configured` is `nil`, `proof-tree-external-display-toggle` aborts with an error message.

`proof-tree-external-display-toggle` [Command]

Toggle the external proof-tree display.
When called outside a proof the external proof-tree display will be enabled for the next proof. When called inside a proof the proof display will be created for the current proof. If the external proof-tree display is currently on, then this toggle will switch it off. At the end of the proof the proof-tree display is switched off.

12.3.4 Urgent and Delayed Actions

The proof-shell filter functions contains two calls to proof-tree specific code. One for urgent actions and one for all remaining actions, that can be delayed.

Urgent actions are those that must be executed before `proof-shell-exec-loop` sends the next item from `proof-action-list` to the proof assistant. For execution speed, the amount of urgent code should be kept small.

`proof-tree-urgent-action` *flags* [Function]

Handle urgent points before the next item is sent to the proof assistant.
Schedule goal updates when existential variables have changed and call '`proof-tree-urgent-action-hook`'. All this is only done if the current output does not come from a command (with the '`proof-tree-show-subgoal` flag) that this package inserted itself.

Urgent actions are only needed if the external proof display is currently running. Therefore this function should not be called when '`proof-tree-external-display`' is nil.

This function assumes that the prover output is not suppressed. Therefore, '`proof-tree-external-display`' being t is actually a necessary precondition.

The not yet delayed output is in the region [`proof-shell-delayed-output-start`, `proof-shell-delayed-output-end`].

The function `proof-tree-urgent-action` is called at a point where it is save to manipulate `proof-action-list`. This is essential, because `proof-tree-urgent-action` inserts goal display commands into `proof-action-list` when existential variables got instantiated and when the sequent text from newly created subgoals is missing.

Most of the proof-tree specific code runs when the proof assistant is already busy with the next item from `proof-action-list`.

`proof-tree-handle-delayed-output` *old-proof-marker cmd flags _span* [Function]
Process delayed output for prooftree.
This function is the main entry point of the Proof General prooftree support. It examines the delayed output in order to take appropriate actions and maintains the internal state.
The delayed output to handle is in the region [`proof-shell-delayed-output-start`, `proof-shell-delayed-output-end`]. Urgent messages might be before that, following *old-proof-marker*, which contains the position of '`proof-marker`', before the next command was sent to the proof assistant.
All other arguments are (former) fields of the '`proof-action-list`' entry that is now finally retired. *cmd* is the command, *flags* are the flags and *span* is the span.

The function `proof-tree-handle-delayed-output` does all the communication with Prooftree.

12.3.5 Full Annotation

In the default configuration Proof General switches the proof assistant into quiet mode if there are more than `proof-shell-silent-threshold` items in `proof-action-list`, see Section *Document centred working* (in Chapter *Advanced Script Management and Editing*) in the *Proof General* users manual. The proof-tree display needs of course the full output from the proof assistant. Therefore `proof-shell-should-be-silent` keeps the proof assistant noisy when the proof-tree display is switched on.

12.4 Configuring Prooftree for a New Proof Assistant

To get the proof-tree display running for a new proof assistant one has to configure the proof-tree Elisp code and adapt the Prooftree program.

12.4.1 Proof Tree Elisp configuration

All variables that need to be configured are in the customization group `proof-tree-internals`. Most of these variables are regular expressions for extracting various parts from the proof assistant output. However, some are functions that need to be implemented as prover specific part of the proof display code.

The variables `proof-tree-configured`, `proof-tree-get-proof-info` and `proof-tree-find-begin-of-unfinished-proof` might be used before the proof assistant is running inside a proof shell. They must therefore be configured as part of the proof assistant editing mode.

The other variables are only used when the proof shell is running. They can therefore be configured with the proof assistant proof-shell mode.

12.4.2 Prooftree Adaption

To make the new proof assistant known to Prooftree, the match in function `configure_prooftree` in `input.ml` must be extended. If the new proof assistant does not support existential variables adding a line

```
| "new-pa-name" -> ()
```

suffices.

If the new prover supports existential variables, Prooftree must be extended with a parser for the existential variable information printout of the proof assistant. The parser for Coq is contained in the file `coq.ml`. Then the function `configure_prooftree` must assign this new parser to the reference `parse_existential_info`.

13 Writing More Lisp Code

You may want to add some extra features to your instance of Proof General which are not supported in the generic core. To do this, you can use the settings described above, plus a small number of fundamental functions in Proof General which you can consider as exported in the generic interface. Be careful using more functions than are mentioned here because the internals of Proof General may change between versions.

13.1 Default values for generic settings

Several generic settings are defined using `defpgcustom` in `proof-config.el`. This introduces settings of the form `<PA>-name` for each proof assistant *PA*.

To set the default value for these settings in prover-specific cases, you should use the special `defpgdefault` macro:

`defpgdefault` [Macro]

Set default for the proof assistant specific variable <PA>-*sym* to *value*.
This should be used in prover-specific code to alter the default values for prover specific settings.

Usage: (defpgdefault SYM *value*)

In your prover-specific code you can simply use the setting `<PA>-sym` directly, i.e., write `myprover-home-page`.

In the generic code, you can use a macro, writing `(proof-ass home-page)` to refer to the `<PA>-home-page` setting for the currently running instance of Proof General.

See Section 14.3 [Configuration variable mechanisms], page 54, for more details on this mechanism.

13.2 Adding prover-specific configurations

Apart from the generic settings, your prover instance will probably need some specific customizable settings.

Defining new prover-specific settings using customize is pretty easy. You should do it at least for your prover-specific user options.

The code in `proof-site.el` provides each prover with two customization groups automatically (based on the name of the assistant): `<PA>` for user options for prover *PA* and `<PA>-config` for configuration of prover *PA*. Typically `<PA>-config` holds settings which are constants but which may be nice to tweak.

The first group appears in the menu

```
ProofGeneral -> Advanced -> Customize -> <PA>
```

The second group appears in the menu:

```
ProofGeneral -> Internals -> <PA> config
```

A typical use of `defcustom` looks like this:

```
(defcustom myprover-search-page
  "http://findtheorem.myprover.org"
  "URL of search web page for myprover."
  :type 'string
  :group 'myprover-config)
```

This introduces a new customizable setting, which you might use to make a menu entry, for example. The default value is the string `"http://findtheorem.myprover.org"`.

13.3 Useful variables

In `proof-site`, some architecture flags are defined. These can be used to write conditional pieces of code for different Emacs and operating systems. They are referred to mainly in `proof-compat` (which helps to keep the architecture and version dependent code in one place).

13.4 Useful functions and macros

The recommended functions you may invoke are these:

- Any of the interactive commands (i.e. anything you can invoke with `M-x`, including all key-bindings)
- Any of the internal functions and macros mentioned below

To insert text into the current (usually script) buffer, the function `proof-insert` is useful. There's also a handy macro `proof-defshortcut` for defining shortcut functions using it.

`proof-insert` *text* [Function]

Insert *text* into the current buffer.

text may include these special characters:

```
%p  - place the point here after input
```

Any other %-prefixed character inserts itself.

`proof-defshortcut` [Macro]

Define shortcut function FN to insert *string*, optional keydef KEY.

This is intended for defining proof assistant specific functions. *string* is inserted using '`proof-insert`', which see. KEY is added onto proof assistant map.

The function `proof-shell-invisible-command` is a useful utility for sending a single command to the process. You should use this to implement user-level or internal functions rather than attempting to directly manipulate the proof action list, or insert into the shell buffer.

`proof-shell-invisible-command` *cmd* **&optional** *wait invisiblecallback* **&rest** *flags* [Function]

Send *cmd* to the proof process.

The *cmd* is '`invisible`' in the sense that it is not recorded in buffer. *cmd* may be a string or a string-yielding expression.

Automatically add '`proof-terminal-string`' if necessary, examining '`proof-shell-no-auto-terminate-commands`'.

By default, let the command be processed asynchronously. But if optional *wait* command is non-nil, wait for processing to finish before and after sending the command.

In case *cmd* is (or yields) nil, do nothing.

invisiblecallback will be invoked after the command has finished, if it is set. It should probably run the hook variables '`proof-state-change-hook`'.

flags are additional flags to put onto the '`proof-action-list`'. The flag `'invisible` is always added to *flags*.

There are several handy macros to help you define functions which invoke `proof-shell-invisible-command`.

`proof-definvisible` [Macro]

Define function FN to send *string* to proof assistant, optional keydef KEY.

This is intended for defining proof assistant specific functions. *string* is sent using '`proof-shell-invisible-command`', which see. *string* may be a string or a function which returns a string. KEY is added onto proof assistant map.

`proof-define-assistant-command` [Macro]
Define FN (docstring DOC): check if *cmdvar* is set, then send *body* to prover.
body defaults to *cmdvar*, a variable.

`proof-define-assistant-command-witharg` [Macro]
Define FN (arg) with DOC: check *cmdvar* is set, *prompt* a string and eval *body*.
The *body* can contain occurrences of arg. *cmdvar* is a variable holding a function or string.
Automatically has history.

`proof-format-filename` *string filename* [Function]
Format *string* by replacing quoted chars by escaped version of *filename*.

%e uses the canonicalized expanded version of filename (including directory, using '`default-directory`' – see '`expand-file-name`').

%r uses the unadjusted (possibly relative) version of *filename*.

%m ('module') uses the basename of the file, without directory or extension.

%s means the same as %e.

Using %e can avoid problems with dumb proof assistants who don't understand `~`, for example.

For all these cases, the escapes in '`proof-shell-filename-escapes`' are processed.

If *string* is in fact a function, instead invoke it on *filename* and return the resulting (string) value.

14 Internals of Proof General

This chapter sketches some of the internal functions and variables of Proof General, to help developers who wish to understand or modify the code.

Most of the documentation below is generated automatically from the comments in the code. Because Emacs lisp is interpreted and self-documenting, the best way to find your way around the source is inside Emacs once Proof General is loaded. Read the source files, and use functions such as `C-h v` and `C-h f`.

The code is split into files. The following sections document the important files, kept in the `generic/` subdirectory.

14.1 Spans

Spans are an abstraction of Emacs *overlays* originally used to help bridge the gulf between GNU Emacs and XEmacs. See the file `lib/span.el`. XEmacs calls these *extents* which is a name still used in some parts of the code.

14.2 Proof General site configuration

The file `proof-site.el` contains the initial configuration for Proof General for the site (or user) and the choice of provers.

The first part of the configuration is to set `proof-home-directory` to the directory that `proof-site.el` is located in, or to the variable of the environment variable `PROOFGENERAL_HOME` if that is set.

`proof-home-directory` [Variable]
Directory where Proof General is installed.
Based on where the file '`proof-site.el`' was loaded from. Falls back to consulting the environment variable '`PROOFGENERAL_HOME`' if proof-site.el couldn't know where it was executed from.

Further directory variables allow the files of Proof General to be split up and installed across a system if need be, rather than under the `proof-home-directory` root.

`proof-images-directory` [Variable]
Where Proof General image files are installed. Ends with slash.

`proof-info-directory` [Variable]
Where Proof General Info files are installed. Ends with slash.

After defining these settings, we define a *mode stub* for each proof assistant enabled. The mode stub will autoload Proof General for the right proof assistant when a file is visited with the corresponding extension. The proof assistants enabled are the ones listed in the `proof-assistants` setting.

`proof-assistants` [Variable]
Choice of proof assistants to use with Proof General.
A list of symbols chosen from: '`coq`' '`easycrypt`' '`phox`' '`qrhl`' '`pgshell`' '`pgocaml`' '`pghaskell`'. If nil, the default will be ALL available proof assistants.

Each proof assistant defines its own instance of Proof General, providing session control, script management, etc. Proof General will be started automatically for the assistants chosen here. To avoid accidently invoking a proof assistant you don't have, only select the proof assistants you (or your site) may need.

You can select which proof assistants you want by setting this variable before '`proof-site.el`' is loaded, or by setting the environment variable '`PROOFGENERAL_ASSISTANTS`' to the symbols you want, for example `"coq easycrypt"`. Or you can edit the file '`proof-site.el`' itself.

Note: to change proof assistant, you must start a new Emacs session.

The file `proof-site.el` also defines a version variable.

`proof-general-version` [Variable]
Version string identifying Proof General release.

14.3 Configuration variable mechanisms

The file `proof-config.el` defines the configuration variables for Proof General, including instantiation parameters and user options. See previous chapters for details of its contents. Here we mention some conventions for declaring user options.

Global user options and instantiation parameters are declared using `defcustom` as usual. User options should have '`*`' as the first character of their docstrings (standard Emacs convention) and live in the customize group `proof-user-options`. See `proof-config.el` for the groups for instantiation parameters.

User options which are generic (having separate instances for each prover) and instantiation parameters (by definition generic) can be declared using the special macro `defpgcustom`. It is used in the same way as `defcustom`, except that the symbol declared will automatically be prefixed by the current proof assistant symbol.

`defpgcustom` [Macro]
Define a new customization variable <PA>-*sym* for the current proof assistant.
This is intended for defining settings which are useful for any prover, but which the user may require different values of across provers.

The function proof-assistant-<SYM> is also defined, which can be used in the generic portion of Proof General to access the value for the current prover.

Arguments *args* are as for '`defcustom`', which see. If a :group argument is not supplied, the setting will be added to the internal settings for the current prover (named <PA>-config).

In specific instances of Proof General, the macro `defpgdefault` can be used to give a default value for a generic setting.

`defpgdefault` [Macro]
Set default for the proof assistant specific variable <PA>-*sym* to *value*.
This should be used in prover-specific code to alter the default values for prover specific settings.

Usage: (defpgdefault SYM *value*)

All new instantiation variables are best declared using the `defpgcustom` mechanism (old code may be converted gradually). Customizations which are liable to be different for different instances of Proof General are also best declared in this way. An example is the use of X Symbol, controlled by *PA*`-x-symbol-enable`, since it works poorly or not at all with some provers.

To access the generic settings, the following four functions and macros are useful.

`proof-ass` [Macro]
Return the value for SYM for the current prover.
This macro should only be invoked once a specific prover is engaged.

`proof-ass-sym` [Macro]
Return the symbol for SYM for the current prover. SYM not evaluated.
This macro should only be called once a specific prover is known.

`proof-ass-symv` [Macro]
Return the symbol for SYM for the current prover. SYM evaluated.
This macro should only be invoked once a specific prover is engaged.

If changing a user option setting amounts to more than just setting a variable (it may have some dynamic effect), we can set the `custom-set` property for the variable to the function `proof-set-value` which does an ordinary `set-default` to set the variable, and then calls a function with the same name as the variable, to do whatever is necessary according to the new value for the variable.

There are several settings which can be switched on or off by the user, which use this `proof-set-value` mechanism. They are controlled by boolean variables with names like `proof-`*foo*`-enable`, and appear at the start of the customize group `proof-user-options`. They should be edited by the user through the customization mechanism, and set in the code using `customize-set-variable`.

In `proof-utils.el` there is a handy macro, `proof-deftoggle`, which constructs an interactive function for toggling boolean customize settings. We can use this to make an interactive function `proof-`*foo*`-toggle` to put on a menu or bind to a key, for example.

This general scheme is followed as far as possible, to give uniform behaviour and appearance for boolean user options, as well as interfacing properly with the `customize` mechanism.

`proof-set-value` *sym value* [Function]
Set a customize variable using '`set-default`' and a function.
We first call '`set-default`' to set *sym* to *value*. Then if there is a function *sym* (i.e. with the same name as the variable *sym*), it is called to take some dynamic action for the new setting.

If there is no function *sym*, we try stripping '`proof-assistant-symbol`' and adding "proof-" instead to get a function name. This extends `proof-set-value` to work with generic individual settings.

The dynamic action call only happens when values **change**: as an approximation we test whether proof-config is fully-loaded yet.

`proof-deftoggle` [Macro]
Define a function VAR-toggle for toggling a boolean customize setting VAR.
The toggle function uses '`customize-set-variable`' to change the variable. *othername* gives an alternative name than the default <VAR>-toggle. The name of the defined function is returned.

14.4 Global variables

Global variables are defined in `proof.el`. The same file defines a few utility functions and some triggers to load in the other files.

`proof-script-buffer` [Variable]
The currently active scripting buffer or nil if none.

`proof-shell-buffer` [Variable]
Process buffer where the proof assistant is run.

`proof-response-buffer` [Variable]
The response buffer.

`proof-goals-buffer` [Variable]
The goals buffer.

`proof-buffer-type` [Variable]
Symbol for the type of this buffer: `'script`, `'shell`, `'goals`, or `'response`.

`proof-included-files-list` [Variable]
List of files currently included in proof process.
This list contains files in canonical truename format (see '`file-truename`').

Whenever a new file is being processed, it gets added to this list via the '`proof-shell-process-file`' configuration settings. When the prover retracts a file, this list is resynchronised via the '`proof-shell-retract-files-regexp`' and '`proof-shell-compute-new-files-list`' configuration settings.

Only files which have been **fully** processed should be included here. Proof General itself will automatically add the filenames of a script buffer which has been completely read when scripting is deactivated. It will automatically remove the filename of a script buffer which is completely unread when scripting is deactivated.

NB: Currently there is no generic provision for removing files which are only partly read-in due to an error, so ideally the proof assistant should only output a processed message when a file has been successfully read.

`proof-shell-proof-completed` [Variable]
Flag indicating that a completed proof has just been observed.
If non-nil, the value counts the commands from the last command of the proof (starting from 1).

`proof-shell-error-or-interrupt-seen` [Variable]
Flag indicating that an error or interrupt has just occurred.
Set to `'error` or `'interrupt` if one was observed from the proof assistant during the last group of commands.

14.5 Proof script mode

The file `proof-script.el` contains the main code for proof script mode, as well as definitions of menus, key-bindings, and user-level functions.

Proof scripts have two important variables for the locked and queue regions. These variables are local to each script buffer (although we only really need one queue span in total rather than one per buffer).

`proof-locked-span` [Variable]
The locked span of the buffer.
Each script buffer has its own locked span, which may be detached from the buffer. Proof General allows buffers in other modes also to be locked; these also have a non-nil value for this variable.

`proof-queue-span` [Variable]
The queue span of the buffer. May be detached if inactive or empty.
Each script buffer has its own queue span, although only the active scripting buffer may have an active queue span.

Various utility functions manipulate and examine the spans. An important one is `proof-init-segmentation`.

`proof-init-segmentation` [Function]
Initialise the queue and locked spans in a proof script buffer.
Allocate spans if need be. The spans are detached from the buffer, so the regions are made empty by this function. Also clear list of script portions.

For locking files loaded by a proof assistant, we use the next function.

`proof-complete-buffer-atomic` *buffer* [Function]
Ensure *buffer* marked completely processed, completing with a single step.

If buffer already contains a locked region, only the remainder of the buffer is closed off atomically (although undo for the initial portion is unlikely to work, the decoration may be worth retaining).

This works for buffers which are not in proof scripting mode too, to allow other files loaded by proof assistants to be marked read-only.

Atomic locking is instigated by the next function, which uses the variables `proof-included-files-list` documented earlier (see Chapter 8 [Handling Multiple Files], page 35, and see Section 14.4 [Global variables], page 55).

`proof-register-possibly-new-processed-file` *file* **&optional** *informprover noquestions* [Function]
Register a possibly new *file* as having been processed by the prover.

If *informprover* is non-nil, the proof assistant will be told about this, to co-ordinate with its internal file-management. (Otherwise we assume that it is a message from the proof assistant which triggers this call). In this case, the user will be queried to save some buffers, unless *noquestions* is non-nil.

No action is taken if the file is already registered.

A warning message is issued if the register request came from the proof assistant and Emacs has a modified buffer visiting the file.

(Unlocking is done by `proof-shell-process-urgent-message-retract` together with `proof-restart-buffers`.)

An important pair of functions activate and deactivate scripting for the current buffer. A change in the state of active scripting can trigger various actions, such as starting up the proof assistant, or altering `proof-included-files-list`.

`proof-activate-scripting` **&optional** *nosaves queuemode* [Command]
Ready prover and activate scripting for the current script buffer.

The current buffer is prepared for scripting. No changes are necessary if it is already in Scripting minor mode. Otherwise, it will become the new active scripting buffer, provided scripting can be switched off in the previous active scripting buffer with '`proof-deactivate-scripting`'.

Activating a new script buffer is a good time to ask if the user wants to save some buffers; this is done if the user option '`proof-query-file-save-when-activating-scripting`' is set and provided the optional argument *nosaves* is non-nil.

The optional argument *queuemode* relaxes the test for a busy proof shell to allow one which has mode *queuemode*. In all other cases, a proof shell busy error is given.

Finally, the hooks '`proof-activate-scripting-hook`' are run. This can be a useful place to configure the proof assistant for scripting in a particular file, for example, loading the correct theory, or whatever. If the hooks issue commands to the proof assistant (via '`proof-shell-invisible-command`') which result in an error, the activation is considered to have failed and an error is given.

`proof-deactivate-scripting` **&optional** *forcedaction* [Command]
Try to deactivate scripting for the active scripting buffer.

Aims to set '`proof-script-buffer`' to nil and turn off the modeline indicator. No action is required there is no active scripting buffer.

We make sure that the active scripting buffer either has no locked region or a full locked region (everything in it has been processed). If this is not already the case, we question the user whether to retract or assert, or automatically take the action indicated in the user option '`proof-auto-action-when-deactivating-scripting`'.

If '`proof-no-fully-processed-buffer`' is t there is only the choice to fully retract the active scripting buffer. In this case the active scripting buffer is retracted even if it was fully processed. Setting '`proof-auto-action-when-deactivating-scripting`' to `'process` is ignored in this case.

If the scripting buffer is (or has become) fully processed, and it is associated with a file, it is registered on '`proof-included-files-list`'. Conversely, if it is (or has become) empty, we make sure that it is **not** registered. This is to be certain that the included files list behaves as we might expect with respect to the active scripting buffer, in an attempt to harmonize mixed scripting and file reading in the prover.

This function either succeeds, fails because the user refused to process or retract a partly finished buffer, or gives an error message because retraction or processing failed. If this function succeeds, then '`proof-script-buffer`' is nil afterwards.

The optional argument *forcedaction* overrides the user option '`proof-auto-action-when-deactivating-scripting`' and prevents questioning the user. It is used to make a value for the '`kill-buffer-hook`' for scripting buffers, so that when a scripting buffer is killed it is always retracted.

The function `proof-segment-up-to` is the main one used for parsing the proof script buffer. There are several variants of this function available corresponding to different parsing strategies; the appropriate one is aliased to `proof-segment-up-to` according to which configuration variables have been set.

- If `proof-script-sexp-commands` is set, the choice is `proof-script-generic-parse-sexp`. item If only `proof-script-command-end-regexp` or `proof-terminal-string` are set, then the default is `proof-script-generic-parse-cmdend`.
- If `proof-script-command-start-regexp` is set, the choice is `proof-script-generic-parse-cmdstart`.

The function `proof-semis-to-vanillas` uses `proof-segment-up-to` to convert a parsed region of the script into a series of commands to be sent to the proof assistant.

`proof-script-generic-parse-cmdend` [Function]
For '`proof-script-parse-function`' if '`proof-script-command-end-regexp`' set.

`proof-script-generic-parse-cmdstart` [Function]
For '`proof-script-parse-function`' if '`proof-script-command-start-regexp`' is set.

`proof-script-generic-parse-sexp` [Function]
Used for '`proof-script-parse-function`' if '`proof-script-sexp-commands`' is set.

`proof-semis-to-vanillas` *semis* **&optional** *queueflags* [Function]
Create vanilla spans for *semis* and a list for the queue.
Proof terminator positions *semis* has the form returned by the function '`proof-segment-up-to`'. The argument list is destroyed. The callback in each queue element is '`proof-done-advancing`'.

If the variable '`proof-script-preprocess`' is set (to the name of a function), call that function to construct the first element of each queue item.

The optional *queueflags* are added to each queue item.

The function `proof-assert-until-point` is the main one used to process commands in the script buffer. It's actually used to implement the assert-until-point, electric terminator keypress, and find-next-terminator behaviours. In different cases we want different things, but usually the information (i.e. are we inside a comment) isn't available until we've actually run `proof-segment-up-to (point)`, hence all the different options when we've done so.

`proof-assert-until-point` **&optional** *displayflags* [Function]
Process the region from the end of the locked-region until point.

The main command for retracting parts of a script is `proof-retract-until-point`.

`proof-retract-until-point` **&optional** *undo-action displayflags* [Function]
Set up the proof process for retracting until point.
This calculates the commands to undo to the current point within the locked region. If invoked outside the locked region, undo the last successfully processed command. See '`proof-retract-target`'.

After retraction has succeeded in the prover, the filter will call '`proof-done-retracting`'. If *undo-action* is non-nil, it will then be invoked on the region in the proof script corresponding to the proof command sequence. *displayflags* control output shown to user, see '`proof-action-list`'.

Before the retraction is calculated, we enforce the file-level protocol with '`proof-activate-scripting`'. This has a couple of effects:

1. If the file is completely processed, we have to re-open it for scripting again which may involve retracting other (dependent) files.

2. We may query the user whether to save some buffers.

Step 2 may seem odd – we're undoing (in) the buffer, after all – but what may happen is that when scripting starts going forward again, we hit a command that loads other files, but the user hasn't saved the latest edits. Therefore it is right to query saves here.

To clean up when scripting is stopped, a script buffer is killed, a file is retract (and thus must be unlocked), or the proof assistant exits, we use the functions `proof-restart-buffers` and `proof-script-remove-all-spans-and-deactivate`.

`proof-restart-buffers` *buffers* [Function]
Remove all extents in *buffers* and maybe reset '`proof-script-buffer`'.
The high-level effect is that all members of *buffers* are completely unlocked, including all the necessary cleanup. No effect on a buffer which is nil or killed. If one of the buffers is the current scripting buffer, then '`proof-script-buffer`' will deactivated.

`proof-script-remove-all-spans-and-deactivate` [Function]
Remove all spans from scripting buffers via '`proof-restart-buffers`'.

14.6 Proof shell mode

The proof shell mode code is in the file `proof-shell.el`. Proof shell mode is defined to inherit from `scomint-mode` using `define-derived-mode` near the end of the file. The `scomint.el` package stands for "simplified comint", where `comint-mode` is the standard Emacs mode for running an embedded command interpreter. In `scomint`, many of the interactive commands have been removed to speed up the process handling, because it isn't intended that the user interacts directly with the shell in Proof General.

The bulk of the code in the `proof-shell` package is concerned with sending code to and from the shell, and processing output for the associated buffers (goals and response).

Good process handling is a tricky issue. Proof General attempts to manage the process strictly, by maintaining a queue of commands to send to the process. Once a command has been processed, another one is popped off the queue and sent.

There are several important internal variables which control interaction with the process.

`proof-shell-busy` [Variable]

A lock indicating that the proof shell is processing.

The lock notes that we are processing a queue of commands being sent to the prover, and indicates whether the commands correspond to script management from a buffer (rather than being ad-hoc query commands to the prover).

When processing commands from a buffer for script management, this will be set to the queue mode `'advancing` or `'retracting` to indicate the direction of movement.

When this is non-nil, '`proof-shell-ready-prover`' will give an error if called with a different requested queue mode.

See also functions '`proof-activate-scripting`' and '`proof-shell-available-p`'.

`proof-marker` [Variable]

Marker in proof shell buffer pointing to previous command input.

`proof-action-list` [Variable]

The main queue of things to do: spans, commands and actions.
The value is a list of lists of the form

```
(span commands action [DISPLAYFLAGS])
```

which is the queue of things to do.

span is a region in the sources, where *commands* come from. Often, additional properties are recorded as properties of *span*.

commands is a list of strings, holding the text to be send to the prover. It might be the empty list if nothing needs to be sent to the prover, such as, for comments. Usually *commands* contains just 1 string, but it might also contains more elements. The text should be obtained with '`(mapconcat` 'identity *commands* `" "`)', where the last argument is a space.

action is the callback to be invoked when this item has been processed by the prover. For normal scripting items it is '`proof-done-advancing`', for retract items '`proof-done-retracting`', but there are more possibilities (e.g. '`proof-done-invisible`', '`proof-shell-set-silent`', '`proof-shell-clear-silent`' and '`proof-tree-show-goal-callback`').

The *displayflags* are set for non-scripting commands or for when scripting should not bother the user. They may include

```
'invisible                non-script command (‘proof-shell-invisible-command’)
'no-response-display      do not display messages in response buffer
'no-error-display         do not display errors/take error action
'no-goals-display         do not goals in goals buffer
'proof-tree-show-subgoal  item inserted by the proof-tree package
'priority-action          item added via proof-add-to-priority-queue
```

Note that `'invisible` does not imply any of the others. If flags are non-empty, interactive cues will be surpressed. (E.g., printing hints).

See the functions '`proof-start-queue`' and '`proof-shell-exec-loop`'.

In Proof General 4.2 and earlier it was always the case that all items from the queue region were present in `proof-action-list`. Because of the new parallel background compilation for Coq, this is no longer the case. Prover specific code may now store items from the queue region somewhere else. To notify generic Proof General about this, it must set `proof-second-action-list-active` for the time where some queue items are missing from `proof-action-list`. In this case Proof General keeps the proof shell lock and the queue span even in case `proof-action-list` gets empty. Coq uses this feature to hold back Require commands and the following text until the asynchronous background compilation finishes.

`proof-second-action-list-active` [Variable]

Signals that some items are waiting outside of '`proof-action-list`'.
If this is t it means that some items from the queue region are waiting for being processed in a place different from '`proof-action-list`'. In this case Proof General must behave as if '`proof-action-list`' would be non-empty, when it is, in fact, empty.

This is used, for instance, for parallel background compilation for Coq: The Require command and the following items are not put into '`proof-action-list`' and are stored somewhere else until the background compilation finishes. Then those items are put into '`proof-action-list`' for getting processed.

`pg-subterm-anns-use-stack` [Variable]

Choice of syntax tree encoding for terms.

If nil, prover is expected to make no optimisations. If non-nil, the pretty printer of the prover only reports local changes. For Coq 6.2, use t.

The function `proof-shell-start` is used to initialise a shell buffer and the associated buffers.

`proof-shell-start` [Command]

Initialise a shell-like buffer for a proof assistant.
Does nothing if proof assistant is already running.

Also generates goal and response buffers.

If '`proof-prog-name-ask`' is set, query the user for the process command.

The function `proof-shell-kill-function` performs the converse function of shutting things down; it is used as a hook function for `kill-buffer-hook`. Then no harm occurs if the user kills the shell directly, or if it is done more cautiously via `proof-shell-exit`. The function `proof-shell-restart` allows a less drastic way of restarting scripting, other than killing and restarting the process.

`proof-shell-kill-function` [Function]

Function run when a proof-shell buffer is killed.
Try to shut down the proof process nicely and clear locked regions and state variables. Value for '`kill-buffer-hook`' in shell buffer, called by '`proof-shell-bail-out`' if process exits.

`proof-shell-exit` **&optional** *dont-ask* [Command]

Query the user and exit the proof process.

This simply kills the '`proof-shell-buffer`' relying on the hook function

'`proof-shell-kill-function`' to do the hard work. If optional argument *dont-ask* is non-nil, the proof process is terminated without confirmation.

The kill function uses '`<PA>-quit-timeout`' as a timeout to wait after sending '`proof-shell-quit-cmd`' before rudely killing the process.

This function should not be called if '`proof-shell-exit-in-progress`' is t, because a recursive call of '`proof-shell-kill-function`' will give strange errors.

`proof-shell-bail-out` *process event* [Function]
Value for the process sentinel for the proof assistant *process*.
If the proof assistant dies, run '`proof-shell-kill-function`' to cleanup and remove the associated buffers. The shell buffer is left around so the user may discover what killed the process. *event* is the string describing the change.

`proof-shell-restart` [Command]
Clear script buffers and send '`proof-shell-restart-cmd`'.
All locked regions are cleared and the active scripting buffer deactivated.

If the proof shell is busy, an interrupt is sent with '`proof-interrupt-process`' and we wait until the process is ready.

The restart command should re-synchronize Proof General with the proof assistant, without actually exiting and restarting the proof assistant process.

It is up to the proof assistant how much context is cleared: for example, theories already loaded may be "cached" in some way, so that loading them the next time round only performs a re-linking operation, not full re-processing. (One way of caching is via object files, used by Coq).

14.6.1 Input to the shell

Input to the proof shell via the queue region is managed by the functions `proof-extend-queue` and `proof-shell-exec-loop`.

`proof-extend-queue` *end queueitems* [Function]
Extend the current queue with *queueitems*, queue end *end*.
To make sense, the commands should correspond to processing actions for processing a region from (buffer-queue-or-locked-end) to *end*. The queue mode is set to `'advancing`

`proof-extend-queue` *end queueitems* [Function]
Extend the current queue with *queueitems*, queue end *end*.
To make sense, the commands should correspond to processing actions for processing a region from (buffer-queue-or-locked-end) to *end*. The queue mode is set to `'advancing`

`proof-shell-exec-loop` [Function]
Main loop processing the '`proof-action-list`', called from shell filter.

'`proof-action-list`' contains a list of (*span command action* [FLAGS]) lists.

If this function is called with a non-empty '`proof-action-list`', the head of the list is the previously executed command which succeeded. We execute the callback (*action span*) on the first item, then (*action span*) on any following items which have null as their cmd components.

If a there is a next command after that, send it to the process.

If the action list becomes empty, unlock the process and remove the queue region.

The return value is non-nil if the action list is now empty or contains only invisible elements for Prooftree synchronization.

Input is actually inserted into the shell buffer and sent to the process by the low-level function `proof-shell-insert`.

`proof-shell-insert` *strings action* **&optional** *scriptspan* [Function]
Insert *strings* at the end of the proof shell, call '`scomint-send-input`'.

strings is a list of strings (which will be concatenated), or a single string.

The *action* argument is a symbol which is typically the name of a callback for when each string has been processed.

This calls '`proof-shell-insert-hook`'. The arguments *action* and *scriptspan* may be examined by the hook to determine how to modify the string variable (exploiting dynamic scoping) which will be the command actually sent to the shell.

Note that the hook is not called for the empty (null) string or a carriage return.

We strip the string of carriage returns before inserting it and updating '`proof-marker`' to point to the end of the newly inserted text.

Do not use this function directly, or output will be lost. It is only used in '`proof-add-to-queue`' when we start processing a queue, and in '`proof-shell-exec-loop`', to process the next item.

When Proof General is processing a queue of commands, the lock is managed using a couple of utility functions. You should not need to use these directly.

`proof-grab-lock` **&optional** *queuemode* [Function]

Grab the proof shell lock, starting the proof assistant if need be.
Runs '`proof-state-change-hook`' to notify state change. If *queuemode* is supplied, set the lock to that value.

`proof-release-lock` [Function]

Release the proof shell lock. Clear '`proof-shell-busy`'.

14.6.2 Output from the shell

Two main functions deal with output, `proof-shell-classify-output` and `proof-shell-process-urgent-message`. In effect we consider the output to be two streams intermingled: the "urgent" messages which have "eager" annotations, as well as the ordinary ruminations from the prover.

The idea is to conceal as much irrelevant information from the user as possible; only the remaining output between prompts and after the last urgent message will be a candidate for the goal or response buffer. The internal variable `proof-shell-urgent-message-marker` tracks the last urgent message seen.

When output is grabbed from the prover process, the first action is to strip spurious carriage return characters from the end of lines, if `proof-shell-strip-crs-from-output` requires it. Then the output is stored into `proof-shell-last-output`, and its type is stored in `proof-shell-last-output-kind`. Output which is deferred or possibly discarded until the queue is empty is copied into `proof-shell-delayed-output`, with type `proof-shell-delayed-output-kind`. A record of the last prompt seen from the prover process is also kept, in `proof-shell-last-prompt`.

`proof-shell-strip-crs-from-output` [Variable]

If non-nil, remove carriage returns (^M) at the end of lines from output.
This is enabled for cygwin32 systems by default. You should turn it off if you don't need it (slight speed penalty).

`proof-shell-last-prompt` [Variable]

A raw record of the last prompt seen from the proof system.
This is the string matched by '`proof-shell-annotated-prompt-regexp`'.

`proof-shell-last-output` [Variable]

A record of the last string seen from the proof system.
This is raw string, for internal use only.

`proof-shell-last-output-kind` [Variable]

A symbol denoting the type of the last output string from the proof system.
Specifically:

```
'interrupt      An interrupt message
```

```
'error          An error message
'loopback       A command sent from the PA to be inserted into the script
'response       A response message
'goals          A goals (proof state) display
'systemspecific Something specific to a particular system,
                 -- see ‘proof-shell-handle-output-system-specific’
```

The output corresponding to this will be in ‘`proof-shell-last-output`’.

See also ‘`proof-shell-proof-completed`’ for further information about the proof process output, when ends of proofs are spotted.

This variable can be used for instance specific functions which want to examine ‘`proof-shell-last-output`’.

`proof-shell-last-output-kind` [Variable]

A symbol denoting the type of the last output string from the proof system.
Specifically:

```
'interrupt      An interrupt message
'error          An error message
'loopback       A command sent from the PA to be inserted into the script
'response       A response message
'goals          A goals (proof state) display
'systemspecific Something specific to a particular system,
                 -- see ‘proof-shell-handle-output-system-specific’
```

The output corresponding to this will be in ‘`proof-shell-last-output`’.

See also ‘`proof-shell-proof-completed`’ for further information about the proof process output, when ends of proofs are spotted.

This variable can be used for instance specific functions which want to examine ‘`proof-shell-last-output`’.

`proof-shell-delayed-output-start` [Variable]

A record of the start of the previous output in the shell buffer.
The previous output is held back for processing at end of queue.

`proof-shell-delayed-output-end` [Variable]

A record of the start of the previous output in the shell buffer.
The previous output is held back for processing at end of queue.

`proof-shell-delayed-output-flags` [Variable]

A copy of the ‘`proof-action-list`’ flags for ‘`proof-shell-delayed-output`’.

`proof-shell-handle-immediate-output` *cmd start end flags* [Function]

See if the output between *start* and *end* must be dealt with immediately.
To speed up processing, PG tries to avoid displaying output that the user will not have a chance to see. Some output must be handled immediately, however: these are errors, interrupts, goals and loopbacks (proof step hints/proof by pointing results).

In this function we check, in turn:

```
‘proof-shell-interrupt-regexp’
‘proof-shell-error-regexp’
‘proof-shell-proof-completed-regexp’
‘proof-shell-result-start’
```

Other kinds of output are essentially display only, so only dealt with if necessary.

To extend this, set '`proof-shell-handle-output-system-specific`', which is a hook to take particular additional actions.

This function sets variables: '`proof-shell-last-output-kind`', and the counter '`proof-shell-proof-completed`' which counts commands after a completed proof.

`proof-shell-handle-delayed-output` [Function]

Display delayed goals/responses, when queue is stopped or completed.
This function handles the cases of '`proof-shell-output-kind`' which are not dealt with eagerly during script processing, namely `'response` and `'goals` types.

This is useful even with empty delayed output as it will empty the buffers.

The delayed output is in the region [`proof-shell-delayed-output-start`,`proof-shell-delayed-output-end`].

If no goals classified output is found, the whole output is displayed in the response buffer.

If goals output is found, the last matching instance, possibly bounded by '`proof-shell-end-goals-regexp`', will be displayed in the goals buffer (and may be further analysed by Proof General).

Any output that appears **before** the last goals output (but after messages classified as urgent, see '`proof-shell-filter`') will also be displayed in the response buffer.

For example, if *output* has this form:

```
messsage-1
goals-1
message-2
goals-2
junk
```

then *goals-2* will be displayed in the goals buffer, and *message-2* in the response buffer. *junk* will be ignored.

Notice that the above alternation (and separation of *junk*) can only be distinguished if both '`proof-shell-start-goals-regexp`' and '`proof-shell-end-goals-regexp`' are set. With just the start goals regexp set, *goals-2 junk* will appear in the goals buffer and no response output would occur.

The goals and response outputs are copied into '`proof-shell-last-goals-output`' and '`proof-shell-last-response-output`' respectively.

The value returned is the value for '`proof-shell-last-output-kind`', i.e., `'goals` or `'response`.

`proof-shell-urgent-message-marker` [Variable]

Marker in proof shell buffer pointing to end of last urgent message.

`proof-shell-process-urgent-message` *start end* [Function]

Analyse urgent message between *start* and *end* for various cases.

Cases are: **trace** output, included/retracted files, cleared goals/response buffer, variable setting, xml-encoded *pgip* response, theorem dependency message or interactive output indicator.

If none of these apply, display the text between *start* and *end*.

The text between *start* and *end* should be a string that starts with text matching '`proof-shell-eager-annotation-start`' and ends with text matching '`proof-shell-eager-annotation-end`'.

The main processing point which triggers other actions is `proof-shell-filter`. It is called from `proof-shell-filter-wrapper`, which itself is called from an ordinary Emacs process filter inside the simplified `comint` library that is distributed with Proof General (in `lib/scomint.el`).

`proof-shell-filter` [Function]

Master filter for the proof assistant shell-process.
A function for '`scomint-output-filter-functions`'.

Deal with output and issue new input from the queue. This is an important internal function. The output must be collected from '`proof-shell-buffer`' for the following reason. This function might block inside '`process-send-string`' when sending input to the proof assistant or to prooftree. In this case Emacs might call the process filter again while the previous instance is still running. '`proof-shell-filter-wrapper`' detects and delays such calls but does not buffer the output.

Handle urgent messages first. As many as possible are processed, using the function '`proof-shell-process-urgent-messages`'.

If a prompt is seen, run '`proof-shell-filter-manage-output`' on the output between the new prompt and the last input (position of '`proof-marker`') or the last urgent message (position of '`proof-shell-urgent-message-marker`'), whichever is later. For example, in this case:

```
PROMPT> input
output-1
urgent-message-1
output-2
urgent-message-2
output-3
PROMPT>
```

'`proof-marker`' points after *input*.

'`proof-shell-urgent-message-marker`' points after *urgent-message-2*, after both urgent messages have been processed by '`proof-shell-process-urgent-messages`'. Urgent messages always processed; they are intended to correspond to informational notes that the prover makes to inform the user or interface on progress.

In this case, the ordinary outputs *output-1* and *output-2* are ignored; only *output-3* will be processed by '`proof-shell-filter-manage-output`'.

Error or interrupt messages are expected to terminate an interactive output and appear last before a prompt and will always be processed. Error messages and interrupt messages are therefore **not** considered as urgent messages.

The first time that a prompt is seen, '`proof-marker`' is initialised to the end of the prompt. This should correspond with initializing the process. After that, '`proof-marker`' is only changed when input is sent in '`proof-shell-insert`'.

`proof-shell-filter-manage-output` *start end* [Function]

Subroutine of '`proof-shell-filter`' for output between *start* and *end*.

First, we invoke '`proof-shell-handle-immediate-output`' which classifies and handles output that must be dealt with immediately.

Other output (user display) is only displayed when the proof action list becomes empty, to avoid a confusing rapidly changing output that slows down processing.

After processing the current output, the last step undertaken by the filter is to send the next command from the queue.

`proof-shell-filter-wrapper` *str-do-not-use* [Function]

Wrapper for '`proof-shell-filter`', protecting against parallel calls.
In Emacs a process filter function can be called while the same filter is currently running for the same process, for instance, when the filter blocks on I/O. This wrapper protects the main entry point, '`proof-shell-filter`' against such parallel, overlapping calls.

The argument *str-do-not-use* contains the most recent output, but is discarded. '`proof-shell-filter`' collects the output from '`proof-shell-buffer`' (where it is inserted by '`scomint-output-filter`'), relieving this function from the task to buffer the output that arrives during parallel, overlapping calls.

14.7 Debugging

To debug Proof General, it may be helpful to set the configuration variable `proof-general-debug`.

`proof-general-debug` [User Option]

Non-nil to run Proof General in debug mode.
This changes some behaviour (e.g. markup stripping) and displays debugging messages in the response buffer. To avoid erasing messages shortly after they're printed, set '`proof-tidy-response`' to nil. This is only useful for PG developers.

The default value is `nil`.

For more information about debugging Emacs lisp, consult the Emacs Lisp Reference Manual. I recommend using the source-level debugger `edebug`.

Appendix A Plans and Ideas

This appendix contains some tentative plans and ideas for improving Proof General.

This appendix is no longer extended: instead we keep a list of Proof General projects on the web, and forthcoming plans and ideas in the `TODO` and `todo` files included in the ordinary and developers PG distributions, respectively. Once the items mentioned below are implemented, they will be removed from here.

Please send us contributions to our wish lists, or better still, an offer to implement something from them!

A.1 Proof by pointing and similar features

This is a note by David Aspinall about proof by pointing and similar features.

Proof General already supports proof by pointing, and experimental support was provided in LEGO. We would like to extend this support to other proof assistants. Unfortunately, proof by pointing requires rather heavy support from the proof assistant. There are two aspects to the support:

- term structure mark-up
- proof by pointing command generation

Term structure mark-up is useful in itself: it allows the user to explore the structure of a term using the mouse (the smallest subexpression that the mouse is over is highlighted), and easily copy subterms from the output to a proof script.

Command generation for proof by pointing is usually specific to a particular logic in use, if we hope to generate a good proof command unambiguously for any particular click. However, Proof General could easily be generalised to offer the user a context-sensitive choice of next commands to apply, which may be more useful in practice, and a worthy addition to Proof General.

Implementors of new proof assistants should be encouraged to consider supporting term-structure mark up from the start. Command generation should be something that the logic-implementor can specify in some way.

Of the supported provers, we can certainly hope for proof-by-pointing support from Coq, since the CtCoq proof-by-pointing code has been moved into the Coq kernel lately. I hope the Coq community can encourage somebody to do this.

A.2 Granularity of atomic command sequences

This is a proposal by Thomas Kleymann for generalising the way Proof General handles sequences of proof commands (see *Goal-save sequences* in the user manual), particularly to make retraction more flexible.

The blue region of a script buffer contains the initial segment of the proof script which has been processed successfully. It consists of atomic sequences of commands (ACS). Retraction is supported to the beginning of every ACS. By default, every command is an ACS. But the granularity of atomicity should be able to be adjusted.

This is essential when arbitrary retraction is not supported. Usually, after a theorem has been proved, one may only retract to the start of the goal. One needs to mark the proof of the theorem as an ACS. At present, support for goal-save sequences (see *Goal-save sequences* in the user manual), has been hard wired. No other ACS are currently supported. We propose the following to overcome this deficiency:

`proof-atomic-sequents-list`
: is a list of instructions for setting up ACSs. Each instruction is a list of the form (*`end start &optional forget-command`*). *end* is a regular expression to recognise

the last command in an ACS. *start* is a function. Its input is the last command of an ACS. Its output is a regular expression to recognise the first command of the ACS. It is evaluated once and, starting with the command matched by *end*, the output is successively matched against previously processed commands until a match occurs (or the beginning of the current buffer is reached). The region determined by (*start*,*end*) is locked as an ACS. Optionally, the ACS is annotated with the actual command to retract the ACS. This is computed by applying *forget-command* to the first and last command of the ACS.

For convenience one might also want to allow *start* to be the symbol '`t`' as a convenient short-hand for `'(lambda (str) ".")` which always matches.

A.3 Browser mode for script files and theories

This is a proposal by David Aspinall for a browser window.

A browser window should provide support for browsing script files and theories. We should be able to inspect data in varying levels of detail, perhaps using outlining mechanisms. For theories, it would be nice to query the running proof assistant. This may require support from the assistant in the form of output which has been specially marked-up with an SGML like syntax, for example.

A browser would be useful to:

- Provide impoverished proof assistants with a browser
- Extend the uniform interface of Proof General to theory browsing
- Interact closely with proof script writing

The last point is the most important. We should be able to integrate a search mechanism for proofs of similar theorems, theorems containing particular constants, etc.

Appendix B Demonstration Instantiations

This appendix contains the code for the two demonstration instantiations of Proof General, for Isabelle.

These instantiations make an almost-bare minimum of settings to get things working. To add embellishments, you should refer to the instantiations for other systems distributed with Proof General.

B.1 demoisa-easy.el

```
;; demoisa-easy.el Example Proof General instance for Isabelle
;;
;; Copyright (C) 1999 LFCS Edinburgh.
;;
;; Author: David Aspinall <David.Aspinall@ed.ac.uk>
;;
;; $Id$
;;
;; This is an alternative version of demoisa.el which uses the
;; proof-easy-config macro to do the work of declaring derived modes,
;; etc.
;;
;; See demoisa.el and the Proof General manual for more documentation.
;;
;; To test this file you must rename it demoisa.el.
;;

(require 'proof-easy-config)            ; easy configure mechanism

(proof-easy-config
 'demoisa "Isabelle Demo"
 proof-prog-name  "isabelle"
 proof-terminal-string           ";"
 proof-script-comment-start             "(*"
 proof-script-comment-end               "*)"
 proof-goal-command-regexp       "^Goal"
 proof-save-command-regexp       "^qed"
 proof-goal-with-hole-regexp     "qed_goal \"\\(\\(.*\\)\\)\""
 proof-save-with-hole-regexp     "qed \"\\(\\(.*\\)\\)\""
 proof-non-undoables-regexp      "undo\\|back"
 proof-goal-command              "Goal \"%s\";"
 proof-save-command              "qed \"%s\";"
 proof-kill-goal-command         "Goal \"PROP no_goal_set\";"
 proof-showproof-command         "pr()"
 proof-undo-n-times-cmd          "pg_repeat undo %s;"
 proof-auto-multiple-files       t
 proof-shell-cd-cmd              "cd \"%s\""
 proof-shell-interrupt-regexp    "Interrupt"
 proof-shell-start-goals-regexp  "Level [0-9]"
 proof-shell-end-goals-regexp    "val it"
 proof-shell-quit-cmd            "quit();"
 proof-assistant-home-page
```

```
 "http://www.cl.cam.ac.uk/Research/HVG/Isabelle/"
 proof-shell-annotated-prompt-regexp
 "^\\(val it = () : unit\n\\)?ML>? "
 proof-shell-error-regexp
 "\\*\\*\\*\\|^.*Error:\\|^uncaught exception \\|^Exception- "
 proof-shell-init-cmd
 "fun pg_repeat f 0 = () | pg_repeat f n = (f(); pg_repeat f (n-1));"
 proof-shell-proof-completed-regexp "^No subgoals!"
 proof-shell-eager-annotation-start
 "^\\[opening \\|^###\\|^Reading")

(provide 'demoisa)
```

B.2 demoisa.el

```
;; demoisa.el Example Proof General instance for Isabelle
;;
;; Copyright (C) 1999 LFCS Edinburgh.
;;
;; Author: David Aspinall <David.Aspinall@ed.ac.uk>
;;
;; $Id$
;;
;; ===================================================================
;;
;; See README in this directory for an introduction.
;;
;; Basic configuration is controlled by one line in `proof-site.el'.
;; It has this line in proof-assistant-table:
;;
;;     (demoisa "Isabelle Demo" "\\.ML$")
;;
;; From this it loads this file "demoisa/demoisa.el" whenever
;; a .ML file is visited, and sets the mode to `demoisa-mode'
;; (defined below).
;;
;; I've called this instance "Isabelle Demo Proof General" just to
;; avoid confusion with the real "Isabelle Proof General" in case the
;; demo gets loaded by accident.
;;
;; To make the line above take precedence over the real Isabelle mode
;; later in the table, set PROOFGENERAL_ASSISTANTS=demoisa in the
;; shell before starting Emacs  (or customize proof-assistants).
;;


(require 'proof) ; load generic parts


;; ======== User settings for Isabelle ========
;;
;; Defining variables using customize is pretty easy.
```

```
;; You should do it at least for your prover-specific user options.
;;
;; proof-site provides us with two customization groups
;; automatically:  (based on the name of the assistant)
;;
;; 'isabelledemo        -  User options for Isabelle Demo Proof General
;; 'isabelledemo-config -  Configuration of Isabelle Proof General
;;    (constants, but may be nice to tweak)
;;
;; The first group appears in the menu
;;   ProofGeneral -> Advanced -> Customize -> Isabelledemo
;; The second group appears in the menu:
;;   ProofGeneral -> Internals -> Isabelledemo config
;;

(defcustom isabelledemo-prog-name "isabelle"
  "*Name of program to run Isabelle."
  :type 'file
  :group 'isabelledemo)

(defcustom isabelledemo-web-page
  "http://www.cl.cam.ac.uk/Research/HVG/isabelle.html"
  "URL of web page for Isabelle."
  :type 'string
  :group 'isabelledemo-config)


;;
;; ======== Configuration of generic modes ========
;;

(defun demoisa-config ()
  "Configure Proof General scripting for Isabelle."
  (setq
   proof-terminal-string ";"
   proof-script-comment-start "(*"
   proof-script-comment-end "*)"
   proof-goal-command-regexp    "^Goal"
   proof-save-command-regexp    "^qed"
   proof-goal-with-hole-regexp  "qed_goal \"\\(\\(.*\\)\\)\""
   proof-save-with-hole-regexp  "qed \"\\(\\(.*\\)\\)\""
   proof-non-undoables-regexp   "undo\\|back"
   proof-undo-n-times-cmd "pg_repeat undo %s;"
   proof-showproof-command "pr()"
   proof-goal-command "Goal \"%s\";"
   proof-save-command "qed \"%s\";"
   proof-kill-goal-command "Goal \"PROP no_goal_set\";"
   proof-assistant-home-page isabelledemo-web-page
   proof-auto-multiple-files    t))


(defun demoisa-shell-config ()
```

```
  "Configure Proof General shell for Isabelle."
  (setq
   proof-shell-annotated-prompt-regexp   "^\\(val it = () : unit\n\\)?ML>? "
   proof-shell-cd-cmd "cd \"%s\""
   proof-shell-interrupt-regexp         "Interrupt"
   proof-shell-error-regexp "\\*\\*\\*\\|^.*Error:\\|^uncaught exception \\|^Exception-
   proof-shell-start-goals-regexp "Level [0-9]"
   proof-shell-end-goals-regexp "val it"
   proof-shell-proof-completed-regexp   "^No subgoals!"
   proof-shell-eager-annotation-start   "^\\[opening \\|^###\\|^Reading"
   proof-shell-init-cmd  ; define a utility function, in a lib somewhere?
   "fun pg_repeat f 0 = ()
      | pg_repeat f n = (f(); pg_repeat f (n-1));"
   proof-shell-quit-cmd "quit();"))



;;
;; ======== Defining the derived modes ========
;;

;; The name of the script mode is always <proofsym>-script,
;; but the others can be whatever you like.
;;
;; The derived modes set the variables, then call the
;; <mode>-config-done function to complete configuration.

(define-derived-mode demoisa-mode proof-mode
    "Isabelle Demo script" nil
    (demoisa-config)
    (proof-config-done))

(define-derived-mode demoisa-shell-mode proof-shell-mode
   "Isabelle Demo shell" nil
   (demoisa-shell-config)
   (proof-shell-config-done))

(define-derived-mode demoisa-response-mode proof-response-mode
  "Isabelle Demo response" nil
  (proof-response-config-done))

(define-derived-mode demoisa-goals-mode proof-goals-mode
  "Isabelle Demo goals" nil
  (proof-goals-config-done))

;; The response buffer and goals buffer modes defined above are
;; trivial.  In fact, we don't need to define them at all -- they
;; would simply default to "proof-response-mode" and "pg-goals-mode".

;; A more sophisticated instantiation might set font-lock-keywords to
;; add highlighting, or some of the proof by pointing markup
;; configuration for the goals buffer.
```

```
(provide 'demoisa)
```

Function and Command Index

D

P

Variable and User Option Index

C

I

O

P

Concept Index

Table of Contents