User Guide to the CTCOQ Proof Environment

Janet Bertot, Yves Bertot, Yann Coscoy, Healfdene Goguen, Francis Montagnac

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

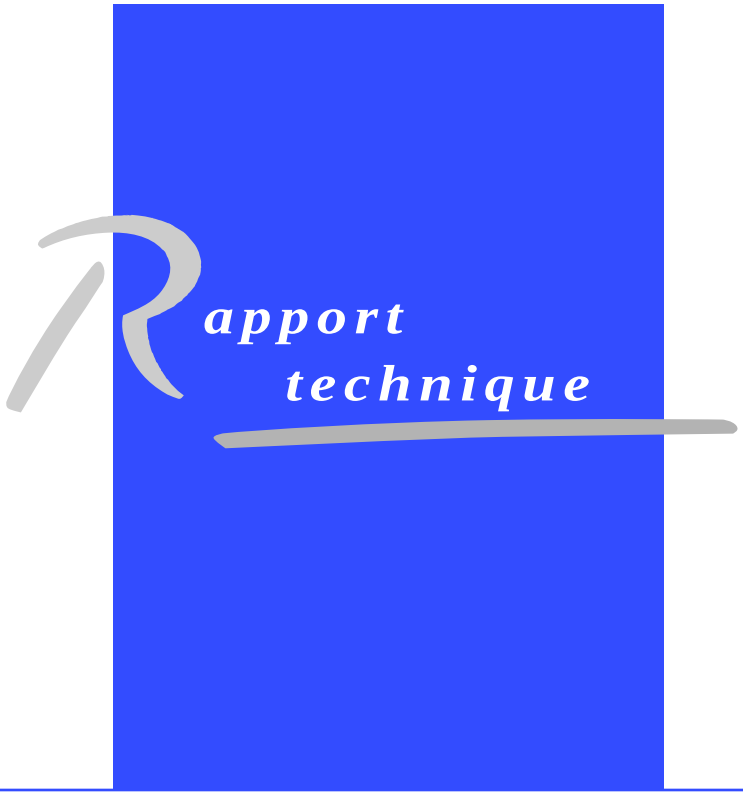
User Guide to the CTCOQ *Proof Environment*

J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac

N˚ 0210

Octobre 1997

THÈME 2



ISSN 0249-0803

User Guide to the CtCoq Proof Environment

J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen* , and F. Montagnac

Thème 2 — Génie logiciel
et calcul symbolique
Projet CROAP

Rapport technique n ° 0210 — Octobre 1997 — 63 pages

Abstract: CtCoq is a graphical environment to perform proofs in Coq.

Key-words: CtCoq, Coq, proof system

(Résumé : tsvp)

* with support from NSF

Unité de recherche INRIA Sophia Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex (France)
Téléphone : 04 93 65 77 77 – international : +33 4 93 65 77 77 – Fax : 04 93 65 77 65 – international : +33 4 93 65 77 65

Guide utilisateur pour l'environnement de preuve CtCoq 1.23/1.9/1.36

Résumé : CtCoq est un environnement pour effectuer des preuves en Coq.

Mots-clé : CtCoq, Coq, système de preuve

Contents

1 Introduction

The Coq system[Coq96] is a proof assistant based on typed theory and the calculus of constructions. It enables users to interactively state mathematical results and their proofs and assists in checking the validity of these proofs. Among other features, it provides facilities for goal directed proofs, where the user states a logical formula to be proven (a goal), and applies commands that break down the proof of this formula into the proof of several, presumably simpler, formulas. Performing goal directed proofs is one of the main activities of users of this system.

The user-interface described in this paper aims at providing additional support to the users of the Coq system. The currently supported activities are the following ones:

- Visualizing and editing command files. The tools for supporting this activity are syntax directed editing windows. These windows recognize the structure of the command language for the Coq system. It makes it possible to ensure the syntactic validity of commands, to highlight important keywords and definitions, to provide notational extensions with graphical symbols and colors, and to provide sophisticated command generation depending on the current state of the proof system.

- Visualizing the current state of the theorem prover during a session with the Coq system. Tools are provided to browse through the object databases of the theorem prover. For example, users can open windows displaying all the theorems related to a specific predicate.

- Performing or replaying goal directed proofs. The current version is especially oriented to supporting this activity. During goal directed proofs, the CtCoq user interface provides help for visualizing the current set of goals, quickly generating new commands, and maintaining a clean and coherent proof script.

- Explaining proofs in a pseudo natural language. Complete and incomplete proofs can be printed in a form that highlights the structure of proofs and can be understood by non-specialists of the Coq system.

The files manipulated or generated using the CtCoq interface are written in the regular command language of the Coq system. In this respect, using this interface does not preclude working with the Coq system alone, or collaborating with people who don't use the interface.

The current version of the CtCoq system is not complete in several respects. A basic list of needed developments contains the following points:

1. Support for state management, undoing, or saving backups. While commands of the family of `Save State` are treated, they are not completely integrated in the maintenance of the proof script (i.e., archive coherency).

2. Support for managing several goal-directed proof attempts at a time. Several proofs may be attempted at any one time, it is the script maintenance that is not fully integrated (i.e., archive maintenance). That is, one can incorporate lemmas and theorems developed in separate windows into a single window but this is not done automatically.

3. Support for windows related to the hint database used for automatic theorem proving.

4. Support for extracting algorithms.

5. Support for syntactic extensions.

This guide is organized as follows. Section 2 describes a sample session with the system. Sections 3 to 10 dwell on more specific aspects related respectively to the various kind of windows present in the interface, syntax directed editing and notation extensions, executing commands in Coq, mouse interaction during proofs, input and output formats, on-line help, configuring the system, and safety features.

Instructions to install the whole CtCoq system are given in Appendix A, and Appendix B gives the initial mouse and keyboard bindings for the system.

2 An Example Session

In this section, we give a little demonstration of the various functionalities of the interface, around the proof of a basic arithmetic theorem:

$$\forall x, y, z : nat. x \leq y \wedge y < z \Rightarrow z > x$$

2.1 Starting the Whole System

To start the CtCoq interface, you need to have X already running on your graphical workstation. We also advice that you have a Netscape[1] window iconified in a corner

[1] Netscape is a trademark of the company that owns it.

of your screen (it will be used to provide on-line help). You can then start the system by typing the following command in a shell window.

```
$ ctcoq
```

Two windows are created. One is called the *Centaur messages* window, and the other is a multiple window called *Command + State + Search Theorems*. The Centaur messages window, which is labelled "Centaur (ctcoq)", is used to display error messages. Through this window's menu bar, however, one can open windows to view the contents of files or view the clipboard, etc. The multiple window is composed of three sub-windows:

Command (at the top of the multiple window). This window is used to edit and record the commands sent to the proof assistant.

State (in the middle). This window is used to display the current list of subgoals during goal directed proofs. This window is initially empty.

Search Theorems (at the bottom). This window is used to display the result of queries sent to the object database of the proof system. Typically, results of the command `Search` will be displayed in this window. The *Search Theorems* window is initially empty.

Note that it is in the multiple window that proofs are developed. While only one such multiple window is opened when the system is started, it is possible to open several of them (for more details see section 3.2).

2.2 Editing and Sending Commands

To prove our basic arithmetic theorem, we first need the arithmetic package to be loaded in the proof assistant database. To perform this we need to input the command `Require Arith.` and to send it to the proof assistant.

Commands for the proof assistant are edited in the *Command* window and sent using the buttons that appear between this window and the *State* window or keyboard accelerators. The window and its decoration provide various facilities for editing. Typically, the user selects expressions inside the window and triggers editing actions using either the options of the `Edit` menu from the menu bar, menus provided by the `Editing-Tools` menu from the menu bar, or actions associated to keystrokes while the mouse is in the window. In this example session, we shall only demonstrate plain text editing.

The word `COMMAND` that appears in the top left corner represents a place-holder for a command. Type `Require Arith.<Escape>`. At the first character you type the word `COMMAND` disappears and the text you type appears on the screen. When you type `<Escape>` this text changes aspect: keywords are high-lighted and the word `Import` is inserted. The `<Escape>` key has triggered the parsing of the text you type: the text has been replaced by a tree structure.

If you discover an error after parsing, you can correct specifically the wrong sub-expression by clicking on it and type `<Escape> t`. The expression is transformed back into text that you can modify and reparse.

Execute the command by selecting it with the left mouse button and clicking on the Do it button provided below the *Command* window, or by typing `^C g`. This may take a while and during that time the command appears on pink background. When it finishes, the content of the *Command* window is modified again. The executed command changes its appearance (the background becomes light brown) and a new place-holder `COMMAND` is inserted. The darker background indicates that this command is now recorded in the script.

The user can now edit the new place-holder to state the theorem that he or she wants to prove. If this place holder is not selected (grey background), select it by clicking on it with the left button of the mouse and type

```
Theorem my_theorem :
     (x,y,z:nat)((le x y) /\ (y < z))-> (gt z x).<Escape>
```

As before, the `<Escape>` character will not appear on the screen, but provoke this command to be parsed. After parsing, the logical formula is presented with more typical mathematical conventions, using special characters like a quantifier or an arrow.

Once the command is complete, it can be executed by clicking on the Do it button. This extends the recorded script (light brown area) to this command and the *State* window receives the statement of this theorem as a statement to prove.

2.3 Performing Goal Directed Proofs

It is possible to proceed with the proof by entering all the commands by hand. However, it is also possible to use the functionality of *proof-by-pointing* to accelerate this task. The user can simply select any expression in the *State* window and the user-interface generates the command that most naturally brings this expression into focus. This command is then inserted in the *Command* window, at the position

designated by the current selection (gray background). Of course, this only works if the current selection designates a position that can receive a command.

Here, the next command to perform the proof can be generated by simply clicking on the expression `(z > x)` in the *State* window with the right button of the mouse. A command is inserted in the *Command* window. Clicking on the Do it button sends this command to Coq and the user-interface updates accordingly: the command is included in the brown area and the goal changes. To avoid having to click on the Do it button after each interaction, you can also check the square box that appears on the left of the "Do it" to enter automatic-do-it mode, where complete commands are automatically sent to the Coq system. You can also switch the the Automatic-do-it mode on and off by typing `^C t`.

The new content of the *State* window shows a goal composed of two parts, separated by an horizontal line. The top part is a statement to prove, also called the *conclusion*. The bottom part of the goal is a local context of named objects and the *hypotheses* that can be used to prove the conclusion.

2.4 Browsing through the Theorem Database

Having to prove a statement of the form `(z > x)` we want to see all the theorems of the database that permit proving this kind of statement. This can be done by selecting `(z > x)` with the left button of the mouse and typing "`^C s`" while the mouse is still in the window. The letter `s` stands for `Search`, which is a command provided by the Coq system for looking through the object database for theorems related to specific types, functions, or predicates. The output of this search is displayed in the *Search Theorems* window. Note that the search works from any of the three windows, *Command*, *State*, and *Search Theorems*.

Proof-by-pointing also works in the *Search Theorems* window.

1. Scroll the window down a little to make `gt_le_trans` appear, or type `<Escape> s` with the mouse in the *SEarch Theorems* window and type `gt_le_trans` in the dialog that appears and click Ok.

2. Select the name of theorem `gt_le_trans` with the left button of the mouse,

3. Depress the control key, and with this key still depressed, click with the right button of the mouse: this makes a little menu appear in which you can drag the mouse to the Apply option[2].

[2] Please do not release the control key before you finish this operation. If you do, the menu sticks to the screen, to make it disappear you simply click on the Cancel button at the top of the menu.

Instead of step 3, you can also type `a` (for `Apply`), after doing step 2 above, while the mouse is in the *Search Theorems* window.

The generated template is not complete, however, and the selection in the *Command* window is on a place-holder named `Value_for_m`. Such a value is needed the apply command is low-level and does not guess the value of the variable `m`. The user-interface will refuse sending the command as long as it contains any such empty place-holders.

2.5 Moving Values Around

In our ongoing example, the relevant value for `m` is `y`. To edit the place-holder, it is possible to use text editing or the options of the `Edit` menu from the menu bar. It is also possible to use a faster functionality, provided by the middle button in all the windows of the user-interface. When an expression is selected with the middle button, the user-interface attempts to replace the expression currently selected in the *Command* window with the new value. Of course, this operation is performed only if the new value fits with the syntactic constraints in the context where it is pasted.

Here, the user simply has to make sure the place holder is selected (if needed, using the left button of the mouse), and select any instance of `y` in any window, using the middle button of the mouse.

The now complete command is sent immediately to the proof assistant if you are in automatic-do-it-mode, otherwise you have to click on Do it . This provokes the content of the state window to change: there are now two subgoals.

2.6 Undoing a step

You may have made a mistake and executed the wrong command. Undoing this will require more than just undoing the last editing action: you also need to tell Coq to discard the last command. This is done by clicking on the Discard button that appears below the *Command* window. You can then select the wrong expression and re-edit it. If you don't discard the command, re-editing will not be possible: the light brown area is *read-only*.

2.7 Proof-by-Pointing Continued

The second subgoal is straightforward, since one of the hypotheses (`H'`) is a conjunction containing the statement to prove (`x` $\leq$ `y`). Generating the command to break

this conjunction into pieces and complete the proof is simply performed by pointing to the expression `(x ≤ y)` that occurs behind "`H':`" in the second subgoal to express that you want to use this part of the conjunction to perform the proof. Of course, this has to be done using the right button of the mouse and a new command is inserted in the *Command* window. The goal is solved and disappears from the *State* window. The other subgoal can be proven similarly by clicking on `(y < z)` with the right button of the mouse because `(z > y)` and `(y < z)` are viewed as the same statement by the theorem prover (they are δ-convertible).

2.8 Storing the Theorem

When the proof is finished, there remains a place-holder at the end of the script that can be edited as text to `Qed`, or one can simply type `^C v` (for "saVe") in the *Command* window, and one can then execute this command. You can check that your new theorem is added by using the `Search` command (with the key binding `^C s`, when the selection is on `>`).

2.9 Displaying the Proof Text

Once the proof is finished and saved, it is possible to produce a textual explanation by selecting the name `my_theorem` wherever it may appear (in the *Command* or in the *Search Theorems* window) and typing `^C p` while the mouse is still in the same window. The production of this textual explanation takes a while and this eventually provokes the creation of a new window named *Proof text*. A pseudo natural english explanation of the proof is displayed in this window. There, it is also possible to view the proof text of theorems like `gt_le_trans` by clicking on that name and typing `^C p` again[3]

2.10 Retracing our steps

It is possible to remove a proven theorem or a definition from the Coq database, making use of the Coq command `Reset`. In the CTCOQ system, this command is supported by a simple combination of a mouse click and a keystroke. In our example, to cancel the proof of `my_theorem` it is only necessary to click on `my_theorem` or the corresponding `Qed` statement and type `^C r` (for `Reset`), while the mouse is still

[3]This part of the system is less robust than the rest. It may happen that the system complains about a `serious protocol error`. If this happens, please close the window by clicking on the square box in the top left corner.

in the *Command* window. This operation has two effects. One is not visible and corresponds to the removal of all proofs and definitions that occurred chronologically after the proof of `my_theorem` from the Coq system's database, the other is the change of aspect of all the commands occurring in the *Command* window after the command that start the proof of `my_theorem`: these commands switch from a light brown background to a pale background. This change of aspect expresses that these commands are no longer part of the recorded script. They can now be edited at will and eventually replayed.

In our example, we shall simply resend to the Coq system all the commands except for the `Qed` command. To do so, we have to select all these commands by simply dragging the mouse over these commands while keeping the left button depressed. We can then click on the Do it button, or type `^C g`. The selected commands change their appearance: most of them turn yellow, while a pink area moves down the selected commands as they are processed by the Coq system. After being processed, commands are stored in the brown area. While the proof is not saved, that is as long as the `Qed` command is not sent to the proof system, it is also possible to tell the Coq system to undo goal-directed commands one by one. To do so, it is only necessary to click on the Discard button or by typing `^C _`. Note that the Abort button can be used to discard several goal directed commands in one step. The keyboard accelerator `^C k` is associated to this button.

NB: The Discard button can only undo commands for which **undoing** *makes sense in the Coq system. One can* **undo** *the commands that define objects, one can undo a goal directed command as Coq supports this as well, however, one cannot undo a* `Require` *command, which has loaded information in the database and for which Coq does not support a clean removal of this information.*

2.11 Wrapping it all up

At this point, the script of the proof is only stored in the interface. To save it on disk, one can use the `Save` option of the `File` menu. The interface can then be terminated by using the `Quit` option of the File menu. This also automatically terminates the Coq system, by sending a `Quit` command.

3 Window Structure

The various windows on the screen are Centaur *editors*, also called *Ctedit* windows. These windows are decorated with buttons and menus that provide help to the user

for browsing their contents and editing it. The use of these buttons is described in the Centaur documentation[Centaur94], chapter "The User Interface Manual". Especially useful functionalities are those provided by the `Edit` pulldown (structure-oriented editing), the `Selections` pulldown (highlighting and elision mechanism), and for the *Command* window, the `File` pulldown (loading and saving files) and the `Editing-Tools` pulldown (menu-directed editing). Important options of this menu are the options `Beginner`, `Expert`, and `Add box`.

3.1 The windows of CtCoq

The CtCoq interface provides seven kinds of windows. The *Command* window supports editing and script maintenance. The *State* window displays the current list of goals during goal directed proofs. The *Search Theorems* window supports theorem database visualization. These three windows are grouped in a *multiple* window, present on the screen from the beginning of the working session. The user can also open new multiple windows, by choosing the option New Proof Window from the File menu or by typing `^C d` (*for duplicate window*) while the mouse is in the *Command* window. It is also possible to keep the contents of the *Search Theorems* window in a *Library Fragment* window by typing `^C d` (*for duplicate window*) while the mouse is in the *Search Theorems* window. A *Proof text* window can be created when the user requests the text of the proof of a theorem (by typing `^C p` for *Print* in a window where a theorem's name is selected) or when the user requests the text of the current proof (by typing `^C q` in the *Command* window). A *Goal* window can also be opened by typing `^C h` while the selection is on an executed command of an unfinished proof and the mouse in the *Command* window.

Last, the system also provides a *Centaur Messages* window, where error, warning, or information messages are displayed to help users understand the success or failure of their activity.

For visualizing and editing commands, users will mostly work with *Multiple* windows.

3.2 Groups of windows

The three windows that appear in a multiple window form a sub-group of windows, where the *Command* window has a privileged status. The *Library Fragment* windows created from the *Search Theorems* window also belong to this group. Unfortunately, there is no visual clue indicating to which multiple window a *Library Fragment* window is attached.

New groups are created when using the option New Proof Window from the File menu of a multiple window (or typing `^C d`) in the *Command* window. These groups are also organized in a hierarchical way, where each group is considered a parent of all the groups created from it. This notion of parent is used when using the option Move Thm/Defn to previous of the File menu. Note that the window present at the beginning does not have a parent. For this reason, it also does not have the Move Thm/Defn to previous option or the *kill box* that makes it possible to close it. In the following, we shall call this multiple window the *Master* window.

The *Centaur Message* window, the *Goal* window, and the *Proof* window are not attached to any sub-group, as are the windows created using the option Open in New ... of the `File` menu of the *Centaur Message* window.

3.3 the *State* window

In the *State* window one sees the various subgoals that need to be proven. In this window, goals are displayed in a different order than the order used in the teletype interface of coq. In coq, they are printed with the assumptions appearing on top of the conclusion, in CtCoq they are displayed with the conclusion appearing on top of the assumptions. There are good reasons for choosing a different setup in the graphical user-interface: old assumptions thus appear in the bottom of the window and have a tendency to disappear, while old data has a tendency to disappear towards the top on a regular alpha-numeric terminal.

One can move about from subgoal to subgoal using either the buttons in the space between the *Command* and *State* window, or using the keyboard accelerators described below when the mouse is in the *State* window.

`^N`. *Next Subgoal*. Changes the current subgoal to the next one. The current subgoal needs to be specified when one uses *proof-by-pointing* from the *Search Theorems* window or from a *Library Fragment* window. This also provokes a scroll of the *State* window to the new current subgoal.

`^P`. *Previous Subgoal*. This changes the current subgoal to the previous one.

3.4 Multiple Proofs in Progress

CtCoq initially opens one multiple window that is composed of three subwindows: the *Command*, *State* and *Search Theorems* windows. These three areas work in conjunction when one is developing a proof. However, imagine one is working on a

proof and suddenly realizes that a lemma is needed. One would like to go back and add the lemma's statement and its proof before the statement of the current proof. However, one cannot add something in the middle of the proof script. The executed commands of the current session appear in light brown in the *Command* window. CtCoq allows one to develop multiple proofs in progress but in order to have a coherent view of the proof engine, the system assumes that each multiple window has **one** proof in progress at any given time.

3.4.1 Additional Proof Windows

To begin proving a `Lemma` when one is already working on a proof, one opens a new multiple window via the *Command* window by clicking on the `New Proof Window` button in the `File` menu. A new multiple window composed of a *Command* window, a *State* window, and a *Search Theorems* window appears on the screen.

Once completed, the `Lemma` and its proof may be incorporated into the proof script of the initial multiple window, either by selecting the `Lemma` statement or its corresponding `Save` (or `Qed`) command with the left mouse button, and then clicking on the button `Move Thm/Defn to Previous` in the *Command* window's `File` menu. This results in the `Lemma` and all commands up to and including the `Save` being moved to the initial *Command* window and placed just before our `Theorem` in progress.

Of course, in the middle of proving our `Lemma` above we may have discovered that we were missing an `Axiom`. We may open yet another proof window from either of the *Command* windows in our two proof windows. However, the objects defined within a proof window always incorporated into the proof window from which it was created.

The State of the Proof Engine

Using multiple proof windows one can develop several proofs in parallel. That is, execute one command, or proof step, from the initial proof window, then a command from our new proof window, and then alternate back and forth between the two. While this is not the situation that initially lead to the introduction of several proof windows, it is possible to completely interleave the proof steps of different proofs.

If one selects several commands to be executed from two different command windows. You noticed that the execution is interleaved, first a command from one window, then a command from the second, then another command from the first, etc. The *script* areas (in light brown) grow with each executed command. The *tentative* areas (in pink) represent the command from each window that was sent to

the proof engine —of course, only one of these is actually being executed, however the others have been sent and we are awaiting an acknowledgement corresponding to the command's successful completion. The *buffer* area (in yellow) behaves as usual, stacking commands to be sent to the theorem prover.

There are a few changes with respect to the `Reset` command. A `Reset` of an object removes all objects defined later chronologically and all proofs in progress are aborted. Thus, a `Reset`, including those generated via the `Discard` button, may update several windows, removing from the *script* areas all commands executed after the defining of the object which we are resetting.

3.4.2 Shortcomings of the Multiple Proofs Mechanism

There are a couple of shortcomings in this first version. The fact that window contents are not automatically incorporated and must be done manually was mentioned above. Also, while the additional proof window contents are checkpointed, one could imagine asking if the user wants to incorporate these objects when one closes the proof window or tries to read in a file in the proof window.

3.5 Error Messages and Coq output

The error messages displayed in the *Centaur Messages* window have two origins. The main source is the user-interface itself; a second source is the Coq proof assistant running in the background. As a general rule, the messages of Coq are displayed without corruption by CtCoq. However, there are cases when the display is incomplete.

When necessary, it is possible to see the exact output from Coq, by having this output printed directly in the standard output of the CtCoq command, that is, in the `shell` window where you started the system. This makes uses of a *logging* facility, which can be turned on and off using a command or a resource.

- The command to turn on logging is `(coq-log t)`. this text must be typed in the `shell` window where you started CtCoq. If later on you want to turn this back off, type `(coq-log ())`. To know whether the logging facility is on or off, type `(coq-log)`.
- The resource to set is `Centaur.Start-data.Log`. To have this resource set in all your sessions, you simply need to insert the following line in your resource file:

```
Centaur.Start-data.Log: t
```

This feature can also be used to view the output of commands that are still not supported by CtCoq, like `Print LoadPath`, `Print Natural Implicit`, `Print Natural Contractible`, `Print Natural Transparent`, `Inspect`, and `Print`.

4 Visualizing and Editing Coq Commands

4.1 Navigating in structured text

The CtCoq system helps in the understanding of the structure of objects at various levels. A first instance of aid comes through the formatting engine, which takes care of indenting lines in a way that respects the structure, highlighting keywords and comments. The user can also point at some expression with the left button and see the whole expression highlighted. This enables him or her to understand the structure a bit better when there are too few, or too many, parentheses.

4.1.1 Navigation keys

The user can also navigate in the tree structure, using the following key bindings:

`^B`, *left*. Move the current selection to the left brother of the currently selected tree.

`^F`, *right*. Move the current selection to the right brother of the currently selected tree.

`<Escape> n`, *next-meta*. Move the current selection to the next empty place-holder in the window.

`<Escape> p`, *previous-meta*. Move the current selection to the previous empty place-holder in the window.

`<Escape> ^U`, *up*. Move the current selection to the father of the currently selected tree.

`<Escape> ^D`, *down*. Move the current selection to the first child of the currently selected tree.

4.1.2 Shrinking and expanding

Another useful feature to help understand the structure of large objects is the possibility to compress parts of these objects in a selective manner. The user can request to hide this sub-expression by selecting it and choosing the option `Shrink` in the `Selections` menu provided in the window's menu bar. The selected sub-expression will then be replaced by the short string "...". Of course, it is possible to make that sub-expression reappear by choosing the `Expand` option in the same menu. The windows *State* and *Search Theorems* do not have a menu bar attached, but the corresponding menu can still be had by depressing the *shift* key from the keyboard and depressing the *right* button of the mouse while the *shift* key is still depressed.

The user can actually specify a printing depth that forces all sub-expressions at a given depth in the structure to be compressed and replaced by the short string given above. This printing depth parameter can be modified at will using the `Set Print Level` option of the `Display` menu from the menu bar. For the windows *State* and *Search Theorems* this menu can be had by depressing the *shift* key from the keyboard and depressing the *middle* button of the mouse while the *shift* key is still depressed.

4.1.3 The gallina sub-language

Sometimes the user would like to see just the definitions and theorem statements rather than the complete file with the proof steps. Basically, one would like to view the current session (or file) as a *Table of Contents*. By clicking on the `Table of Contents` button in the `Display` menu, a new window is opened that shows the contents of the *Command* window without the proof steps. This window is just a different pretty printing of the *Command* window's contents and, in fact, you may copy and paste (particularly using the middle mouse button) to the *Command* window.

4.2 Formulas and Commands Editing

The user can edit the contents of windows by moving sub-expressions around, while respecting the syntactic constraints of the language. Basic editing operations are provided by the options of the `Edit` menu or the options in the editing box that can be obtained by choosing the option `Add box` in the `Editing-Tools` menu from the window's menu bar. There is a wide palette of possible actions that respect the language's syntactic constraints.

4.2.1 Keyboard Controlled Editing Commands

In the *Command* windows, apart from the navigation commands already presented in the previous section, it is also possible to trigger several structured editing commands by using keys from the keyboard:

`<Return>`, *insert-meta-after*. When the current selection is the member of a list, insert a new place-holder after the current selection and move the current selection to this place-holder.

`^O`, *insert-meta-before*. When the current selection is the member of a list, insert a new place-holder before the current selection and move the current selection to this place-holder.

`^W`, *kill-list-element*. When the current selection is the member of a list, remove this member from the list and store it in the clipboard.

`<Escape> w`, *copy*. Store a copy of the current selection in the clipboard.

`^Y .`, *change* replace the current selection with the contents of the clipboard.

`^Y <`, *insert-before*. When the current selection is the member of a list, insert the contents of the clipbard after it.

`^Y >`, *insert-after*. When the current selection is the member of a list, insert the contents of the clipbard after it.

4.2.2 Menu Guided Editing

Syntax directed editing is also supported by *menu-guided* editing, which enables the user to insert templates for commands by choosing these templates from a menu. Such a template menu can be opened by choosing the option `Beginner` or `Expert` in the `Editing-Tools` menu from the window's menu bar. The content of this template menu, also called a *Ctmenu*, is updated each time one moves the selection in the window so that only templates which respect the syntactic constraints of the command language are proposed to the user.

There are three kinds of options presented in a *Ctmenu*. Some options are simple templates and make it possible to replace a place holder for a command by a template for another command. A second group of options act more as rewrite rules. They enable the user to replace the currently selected expression by a new expression that contains some components of the old one. The third group are *submenus*. The

submenus are visualized by a label followed by ..., clicking on the ... replaces the contents of the *Ctmenu* window with the submenu's options. Users can choose one of these new options, or go back to the previous menu by clicking on the ... of the first item in the submenu.

As a first example, let us consider that the user wants to replace the formula `(P x)` with `∀x:nat.(P x)`. This is simply done by selecting the expression `(P x)`, clicking on the option `forall` in the template menu, which produces a formula `∀IDENT:FORMULA.(P x)`, replacing `IDENT` by a copy of `x`, which can be done by clicking on `x` using the middle mouse button, and filling in the type for the formula (several predefined types appear in the submenu `Predefined Types/...`). The `forall` option of the template menu is actually a rewrite rule of the following form, where `*x` represents the component that will be reused:

```
*x -> ∀IDENT:FORMULA.*x
```

As a second example, let us consider that the user wants to replace the formula `a ∧ b` with `a ∨ b`. This is simply done by selecting the expression `a ∧ b` and clicking first on the submenu `or/...`, followed by clicking on the entry `and -> or` in the *Ctmenu*. This produces the desired formula. The `and -> or` option of the template menu is actually a rewrite rule of the following form, where `*x` and `*y` represent the components of the destroyed formula that are reused in the new one:

```
*x ∧ *y -> *x ∨ *y
```

4.2.3 *Command* Copy-Paste Shortcut

In most windows the middle button has been reserved to provide a Copy-Paste shortcut to the *Command* window. If the user selects with the left button a position (p) to modify and with the middle button an expression (e) to copy, then this expression (e) is put at that position (p). This behavior is simple to understand if one views the window where the middle button is used as a template menu for the `Command` window.

4.2.4 X Copy and Paste

The X-window system also provides functionalities for moving text between windows from various applications. Our windows are compatible with this protocol, although the bindings with mouse buttons are changed with respect to the usual X-window conventions. To copy the text representation of a term in the inter-application buffer of X-window, users only need to depress the shift key of the keyboard while selecting this term with the left button of the mouse (this is different from the usual X-window

convention, where no key modifier is used). To replace a term with the contents of the X-window buffer, users only need to depress the shift key of the keyboard while selecting this position with the middle button of the mouse (here again, the usual X-window convention does not use any key modifiers).

Using these functionalities, it is very easy to copy and paste data between the CtCoq interface and any other text editing tool. However, this communication mechanism does not yet work when performing text editing in the interface.

4.2.5 Bugs and caveats

For historical reasons, the CtCoq interface does not completely respect the selection protocol advocated in X, and uses the obsolete copy-paste buffer method. For this reason, the X copy paste mechanism will not work when communicating data with applications that do not handle selections in a backward compatible way. Instances of applications that communicate without any problem are Gnu-Emacs and xterm. Instances of application that do not are Tcl-Tk and derivatives and Netscape.

4.2.6 In-Place Text Editing

Text editing is provided in the *Command* window as soon as the user types a printable ascii character, or after explicitly entering this mode by typing `<Escape> t`. All the key bindings for this window disappear until the window returns to the standard behavior.

When entering text-editing mode, the selected tree is replaced by a text area containing its textual representation; the characters appear on a green background, the filler at the end of lines is dark grey. For formulas, this also means that special characters, such as quantifiers and logical connectives, are replaced by the corresponding ascii conventions, coming back to the regular syntax of the Coq proof assistant. A cursor (in light grey) is also placed at the beginning of this text area.

The text editor is rudimentary. It provides a few key bindings similar to the bindings usually found in the *Emacs* text editor:

`^A`, go to the beginning of the current line,

`^E`, go to the end of the current line,

`^B`, go backward one character,

`^F`, go forward one character,

`^P`, go up one line,

`^N`, go down one line,

`^H`, (backspace) delete previous character,

`^?`, (delete) delete previous character,

`^D`, delete current character; at the end of a line, delete the line feed, i.e., concatenate next line to the current one,

`^K`, delete all characters to the end of line; at the end of a line delete the line feed, i.e., concatenate next line to the current one,

`Escape`, terminate text editing. Try parsing the text:

- in case of success, replace the edited tree by the new value,
- in case of failure, leave the text area unchanged for correction,

`^G`, abort text editing; just restore the previous term.

`^C g`, send to coq. This commands provokes the command containing the current text area to be sent to Coq, after parsing the text area. With this command, it not necessary to type `<Escape>` to exit text editing before sending a command to Coq, and there may be several text areas in the command, which will all be parsed.

Up to now, the possibilities provided to add key bindings are undocumented. It is also possible to move the cursor around by pointing with the mouse.

4.2.7 Comments

The parser discards all comments that it sees when parsing a file, except those comments that are attached to toplevel commands. Thus, users of CtCoq should put comments only in these places. In the interface, comments are displayed in variable width font and without the usual comment delimiters. Several comments preceding a toplevel command will be transformed into one single multiple-line comment.

To add a comment on a command, you should proceed as follows:

1. select the command,

2. enter textual editing mode, by typing the comment, starting with the opening delimiter "(*" and finishing with "*)",

3. provoke a parsing of the command by typing `<Escape>`. After parsing, the comment will appear almost as you typed it in, with delimiters removed. However, when you save the script on disk, the comments will be printed correctly with delimiters.

4.2.8 Bugs and caveats

As a general rule, text editing should be restricted to small fragments of text, not exceeding two or three lines.

This text editing facility does not interact very well with structured editing. The user should avoid using the `Edit` menu, the tools provided by the `Editing-Tools` menu, or the middle button of the mouse while editing a subexpression as text. For future versions of the interface, we plan to enhance the interaction between the editing modes.

If the user-interface gets into a state where text editing causes a problem, it is possible to quit the current text editing mode by typing `^G` while the mouse is in the window.

In the current version of CtCoq *most subexpressions* of commands and tactics can be parsed. There are a few tactics, for which even the simpliest subexpression cannot be parsed due to the underlying structure. In these cases, one must select the whole tactic in order to do in-place text editing. We intend to remove this restriction in future versions.

There is one problem that may arise when using text editing that we would like to address here. Suppose that you've selected one element in a list, e.g., one of the constructs in an `Inductive Definition` that you would like to edit `A:B` and rather than changing `A` or `B`, you proceed to enter another element in the list. For example, the text zone (in green) now contains: `A:B | C:D`. When you go to leave text editing mode, by typing `<Escape>`, parsing will fail as we try to parse one element of a list not a sublist. Note that we remain in text editing mode so that none of the text is lost. However, to recuperate and successfully parse the sublist, which may be arbitrarily long, you must copy the textual representation into the `Clipboard` and then insert it into the textual representation of the whole list. This is done as follows:

- Once parsing has failed, select the whole text editing area (in green) using the left mouse button. This area should now appear in light grey.

- In the `Edit` menu, select `Copy`. The textual representation now appears in the `Clipboard` (in black and white). To verify this, select the `Clipboard` entry in the `File` menu in the *Centaur messages* window.
- Next abort the text editing in progress using `^G`.
- Select the list rather than a single element of the list, by simply typing `<Escape> ^U` for "Up" and pass into in place text editing mode by typing `<Escape> t`. Now the whole list appears in its textual representation (on a green background).
- Insert the contents of the `Clipboard` in the text editing area by first selecting the desired position using the left mouse button, and then using either the button `Insert <` or the button `Insert >` from the `Edit` menu. This inserts the `Clipboard` contents either before or after the selected position.

While the process is a bit long to describe, it is not too difficult to understand nor to execute, as we are just manipulating the textual represention of a sublist and trying to attach it correctly in the tree structure. Of course, in future versions we'd like to try to do this automatically, sparing the user from these gymnastics.

4.3 *Adding New Printing Notations*

To add new printing notations, one should write a specification for these notations and indicate to the system that this specification should be used. Basically, you will copy somewhere under your home directory a pretty printing specification that is included in CtCoq to allow for "usernotations", i.e., the user's personal extensions. You will then, using resources files, inform CtCoq to use *your version* of the pretty printer in all future sessions of CtCoq.

The first two steps below get you the files you need; the next step tells CtCoq to look in your directory:

- Create a directory under your home directory where these notations will be stored, say,
  `<your home>/contrib/notations/`,
- Copy all the files from
  `<ctcoq-root>/contrib/vernac/pprinters/usernotations/` to this directory,

- Create the resource file `<your home>/.centaur.rdb` and include the following three resource specifications in this file:
  ```
  Centaur.vernac.ppml.usernotations.Root:  user
  Centaur.vernac.ppml.usernotations.Location:  contrib/notations
  Centaur.vernac.ppml.usernotations.Mode:  compiled
  ```

You are now ready to call a session of CtCoq in which you may modify and test your new notations. Note that once the specification for these new notations are compiled they will be used in all future sessions of CtCoq as long as your Centaur resource file, `.centaur.rdb`, says to look for your version of the *usernotations* pretty printer.

Once you've started a new session of CtCoq, you should open a regular Centaur editor, using the `File` menu of the *Centaur messages* window's menu bar, click on the `New Editor` or the `Open in New ...` button. In this editor read the file: `<your home>/contrib/notations/vernac-usernotations.ppml`.

Add your own rules. You will find a short tutorial on this topic by choosing the option General Help from the `Help` menu and following the link *Extending notations*[4]. You are also invited to get hold of the PPML manual [PPML].

You can then recompile (using the `Compile` button in the `Ppml` menu from the menu bar. That's it. Now to see your specification used in a window that is already on the screen, it suffices to click on the `Redraw` button of the `Display` menu for the window[5]

It is important to note that if you would like to change some fonts or colors for your new notations then you should add the following two resource specifications in your `.centaur.rdb` file:

```
Centaur.vernac.ppml.usernotations.Database.Root: user
Centaur.vernac.ppml.usernotations.Database.Location: \
    contrib/notations/usernotations.rdb
```

Now you may add the resource specifications in this `usernotations.rdb` file. A look at the other pretty printer resource files can give some indication of the types of resources that can be specified.

[4] this link points to the URL `http://www.inria.fr/croap/ctcoq/help/notations.html`.

[5] Recompiling and reloading the pretty printer is only necessary for testing when you've change the usernotation specification. Once in place you may restart a session of CtCoq and your notations are loaded automatically.

While this technique may seem bit complicated to put in place, it does allow the user to have *his* notations in one of *his* directories. One can imagine a more subtle approach where the directories, copying, etc. is done with the click of a button in a running CtCoq. We hope to have this in a future version.

5 Executing commands

The commands of Coq can roughly be sorted into three kinds: the commands that don't change the state of the prover, those that do, and those that make it possible to return to a previous state. The commands that don't change the state of the prover are mainly the commands that make it possible to inspect the state of the proof assistant.

5.1 Viewing the Context

The context is a collection of typed objects. Many commands add new objects in this database. For example, saving a theorem adds this theorem to the context and defining an inductive type adds a collection of typed objects (the constructors, the induction theorems) to the context. In Coq, it is possible to print the entire contents of the context, but this is currently not supported by the CtCoq system. Rather the interface supports a few commands for visualizing the type and value of selected expressions: `Check`, `Search`, `Eval`, and `Compute`. Whenever one of these commands is sent to the proof assistant, the output is displayed in the *Search Theorems* window.

The `Print` command has a special status. If natural language explanation is provided for the requested identifier, then a Proof window is opened with this proof in it (see below in section 5.1.4). Otherwise, this command is silent

5.1.1 Shortcuts

The commands `Check`, `Compute`, `Eval`, and `Search` can be triggered by simply selecting an expression in any window with the left button of the mouse and typing `^C c`, `^C C`, `^C e`, or `^C s`. Note that the case is important: `^C c`, and `^C C` do not behave the same.

The command `Search` takes an identifier as argument. When the user selects an expression that is not restricted to an identifier, the actual argument is the identifier (if any) that is in the function position. For example, selecting `a ∧ b` and typing `^C s` provokes the command `Search and.` to be sent to the Coq system (since `a ∧ b`

is actually a notation for `(and a b)`. For `Check`, `Compute`, and `Eval` however, it is always exactly the selected formula that is selected as the argument.

5.1.2 Using the local context of a goal

The commands `Check`, `Compute` and `Eval` work only for commands whose free variables are bound in the current context. Sometimes, it might be interesting to investigate the type or value of an expression that is valid only in the local context of a goal. This facility is provided by CtCoq through a clever interaction with the Coq system. To use it, you need to select the chosen expression and type `^C ^C c` (for `Check`), `^C ^C C` (for `Compute`), or `^C ^C e` (for `Eval`).

5.1.3 Keeping Carbon Copies

At any time the user can copy the contents of the *Search Theorems* window in a separate window, also called a *Library Fragment* window, by typing `^C d` (for *duplicate window*) while the mouse is in the *Search Theorems* window. This is useful to keep most often used theorems within easy reach. Such *Library Fragment* windows can then be used as menus when editing commands in other windows. This facility lowers the amount of `Check` and `Search` requests sent to the Coq system.

5.1.4 Browsing the natural language presentation of proofs

Setting the current selection on an identifier and typing `^C p` provokes a *Proof text* window to appear on the screen, containing the natural language explanation of the proof denoted by this identifier (the result of the Coq command `Print Natural`). This window is equipped to make navigating between proofs as handy as possible: inside this window, the key bindings for `Check`, `Search`, `Compute`, `Eval`, and `Print Natural` are still valid. Also the window is equipped with two buttons back and forward, that make it possible to travel up and down the series of proof texts displayed so far, in manner very similar to an hypertext browser.

The proofs displayed are cached, so that only one `Print Natural` request is executed in `Coq` for each identifier. However, this cache is not reset if one executes a `Reset` or `Reset Initial` command and the cache may become incoherent with the actual proofs stored in the Coq context. To alleviate this problem, it is possible to force a clearing of the cache by pressing on the clear cache button present in the menubar.

Also, the cached files are kept in the directory `/tmp`. This may cause a problem if several persons are viewing the theorems with the same name but different contents on the same machine. In future versions, we plan to give the possibility to users to change the cache directory, for instance by giving its name in a resource.

5.2 Recording commands

One of the tasks of Coq users is to keep a record of all the commands that have been sent to the prover in the right order. This record, which will shall also call a *script*, makes it possible to replay the proof. There are many good reasons to replay proofs: very often the proof work will be done in two separate phases: a first exploratory phase, where one will try several different solutions, sometimes with useless or complicated steps and a second phase where one will work on cleaning up the script obtained in the first phase, to stress the important parts, factorize repetitive steps in lemmas, and automatize trivial steps that were done by hand in the first phase. This cleaning up phase is very important as it can transform fragile proofs into robust ones, by making minor modifications less likely to break the general structure of the proof.

Providing good help to this work of script management is one of the design principles of CtCoq. The user-interface provides a uniform set of functionalities to execute commands, record them, and go back in history.

5.2.1 Script management basic principles

Script management works by placing the commands sent to the proof system in a special zone, the script area (this area is characterized by a brown background). This zone is "read-only", so that you can copy from it but you can't modify the commands it contains. This restriction ensures that the commands recorded there will not be corrupted, even in erroneous manipulations. Since the interface process works in parallel with the Coq proof assistant, it is possible to continue editing and sending command while the proof assistant is busy working on a previous command. To provide a healthy behavior in this case, the interface uses two other "read-only" zones, the tentative area (characterized by a pink background) and the buffer area (characterized by a yellow background). The tentative area contains the command that is currently being processed by the proof assistant (when such a command exists). The buffer area contains the commands that the user requested to be sent to the proof assistant, but which cannot yet be processed because the proof assistant is still busy with another command.

During a working session these areas move inside the window in various ways. It may also happen that some commands are copied to be added to a given area.

This behavior is guided by six kinds of events. Four events come from the user: they correspond to the user requesting a "Do it" action, an "Undo/Discard" action, an "Abort" or a "Reset". Two events come from the proof assistant: they correspond to a command being successfully executed, "Done", or an error message being emitted during the execution of a command, "Error".

5.2.2 Basic Behavior

Basically, the "Do it" event is triggered when the user selects a command and clicks on the button Do it or types `^C g`. This provokes the chosen command to be inserted in the tentative area while it is processed by the proof assistant. The answer of the proof assistant may be an acknowledgement: "Done". In this case the command is moved from the tentative area to the script area (or, more often, the script area is extended over the command). Alternatively, the answer of the proof assistant may indicate a failure to execute the command: "Error". In this case the command is simply restored to the area it belonged to before.

When adding a new command to the script area two cases may occur. The first case is when the new command is the first command immediately after the end of the script area. In this case the script area is simply extended over this command. The second case is when the new command is elsewhere in the window. In this case, the command is copied to the end of the script area. This ensures that the script will be coherent.

5.2.3 Buffering Commands

Complications can occur because of possible race conditions between the proof assistant and the user. While the proof assistant is busy working on a command, the user can edit and send extra commands. These extra commands will have to be kept in the user interface and sent to the proof assistant in the right order as soon as the proof assistant is available. The buffer area serves this purpose.

5.2.4 Undoing

Another source of complication is the "Undo" facility, that can be obtained by clicking on the button or by typing `^C _`. This functionality is provided only for those commands that are related to goal directed proofs and commands that define Coq

objects.[6] It works differently depending on the area in which the last command sent resides:

- If the last command is already in the script area, then it has already been executed by the proof assistant. An "Undo" command has to be sent to the proof assistant. When receiving the acknowledgement of this "Undo" command, the script area will be updated by removing the last command from the area. Actually, the command is not moved at all, it is only the script area that shrinks.
- If the last command is in the tentative area, the proof assistant is currently working on this command. The user interface must wait for this command to be completely processed. If the answer is an "Error" event, then nothing has to be done since the command is actually not executed. If the answer is a "Done" event, one should proceed as if the command was already in the script area.
- If the last command sent is in the buffer area, it can simply be removed from that area. No communication with the proof assistant is required as the command has yet to be sent.

5.2.5 Aborting

The action of the button `Abort` interrupts the current goal directed proof. This is tantamount to cancelling all commands back to the previous `Theorem` command. The script maintenance tool takes this into account and shrinks the script area to the command immediately preceding that `Theorem` command.

5.2.6 Resetting

There are two possibilities to reset to a previous state. One is simply to select the command defining a new object in the script area with the left button of the mouse and to type `^C r` for `Reset`. This generates a command that provokes a reset to this defining command. As a side effect, all the commands following this one are removed from the script area.

[6] In fact, the commands that can be undone are limited to those for which the Coq system provides a clean mechanism to actually "undo" the command (or for commands that don't change the state of Coq, such as **Check**, **Search**). For the most part, this means the Coq commands: **Abort**, **Reset**, **Undo**. Although we can "undo" an **AddPath** command by generating the appropriate **DelPath** command, there is no support provided, for example, for undoing a **Require** or a **Hint** command. We are still actively working in this area.

The second possibility is to edit a `Reset` command and send it using the Do it button or by typing `^C g`. The user-interface will attempt to update its state in a conservative way: if the corresponding definition is stored in on of the *Command* windows then the script is correctly updated. But otherwise, the script area is completely cleared away. This is a conservative behavior: there is no way to know which of the commands in this script area are still valid and the area is completely cleared.

Other commands have an effect on the state of the Coq system. These commands can be used in the CtCoq interface, but they are not very well integrated in the script management facility: the script simply records these commands without performing the extra cleaning that could be done. These commands are `Restore State`, `Reset After`, `Reset Section`.

All the commands that alter the output of the theorem prover must *absolutely* be avoided. In the current version, the commands `Focus, Unfocus, Begin Silent, End Silent` fall in this category.

5.2.7 Executing a command with textual fragments

If a commands contains textual fragments when the user requests for this command to be executed, the user-interface attempts to parse all these textual fragments before sending the command to Coq. If one of these fragment cannot be parsed, the execution fails.

5.2.8 Executing several commands in one interaction

At any time, it is possible to request for several commands to be executed at the same time, by selecting these commands and typing `^C g` or clicking on the Do it button. All the commands will be executed in the order they appear on the screen, even if the selected commands are already in the brown area that denotes commands that have already been executed. If the executed commands are not the first commands following the brown area, they will be copied to the end of this area as soon as they are successfully executed.

5.2.9 Automatic do it

There is also an *automatic-do-it* functionality to avoid having to click on the Do it button every time. This works as follows: after some editing operations, the system sends the command containing the current selection to the Coq proof assistant auto-

matically. In practice, this *automatic-do-it* functionality does not work all the time. There are three reasons for this functionality to be inactive:

- The command containing the current selection is incomplete because it still contains empty place-holders.
- The last editing operation does not normally trigger the functionality. The commands that do not trigger *automatic-do-it* are the following:
  1. Copy, cut, paste, insert, from the Edit menu (or key-bindings `<Escape> w`, `^W`, `^Y .`, `^Y <`, or `^Y >`).
  2. Text editing.
- The *automatic-do-it* functionality is disabled.

The first two reasons prevent the *automatic-do-it* functionality from getting in the way of users by sending commands that are not yet ready for execution. First of all, sending incomplete commands will invariably provoke disruptive error messages from the Coq proof assistant. However, the choice to have some editing commands that do not trigger the *automatic-do-it* functionality may render the CtCoq system somehow inpredictible and this may change in future versions of the system.

The third reason is naturally implemented to provide the user with ways of controlling this *automatic-do-it* functionality. The user can simply decide to enable and disable it at will. There are two ways to control the state of the CtCoq system with respect to this functionality:

- In the multiple window *Command + State + Search Theorems*, to the left of the Do it button the system provides a check box on which users can click with the mouse buttons to set the functionality on and off,
- While the mouse is in the *Command* window, users can switch the state of the functionality by typing `^C t` (for *toggle*).

Whether a user decides to switch the state of the *automatic-do-it* functionality using the mouse or using the keyboard, this state is permanently visible in the check box that appears to the left of the Do it button provided below the *Command* window.

5.2.10 Executing a whole window

Another shortcut provided for in the *Command* window allows one to execute the window's contents (useful for replaying scripts), or to execute everything which follows the previously executed commands. This funcitionality is triggered by typing

`^C a` (for All) in the *Command* window. If one command in the window provokes an error, the system stops there. After correcting the error, it is possible to type `^C a` again to resume the automatic execution of all the remaining commands. This command is very handy when porting proofs from one axiomatization to another or from version of Coq to another as it makes it possible to jump quickly from one problematic command to the other, without having to take care of the script management.

It may also happen that one wants to execute only part of the window, resuming step-by-step execution when reaching a delicate portion of the proof. This can be achieved by inserting a place-holder at the position where one wants the execution to stop. The interface will record all the commands between the end of the dark brown area and the place-holder holder as commands to be executed and start the process as before.

5.2.11 Reducing the size of the script

The natural tendency is to develop a whole proof in one document. This is unsatisfactory for two reasons. First, it would be better to break down the proof in several files to ease their management. Second, when the content of the *Command* window grows too big, the CtCoq user interface slows down. To avoid this problem, the user interface provides a facility to extract a subpart of the recorded script and store it into an auxiliary file. The user only needs to select the relevant subpart using the left button of the mouse and type `^C B` (for *build load file*, please note that we use a capital `B`). This prompts the user for a file name, checks that this file name is valid, inserts a command "`Load <`*file*`>.`" in the script area, and stores the data in that file. This makes sure that the script area still contains a reference to removed definitions and commands.

In its current state although the CtCoq user interface can re-execute files containing `Load` commands, it offers only a very primitive interface. For example, the build load file is saved where the user indicates, however, we do *not* automatically add this directory to the `LoadPath` of Coq. Thus, to execute a file containing a `Load`, it is the user's responsibility to execute an `AddPath` command. Also if an error occurs while loading a file, one receives the error, however, no attempt is made to open the file, as we cannot localise the error[7].

[7] This "build load file" facility is still very new and it will be better integrated in future versions. In spite of the lack of integration, we included it as it can increase drastically the productivity of CtCoq users.

6 Command Generation

Commands for goal directed proofs can be constructed using the regular editing tools. However, there are a lot of cases where commands can be generated by selecting expressions in the *State* or *Library Fragment* windows and triggering actions by various keystrokes. We describe all these shortcuts in this section.

6.1 Proof by Pointing

Proof-by-pointing enables users to forget about the script and the command language and to construct the proof by working only on the logical formulas in the goals and theorem statements.

The basic principle of proof-by-pointing is to enable the user to guide the proof assistant by simply bringing selected expressions to the foreground. This tool deals only with the basic logical connectives: $\forall$, $\exists$, $\wedge$, $\vee$, $\Rightarrow$, and $\neg$ and with equality. When the user selects a subexpression of a goal, a command is generated that has the following characteristics:

- the command will work on that specific goal,
- if the user selected a sub-expression of a hypothesis then:
  - if the hypothesis is a universally quantified formula, the command will use an instance of this formula (the Coq tactic `Generalize` with an application of the hypothesis to a given argument then `Intros`),
  - if the hypothesis is an existentially quantified formula, the command generates a witness for this formula (the Coq tactic `Elim` then `Intros`),
  - if the hypothesis is a conjunct, the command breaks this conjunct into pieces and continues recursively on the hypothesis corresponding to the selected expression (the Coq tactic `Elim` then `Intros`),
  - if the hypothesis is a disjunct, the command generates the two subgoals corresponding to both cases of the disjunct and continues recursively on the subgoal corresponding to the selected expression (the Coq tactic `LApply`),
  - if the hypothesis is an implication, the command generates a subgoal for the left hand side and attempts to prove the same subgoal using the right hand side of the implication as an extra hypothesis; the command continues recursively on the sub-goal corresponding to the chosen expression (the Coq tactic `Elim` then `Intro`),

  - if the hypothesis is a negation, the command attempts to prove the negated formula (the Coq tactic `Elim`),

- if the user selected a sub-expression of the conclusion then:
  - if the conclusion is a universally quantified formula, the command attempts to prove that formula for an arbitrary constant (the Coq tactic `Intro`),
  - if the conclusion is an existentially quantified formula, the command attempts to prove the formula for a given witness (the Coq tactic `Exists`),
  - if the conclusion is a conjunct, the command generates the two subgoals and continues on the subgoal corresponding to the selected expression (the Coq tactic `Split`),
  - if the conclusion is a disjunct, the command attempts to prove the chosen case (the Coq tactic `Left` or `Right`),
  - if the conclusion is an implication, the command generates a subgoal where a new hypothesis corresponds to the left hand side of the implication and the conclusion is the right hand side of the implication; the command continues recursively on the new hypothesis or on the conclusion depending on the selected expression (the Coq tactic `Intro`),
  - if the conclusion is a negation, the command attempts to prove a contradiction, using the negated formula as a hypothesis (the Coq tactic `Unfold not` then `Intro`),

- if the user directly selected the complete conclusion or a complete assumption, then the command tries to prove the goal by assumption (the Coq tactic `Assumption` or `Exact`).

When the user selects a sub-expression of a universally quantified formula in a hypothesis or a sub-expression of an existentially quantified formula in the conclusion, a value must be given to the proof assistant. For this purpose, the proof-by-pointing tool generates incomplete commands that contain place-holders that have to be filled in. The name of the place-holder is usually related to the name of the bound variable in the quantified formula. If the user-interface refuses to perform a "Do it" action for a command generated by proof-by-pointing, it usually means that a place-holder has been left unfilled in that command. Filling in a place-holder is easily done using the copy-paste shortcut provided by the middle button of the mouse.

The result of `Check` or `Search` commands, when displayed in *Library Fragment* windows, can also be used for proof-by-pointing. These expressions can simply be viewed as extensions of the hypotheses of goals.

6.2 Point and Shoot

The basic behavior of proof-by-pointing is to bring the selected expression to the forefront and simply try to solve the produced goal by assumption. A natural extension to this behavior is to replace this last try by the application of a specific command. For instance, the user can use proof-by-pointing to create a new assumption and simply apply this assumption using the `Apply` command. The CtCoq interface provides this kind of behavior for several commands. To use it, it suffices to choose the relevant expression with the left mouse button. Then the *shooting* function is chosen either in the popup menu that appears when one depresses at the same time the `Control` key and the right mouse button, or by a keyboard accelerator.

As a mnemonic, consider that the user aims at the target with his left finger and then chooses the device. This is the explanation for the name *point-and-shoot*. This feature simply enables the user to combine proof-by-pointing and apply specific commands as a shortcut. The various possibilities for point-and-shoot and their keyboard accelerators are:

a. *point and* `Apply`. When the selected expression can be transformed into an assumption, the generated command is a combination of proof-by-pointing and `Apply` on that assumption. The `Apply` pattern that is generated takes into account the specific form of that specific assumption. For example, it generates place holders to provide values for the universally quantified formulas that will not be instanciated by the `Apply` command. When the selected expression cannot be transformed into an assumption, it simply tries to solve the corresponding goal by assumption.

e. *point and* `Elim`. When the selected expression can be transformed into an assumption, the generated command combines proof-by-pointing and `Elim` on that assumption. Here, the `Elim` pattern that is generated is dumb, and executing the command may generate uninstanciated meta-variables.

g. *point and* `Generalize`. When the selected expression can be transformed into an assumption, the generated command combines proof-by-pointing and `Generalize` on that assumption.

`i`. *point and* `Injection`. The generated command combines proof-by-pointing and `Injection`. Note that `Injection` is itself a command that uses a notion of paths and the system tries to generate a clever path for this command, depending on how deep in the expression the user selected.

`l`. *point and* `Simpl`. The generated command combines proof-by-pointing and `Simpl` on the conclusion or an assumption.

`m`. *point and* `Assumption`. This is the basic proof-by-pointing behavior.

`r`. *point and* `Red`. The generated command combines proof-by-pointing and `Red` on the selected expression.

`w`. *point and* `Rewrite`. When the selected expression can be transformed into an assumption and it is an equality, then a `Rewrite` command is generated that uses this equality. The direction of the rewriting is decided by the member of the equality that has been chosen. If you choose the left hand side then the `Rewrite ->` command is generated, if you choose the right hand side, then the `Rewrite <-` command is generated.

`f`. *point and* `Auto`. The generated command combines proof-by-pointing and `Auto`.

6.3 Toplevel Generated Commands

Other commands can be generated by a combination of the mouse and a keystroke although the idea of bringing a sub-expression to the front does not make much sense for these commands. Usually, the subject of these commands has to be an entire hypothesis or the entire conclusion of a goal. For example, it is possible to require that a hypothesis be removed from the context or that a definition be unfolded. Patterns for these commands can be generated as in point-and-shoot by a combination the left button of the mouse and a keystroke. Here are the various possibilities:

`^C u`. `Unfold`. When the selected expression is an identifier, the interface generates the necessary command for the definition of this identifier to be unfolded at that specific occurrence.

`^C w`. `Clear`. When the selected expression is in a hypothesis, the interface generates the necessary command to remove this hypothesis.

`^C ^ Pattern.` when the selected expression is in a goal, this command introduces a pattern command referring to this expression.

`^C f. Fast (Add Auto).` This is a shortcut to add the Auto tactic to the end of tactic currently under construction. NB: This keyboard accelerator, unlike the previous ones, which are valid uniquely in the *State* window, can be used from the *Command*, *State*, and *Theorems* windows.

6.4 Drag-and-drop Rewrite

To perform rewriting in CtCoq, users must normally construct a Rewrite command where they must provide a theorem, the parameters for its instantiation, and the position where the rewriting occurs. In the tool we present here, these three kinds of parameters are determined simply when the user "drags" an expression with the mouse, by clicking on it, moving the mouse, and releasing the button (with the <Ctrl> key depressed and the left button of the mouse). The drag movement can convey some intuitive information, when the replacement has the effect of moving one piece of data from one place to the other.

In general, the drag movement is used to express one of several kinds of operations:

1. The combination of two expressions on the screen, like the two x's in the expression "x - x" (and replace it by "0", this will be expressed by taking one of the x's and dragging to the other one) or the + and the * in the expression "x * (y + z)" (and replace it by "x * y + x * z", this will be expressed by taking the + and dragging it to the *).

2. The movement of some expression with respect to other expressions, like when one wants to replace "x + y" by "y + x" or "x + (y + z)" by "y + (x + z"). In both cases this will be expressed by taking the x and moving it to the y.

3. The cancellative effects of several operations together, "-(-x)" which can be replaced by "x". This will be expressed by taking one of the "-" signs and moving it to the other.

4. The permutation of operators like in "-(1/x)" which can be replaced by "1/(-x)". This will be expressed by taking the "-" sign and moving it on the "1/" sign.

5. The rearrangement of complex structures like associative patterns of the form "(x + y) + z", which can be replaced by "x + (y + z)". This will be expressed by taking the parentheses and and moving them to the second "plus" sign, to express that we want the parentheses around the second sum.

6.5 The default behavior

We give here the list of all the theorems that can be triggered by the default behaviour of our drag-and-drop mechanism. This list is very short, as it is restricted by the theorems that are present in the standard `Arith` package of Coq. It is included to make it easy to try the functionality. Once convinced, users should adapt the mechanism to their own domain of application, using the tools described below.

6.6 Extending the drag-and-drop mechanism

We have devised a language of drag-and-drop behavior rules, where each possible reaction performed by the drag-and-drop tool is described by a separate behavior rule. To ensure the validity of each operation, each rule rests on a rewriting theorem whose proof should be done separately. Behavior rules can be edited manually, using an interactive editor. We also provide automatic tools that directly infer simple behavior patterns from the form of rewriting theorem statements, when these theorems belong to well known classes of theorems.

6.6.1 An editor for drag-and-drop rules

To edit manually the drag-and-drop rule database used by the tool, type `<Escape> x` when the mouse is in a CtCoq multiple window and type `edit-dad-patterns` at the prompt. This will pop a "Drag and Drop rule editor" window with three sub-windows.

1. a pattern editor (placed on the top and in the left), to edit the pattern of each rule and set the initial and final paths in this pattern,

2. a command pattern editor (bottom and left), and

3. a rule list (right).

he drag-and-drop behavior is given by an ordered collection of drag-and-drop rules, where each rule has a name, a filtering pattern and a command. The names of all rules normally appear in window number 3, while the pattern of the current

rule appears in the window number 1 and the generated command appears in the window number 2.

6.6.2 Selecting a rule collection

When the rule editor is just started, all its windows are empty. The options "Load Rules" and "Take current rules" from the the "File" menu on top of the pattern editor make it possible to start working on a collection stored on disk or the list of rules currently used in the session you are running. It is also possible to start a new collection from scratch by choosing the option "Add rule" from the "Rules" menu.

6.6.3 Adding a new rule

To add a new rule in the rule collection, you can use the option Add Rule from the Rules menu. You can also edit existing rules by simply clicking on a name in the list that appears on the right.

6.6.4 Editing rule patterns

To edit a rule pattern, you can use all editing facilities provided by the menubar on top of the pattern window. You can also copy and paste data from the other windows of CtCoq. However, beware that names of Coq variables matter, so that a pattern with the form "x + y" will not work for an expression of the form "a + b". To make a pattern applicable in a variety of cases, independently of variable names, you need to generalize you pattern by introducing *metavariables*. in CtCoq, the function of such metavariables is fullfilled by place-holders

to replace an expression by a metavariable, you can use the option Name Metavariable from the `Edit` menu . Here again, names of metavariables matter for two purposes:

1. Like in an ML pattern matching rule, a metavariable occurring in the pattern can be used to convey information towards the rule command. Thus, if a metavariable with the same name occurs in the command, this metavariable will be replaced when the rule is triggered by the value matched by the metavariable with the same name in the pattern.

2. Like in a Prolog unification, two variables with the same name in the pattern will indicate that the same syntactic value has to occur in both places when matching against data.

The pattern is not complete if you do not indicate the movement of the mouse that will be recognized by the rule. To do so, you simply need to drag the mouse from the starting position to the final position, having depressed the control key of you keyboard. As a feedback, you will see the starting position take a pink background and the ending position take a green background.

Depending on the pretty-printer used in the window, you may not be able to select the expressions you want, because some position are not denoted by characters on the screen. To turn around this restriction, you may have to change the pretty printer to the default pretty printer, so that every sub-expression of the pattern becomes accessible to the mouse.

6.6.5 Editing the command

The only commands allowed in the current version are rewrite commands. Here again you can use regular editing commands to change the theorem and arguments involved in this rewriting. Note that the rewrite direction can be changed easily using the option Invert Direction from the `Edit` menu.

6.6.6 Changing the order of rules

The drag-and-drop rules are tried in the order they occur in the list displayed in the rightmost window. To achieve a reasonnable behavior, it often necessary to change this order, to put the rules with the most general patterns behind the more specific rules. The interface provides two possibilities for this.

1. The rules can be automatically sorted, using the option Sort rule list from the `Rules` menu.

2. It is also possible to "drag" and "drop" a rule from one position to another by clicking on the rule name with the `Control` key depressed and dragging the mouse. The selected rule name then follows the mouse.

6.6.7 Using new rules

Once the right rules have been edited, they can be used in the current session by choosing the option Use in this session from the `File` menu. Rules can also be stored on disk for later use by choosing the option Save rules from the same menu. this will prompt the user for directory name where several files will be generated to store the rules.

To make sure the saved rules will be used in future sessions, you need to add two resources in your resource file:

```
ctcoq.DragAndDrop.Module.Location: <your directory>
ctcoq.DragAndDrop.Module.Root: user
```

To help you perform this task, the CtCoq user-interface informs you with a message containing the exact text of the resources to set. You can copy and paste these resources to you favorite text editor.

6.6.8 Computing drag and drop rules automatically

There are classes of theorems for which drag-and-drop rules can be automatically computed. In the current implementation, the classes that are recognized are as follows:

Cancellation theorems Cancellation theorems have a conclusion of the form "x - x = m", where the operator "-" and the result "m" can vary. The drag-and-drop behavior associated to this kind of theorem is triggered when one selects one of the x's and drags to the other. The intuitive meaning is to combine the two instances of x.

"zero" theorems Zero theorems have a conclusion of the form "x + 0 = m", "0 + x = m", where the operator "+" and the result "m" can vary (for instance the operator can be the multiplcation operator and m can be "0"). The drag-and-drop behavior associated to this kind of theorem is triggered when one selects the "0" and drags to the operator "+". The intuitive meaning is to combine the "0" with the operation for which it has a special meaning.

Factorisation theorems Factorization theorems have a conclusion of the form (x * y) + (z * y) = (x $ z) @ y, where the operators "$" and "@" do not necessarily have to be the same as "*" and "+". Two drag-and-drop behaviors are associated to these kinds of theorems, corresponding to rewritings in both directions.

1. If one finds an instance of "(x * y) + (z * y)" and the user selects one of the y's and drags to the "+" operator, this instance is replaced by the corresponding instance of "(x $ z) @ y". The intuitive meaning is to drag the y's up the expression structure.

2. If one finds an instance of "(x $ z) @ y" and the user selects the y and drags it to the "x" or the "z", this instance is replaced by the corresponding instance of "(x * y) + (z * y) ". The intuitive meaning is to distribute the y to the arguments of the "$" operator.

Commutativity theorems Commutativity theorems have a conclusion of the form "x + y = y $ x" where the operator "$" does not necessarily have to be the same as "+", the drag-and-drop behavior associated to this kind of theorem is triggered when one selects x or y in an instance of the left-hand side or the right-hand side of the theorem and drags to the other argument.

7 Interacting with the file system and other tools

7.1 Output formats

The CtCoq user-interface can produce data in several formats, some of which both Coq and CtCoq can re-read, some of which only CtCoq can re-read, and some of which cannot be re-read. Normally, the buttons Save and Save as ... (and the key bindings `^X ^S` and `^X ^W` produce a textual format that can be re-read by Coq and CtCoq when the file name ends with ".v" and a format that can only be re-read by CtCoq when the file name ends with ".po".

The button Write PS produces Postscript output that cannot be re-read, but can be printed or displayed with a postscript previewer. Other tools, like the *drag-and-drop* rule editor, use files of their own format, that can be re-read only by CtCoq.

7.2 Saving automatically in polish format

The polish format is a format specific to Centaur. To recover data from a file in a polish format, you can simply use the Open ... option from the `File` menu. You can then save this data in more conventionnal form. The polish format has three main advantages:

- It is quick to produce and re-read,
- Reading does not use the parser, so that incomplete commands can be stored and re-read easily,
- the output routine is very robust and will work more often even if the the CtCoq system goes wrong.

It is possible to request that CtCoq save systematically its data in polish format in addition to the textual format. In this setup, it associates to each file in textual format a file in polish format. When reading a file, it checks modification times for these files and systematically reads the file in polish format in preference to the file in text format.

To enter this mode, it is only necessary to put the following resource in your resource file (see section 9.1).

```
Centaur.vernac.AlternateSaveTrees:t
```

To perform the association between files, CtCoq using a naming convention, simply adding the prefix "." and the suffix ".po" to the file name.

7.3 handling syntactic extensions of Coq

The CtCoq system uses two different processes for the Coq prover and the parser. In the system design, it has been very difficult to ensure that both processes are synchronized with respect to syntactic extensions. The solution offered so far is that the parser executes none of the commands it parses, except for grammar rule declarations and "`Require`" commands. Thus, it is possible to make sure the parser takes into account a set of new grammar rules, by making it parse a `Require` command referring to a module that imports these grammar rules. The `Require` command may fail if the parser does not have the right search path for modules. In this case, it is possible to adjust the path using the option Add Syntax Path from the `File` menu.

7.3.1 Example

If you placed your grammar rules in a file `$HOME/my_syntax/rules.v`, and you want these rules to be used in your session, you should proceed through the following steps:

1. At the shell prompt, do

   ```
   cd $HOME/my_syntax; coqc rules
   ```

2. In CtCoq, choose the option Add Syntax Path from the `File` menu and type `$HOME/my_syntax` at the prompt.

3. Go in one of the CtCoq windows, type "`Require rules.`", and parse this text.

As can be seen in this example, it is only when parsing the text "`Require rules.`" that the parser actually loads the rules. This may cause a problem when loading data to CtCoq in polish format, because this format does not require any parsing to occur. In this case, the Coq system may change its syntax state because of `Require` commands that the parser does not see.

7.3.2 Configuring a permanent setup

To take good advantage of the speed provided by the polish format and the comfort provided by extensible syntax at the same time, it is useful to configure the parser at the beginning of a session. The CtCoq system uses a configuration file to make this initialization possible. This configuration file is found by default in `$HOME/.vernacrc` but it can be changed using the shell variable `USERVERNACRC`.

The commands that can be put in the configuration file are very restricted, they are as follows:

`add_path` *num string*. The number may be any numeric value. The string must be encapsulated between double quotes, it can contain references to shell variables, by preceding identifiers with a dollar sign (`$`). This string should denote a directory.

`load_syntax_file` *num Identifier*. The number may be any numeric value. The identifier must be a Coq module name. This command is equivalent to `Require` *Identifier*.

`quiet_parse_string` *<line break>some text<line break>*`&& END--OF--DATA`. This command forces the parser to parse *some text*. It is useful to force initialization of all the parsing tables after the load of new grammar modules.

It may happen happen that a configuration file contains irrelevant data that provokes the parser to die. To have a better understanding of failures it may then be necessary to see the exact output of the process. This is made possible by typing `(vernac-log t)` in the CtCoq xterm. After verifying that all goes well, it is possible to turn off this trace mode by typing `(vernac-log ())`.

At anytime, it may also be necessary to restart a fresh parser. This is made possible by typing the following line in the CtCoq xterm:

```
({centaur}:remove-parser 'vernac ())
```

When receiving this command CtCoq kills the parser process and warns of its death by issuing a message of the following form. The parser will be automatically restarted afresh at the next parsing request.

7.3.3 Example continued

If you want the grammar rules in `$HOME/mysyntax/rules.vo` to be loaded at the start of all you sessions, put the following text in the file `$HOME/.vernacrc`:

```
add_path 1 "$HOME/mysyntax"
load_syntax_file 2 rules
quiet_parse_string
Goal a
&& END--OF--DATA
```

7.4 Adapting printing rules to syntactic extensions

If your development uses grammar extensions and introduces new operators, it will happen that the text produced by CtCoq will not be readable by Coq. In practice, this will be characterized by *syntax error* messages being emitted by Coq when you try to execute a command. This section explains how to cope with this phenomenon.

7.4.1 Grammar rules with new operators

In Coq, grammar rules can have one of the two following forms:

1. the defined rule can map a new text pattern with an old existing construct, using the "«" and "»" brackets to indicate the old existing construct. Coq files using such grammar extensions will parse with no problem in CtCoq, but the new notations will always be expanded and replaced by the old constructs. This can be coped with by adding pretty printing notations, as explained in section 4.3.

2. the defined rule can map a new text pattern to a new parse tree operator. In this case, CtCoq will use a systematic way to print this new operator and this way of printing may not coincide with the text pattern that is recognized by the grammar rule. It is then necessary to adapt CtCoq to make sure the way of parsing and the way of printing are compatible.

In the rest of this section, we only take care of the second case, and we indicate how to fix incompatibilities between the parser and the pretty printer on an example.

Coq provides two main syntactic categories where users can extend its syntax: `vernac` and `tactic`. The parser of CtCoq recognizes such extensions and maps them respectively to the operators `user_vernac` and `user_tac`. Both these operators are

binary, they receive an identifier as first argument and a list of parameters as second argument, and the operator of the second argument is respectively `varg_list` and `targ_list`. The identifier is always the operator given by grammar rule, while the parameters are the arguments given to this operator in the grammar rule.

7.4.2 Example

For instance, suppose the new grammar rule has the following shape:

```
[ "Some text" some_parser_1($a_1) ... some_parser_n($a_n) ] ->
  [ (A_new_operator $a_1 ... $a_n) ].
```

Corresponding trees have as head operator `user_vernac`, as first child the identifier `A_new_operator` and as second child a list containing the trees corresponding to `$a_1`, ..., `$a_n`, with `varg_list` as head operator. By default, CtCoq prints the following text when displaying this tree or sending it to Coq:

```
A_new_operator <$a_1> ... <$a_n>.
```

7.4.3 Adding pretty printing rules

To restore the compatibility between CtCoq's printing and Coq's parsing, there is a need for a rule that recognizes the tree pattern and prints the expected text. This can be done by adding new printing rules to the set of rules used when printing data in files or for Coq. For this the `usernotations` pretty printer used in section 4.3 is not adapted, because this pretty printer is used only for screen display. It is necessary to define yet another pretty printer and to set up the resources so that this pretty printer is used when communicating to Coq or when saving data in files. Here we show how to perform this when we choose to name the extension `userstd`, and to store it in the directory `$HOME/contrib/userstd`:

1. add the following lines in your resource file (see section 9.1):

```
Centaur.vernac.ppml.userstd.Root: user
Centaur.vernac.ppml.userstd.Location: contrib/userstd
Centaur.vernac.ppml.userstd.Mode: compiled
Centaur.vernac.ppml.std.package (userstd explicit formulas common)
Centaur.vernac.ppml.disp.package : \
                (shrink usernotations userstd \
                display formulas common)
```

2. Create a file named `$HOME/contrib/userstd/vernac-userstd.ppml` with the following contents:

```
pretty printer userstd of vernac is

rules
  <your rule here>

end pretty printer;
```

After triggering the option Reset Resources from the `File` menu in the *Centaur Messages* window, you can compile the pretty printing specification by reading it in a Centaur window (using the option Open in New ...) and choosing the option Compile of the `PPML` menu in this window.

7.4.4 Example continued

In our example, the pretty printing rule that should appear in the `PPML` specification is the following one:

```
user_vernac(ident 'A_new_operator',varg_list[**a]) ->
        [<hv 1, 0, 0> "Some" "Text" (**a) "."];
```

8 Online help

The multiple windows of CtCoq all have a `Help` menu that can be used to request help on the system. The options of this menu provide a fast access to data on the internet through the `Netscape` internet browser. To use these options you should first start a netscape application on your X server. If you do not, these buttons will not do anything.

If you have no access to the internet at the moment you request help, it is possible to use a local copy of the help pages, by installing this copy in a directory of your choice and by setting resources referring to this directory. These resources should have the following form:

```
Centaur.help.show-keys.location: mydoc/keymap-doc.html
Centaur.help.general-help.location: mydoc/general-help.html
Centaur.help*global: ()
Centaur.help*Root: user
```

9 Configuration

The CtCoq system can be configured in several ways to adapt different aspects of its operation to the user's need. One common way of customizing is to set resources, akin to X resources. In some cases however, the configuration can be done via other configuration files or by shell variables.

9.1 Resources

When starting, the CtCoq system reads in a resource file found by default in `.centaur.rdb`. This resource file can be changed using an option on the command line prefixed with `-resfile`. For instance, typing

```
ctcoq -resfile ctcoqdb.rdb
```

ensures that the file `ctcoqdb.rdb` in the current directory will be used instead of `$HOME/.centaur.rdb`. This file can contain resources indicating the pretty printers, the coq command that is actually used in the system, the location of auto-save files, the location of in-line help files.

Resources put in this file usually have this form:

```
Centaur.name1.name2.name3: value
```

However it is possible to replace the word `Centaur` with `ctcoq` or some other identifier. When an arbirtrary identifier is given, the resource will be ignored unless this name is given on the command line, with the `-resname` option, as in the following command:

```
ctcoq -resfile ctcoqdb.rdb -resname ctcoqtry
```

The resource file can be edited in the middle of a session and re-loaded, so that the effect of the changes will take place (if possible). After editing the resource file with a regular text editor, you can have the new resources taken into account by choosing the Reset Resources from the `File` menu in the *Centaur Message* window.

9.1.1 Choosing the coq command

By default, the CtCoq system starts the command `$CTCOQROOT/etc/coqtop` and initializes this command by making it read the contents of a file found in:

```
$CTCOQROOT/contrib/ctcoq/coq-side/coq/coqrc
```

The system supposes that the coqtop command is simply a symbolic link to a regular installation of `coqtop`. You can decide to use your own instance of `coqtop` and your own initialization file by setting the following resources:

```
Centaur.Start-data.Root: user
Centaur.Start-data.Location: bin/coqtop
Centaur.Start-data.Initfile: mycoqrc
Centaur.Start-data.Log : t
```

The first resource indicates that all file locations will be given relatively to your home directory (other choices are `app` and `abs`). The second indicates the location of the command to run. The third indicates the location of a command file to load when starting this command. This `coqrc` file should be very close to the one found in the CtCoq distribution. Otherwise, the CtCoq system may not be able to communicate with the `coqtop` process. The fourth line indicates that one wants a trace of the `coqtop` output process to appear in the CtCoq xterm.

9.1.2 location of auto save files

It is possible to indicate that auto saving files will be stored in one specific directory by giving the following resource:

```
Centaur.checkpoint.directory: /net/home/bertot/tmp/ctcoq-backups
```

9.1.3 on-line help

If you are using CtCoq on a site that is not connected to the internet, or behind a firewall, it may be inconvenient to fetch help pages on the web. In this case you may want to install these help files locally and tell the system to use the local version with the following resources:

```
Centaur.help.general-help.location: help-files/general-help.html
Centaur.help.general-help.global: ()
Centaur.help.general-help.root: user
```

9.2 Startup files

9.2.1 startup file for Centaur

The Centaur process in the CtCoq system can be tuned using lisp commands stored in the file `$HOME/.centaur` file. For instance, you can add a new key binding by typing the following text in this file:

```
(defun #:coq:send-reset-initial (ctedit event)
    (st-emit
        (car (stobject-nodes (send 'subject ctedit) 'coq))
        'command
        0
        ({tree}:restore-from-persistent
            (copy '((vernac . restore_state) (ident Initial))) ())))

(define-command #:coq:commands:ascii-event:table '(#^C #^R)
                '#:coq:send-reset-initial)
```

9.2.2 startup file for coqtop

The file `$HOME/.coqrc` is used in CtCoq exactly as it is in plain coq.

9.2.3 startup file for the parser

it is also possible to put in the file `$HOME/.vernacrc` data that will be used to initialize the parser (see section 7.3.2).

10 Safety features

10.1 The backup mechanism

For the *Command* windows, the CtCoq user-interface provides a backup that saves regularly (every minute) the contents of these windows on disk. This mechanism can be used to recover data that could be lost when a crash occurs. The backup files are in polish format, they have a ".ckp.po" suffix, and they are saved in four possible place:

- If a file name "file.v" is associated to the window and it is possible to write a file ".file.ckp.po" in the same directory, then this is where the backup is saved. For instance, if you are editing the contents of the file found in `/net/home/bertot/tmp/drinking_problem.v` the backup file will be called `/net/home/bertot/tmp/.drinking_problem.v`. At every backuping cycle, the backup files associated to documents that have not been modified since the last save are destroyed.

- If no file name is associate to the document or if it is not possible to save the backup file next to the original file following the naming convention given above, and if the user set the resource `ctcoq.checkpoint.directory` (resource class `Centaur.Checkpoint.Directory`) to <directory> then the backup file has the name `<directory>/tmp/ctcoq<nn>.ckp.po`, where <nn> is the decimal representation of a number chose to avoid conflicts with existing files.

- If no resource has been set, the backup file name is `<current-directory>/.ctcoq<nn>.ckp.po`. Note that the base name starts with a dot, so that this backup file name does not appear in a plain listing of the directory.

- If it is not possible to create a file in the current directory, then the backup file name is `/tmp/ctcoq<nn>.ckp.po`. Note that this situation should be avoided since the directory "/tmp" is cleaned at every reboot of the system (and a reboot usually follows a crash).

If you need to recover data from an auto-save file, you can simply read this data in a CtCoq window using the "Read" button and then save it in a more conventional file. The CtCoq interface does not recognize an auto-save file as a special file when it reads it.

The checkpoint files are automatically destroyed on two occasions:

- When you save the contents of a window, in which case it is no longer modified.

- When you read a new file into the window. The checkpoint file associated to the previous window contents is destroyed.

The last behavior is problematic, since it may provoke the loss of some modifications. We hope to fix this in upcoming versions. Note that the checkpoint files are not destroyed at the time of the new read or save but rather at the time of the next checkpoint. Currently auto-saving is done at one minute intervals. (To note as well, the standard `Editor` windows do not yet profit from this feature.)

10.2 Memory safety

When the system runs short of memory, two things may happen:

1. If the shortage is benign, the system simply warns the user that memory is getting low and advizes him or her to close useless files,

2. If the shortage is so serious that the sytem may break down at any moment, the system provokes an auto-saving of all the open files and shuts down.

It may seem brutal to shut down the session, but the alternative is to let the user continue without any safety, with the risk of loosing all the work. When there is really no more memory available, no message can be displayed in the Centaur message window (there are messages, but they are printed in the xterm window where the system was started, and this window is usually out of sight), even the backup mechanism may go bersek and destroy previous auto-save files. So shutting operations while the system is still sure to operate correctly seems the best trade-off.

References

[Coq96] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-F. Filliâtre, Eduardo Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, B. Werner, *The Coq Proof Assistant Reference Manual*, version 6.1, available from INRIA Rocquencourt, December 1996.

[Centaur94] I. Jacobs, *ed.*, *The Centaur 2.0 Manual*, available from INRIA Sophia-Antipolis, September 1994.

[PPML] I. Jacobs, *ed.*, *The Centaur 2.0 Manual*, available from INRIA Sophia-Antipolis, September 1994. *Internet address:* `ftp://babar.inria.fr/pub/croap/Docs/ppml.ps`.

Appendix

A Installing the Graphical Interface

Versions of the CtCoq user interface can be retrieved using `ftp` from the machine `babar.inria.fr` (internet address 138.96.24.21) using the login name `anonymous` (as usual, you should give your e-mail address as the password). You should also be in binary mode. This release works only with the version of `Coq` V6.1 (Dec 24 1996).

The following parameters are used throughout this document:

- `CtcoqDir`: this is the pathname of the directory where you want your users to find CtCoq. It could be different from the absolute pathname of this directory due to NFS mounts (like those created by the automount facility on SunOS) or program version hiding using symbolic links. Example: /usr/local/ctcoq
- `Arch`: the name of the architecture to run CtCoq on. The known architectures are currently[8]:

  ```
  sun4OS4 sun4OS5 decosf1
  ```

  corresponding respectively to Sparc machines running SunOS 4.x or Solaris, and DecAlpha machines running OSF1 3.x[9]. Note that you may use the same hierarchy for several architectures.
- `COQTOPDIR`: this is the pathname of the directory in which you made the `Coq` system. This directory can be found in the `coqtop` command (it's a shell script) as the value of the `COQTOP` variable[10].
- `COQBINDIR`: this is the pathname of the directory in which you put the `Coq` binaries. It can be found as well in the `coqtop` command as the value of the `BINDIR` variable.
- `FtpDir`: the directory where you put the CtCoq archives.

[8] We can also provide versions for the `linux` operating system, but due to the wide variety of versions the installation procedure is more complicated. Please contact us.

[9] We will be soon able to provide a core image of the system for DecAlpha machines running OSF1 V4.0

[10] If you installed the `Coq` system following the 6- item of its installation procedure, you should use COQLIB, the directory where the `Coq` libraries reside

To ease the installation procedure, we suggest you set these variables in your shell to the appropriate values. You will then only have to paste the requested commands from this file to your shell. You will **NOT** need to set these variables to use CtCoq.

You need approximately 15 Mbytes of disk space to perform this extraction.

This tape is split in four `Gnu` compressed tar archives:

```
ctcoq.tar.gz
ctcoq-$Arch.tar.gz
ctcoq-doc.tar.gz
ctcoq-html-doc.tar.gz
```

You can extract these tapes by doing the following:

```
mkdir $CtcoqDir
cd $CtcoqDir
gunzip < $FtpDir/ctcoq.tar.gz | tar xf -
gunzip < $FtpDir/ctcoq-$Arch.tar.gz | tar xf -
gunzip < $FtpDir/ctcoq-doc.tar.gz | tar xf -
```

You must then make CtCoq know where `Coq` is installed on your system with:

```
ln -s $COQTOPDIR         $CtcoqDir/coq
mkdir $CtcoqDir/etc/$Arch
ln -s $COQBINDIR/coqtop  $CtcoqDir/etc/$Arch/coqtop
```

You can verify that these links are valid with:

```
ls -FC $CtcoqDir/coq
```

that should give something like:

```
CHANGES*   Makefile*  bin/       contrib/   src/       tactics/   tools/
INSTALL*   README*    config     doc/       states/    theories/
```

and

```
ls -sL $CtcoqDir/etc/$Arch/coqtop
```

that should give something like:

```
6   /net/croap/src/ctcoqv6/etc/sun4OS5/coqtop
```

CtCoq will use the following under `$CtcoqDir/coq`:

```
tactics
tactics/contrib/linear
tactics/contrib/natural
tactics/contrib/omega
theories/INIT
src/syntax
```

Using Tcl/Tk

In addition, `ctcoq` can use an elaborate file browser for most of the commands that require to find a file or a directory name. To use this facility, you need to check that you have on your site an installation of `Tcl` and `Tk` with respective versions `7.4` and `4.0` or higher [11]. To make sure `ctcoq` will find the command, establish a link between the relevant wish command and `$CtcoqDir/etc/$Arch/wish4.x` by typing the following command:

```
ln -s <the wish command> $CtcoqDir/etc/$Arch/wish4.x
```

Resources must also be set to indicate that this approach is taken or not [12]. The default resources assume that you have a valid version of `Tcl/Tk` installed. If it is not the case set the following resource:

```
Centaur.Query-file-name-impl: gfxobj-query-filename
```

to use a builtin query dialog instead of a file browser.

[11] To know whether `Tcl/Tk` is installed in you environment, type `wish` and type the following commands at the prompt:
`echo $tcl_version`
`echo $tk_version`

[12] To set resources for your personal use, you should add the line in the file `$HOME/.centaur.rdb`. To set resources that will be valid for all users of `ctcoq`, you should add the line in the file `$CtcoqDir/customize/Centaur.rdb`, replacing "`Centaur.StserverManager`" with "`?.stserverManager`". You may have to create the `$CtcoqDir/customize` directory. Also, if you have a version of `Tcl/Tk` linked with the `Tix` library, you can also use a slightly better file browser by typing the following command:
`ln -s <the tixwish command> $CtcoqDir/etc/$Arch/tixwish`
and by setting the following resource:
`Centaur.StserverManager.ct-file-selector.Command: tixfileselector`

Using Perl

To manage Coq comments (see section 4.2.7), the user-interface also uses a script that needs `perl` to run. To make `ctcoq` will be able to run this command, establish a link between the relevant `perl` command and `$CtcoqDir/etc/perl` (or `$CtcoqDir/etc/$Arch/perl` if this command is seen differently on various architectures) by typing the following command:

```
ln -s <the perl command> $CtcoqDir/etc/$Arch/perl
```

If you don't have perl on your machine, the `ctcoq` system wil work the same, except that all comments in your coq files will be lost.

Using local online documentation

The pages displayed by Netscape when you use the options provided by the `Help` menu can be accessed through the following URL:

```
http://www.inria.fr/croap/ctcoq/help/general-help.html
```

If you are working behind a firewall it may be interesting to install these pages locally. You only need to extract the files from `ctcoq-html-doc.tar.gz`. A way to install the documentation for all local users is to execute the following commands:

```
mkdir $CtcoqDir/docs/contrib/vernac/html
cd $CtcoqDir/docs/contrib/vernac/html
gunzip < $FtpDir/ctcoq-html-doc.tar.gz | tar xf -
```

Then create the file `$CtcoqDir/customize/Centaur.rdb` and add the following lines in it:

```
?.help.show-keys.location: docs/contrib/vernac/html/keymap-doc.html
?.help.general-help.location: docs/contrib/vernac/html/general-help.html
?.help*global: ()
?.help*root: app
```

Recuperating the license file

The `ctcoq` user-interface will not run without a license file that can be requested by mail at the following address:

```
ctcoq-request@sophia.inria.fr
```

When requesting for this license file, you must give us a mail address and the architecture on which you intend to run the machine. Also, we will appreciate to have very succint information about your institution (its name and location, for instance).

An Internet `WWW` page providing information about `ctcoq` and a form to request the license are set up at the following address:

```
http://www.inria.fr/croap/ctcoq/ctcoq-eng.html
http://www.inria.fr/croap/ctcoq/getinfo.html
```

In return for this information, you will receive the license file in a mail, with instructions for using this file.

This terminates the installation.

To have access to the commands of the CtCoq system, you must add the directories `$CtcoqDir/bin` and `$CtcoqDir/bin/$Arch` in your path variable.
You can then run the whole system by calling: `ctcoq`.

Moving the installation around

If you want the whole system to be visible in another directory (like `/usr/local/bin`), you should *not* move or copy any file from the `$CtcoqDir` directory. Rather it is necessary to place symbolic links to all the commands present in `$CtcoqDir/bin` and `$CtcoqDir/bin/$Arch`.

If you want to move the whole installation in another directory, you can do it using a command that respects symbolic links, like `cpio` or `tar`. In any case, you should never move parts of the installation separately.

Running processes on different machines

The `ctcoq` system also allows the users to spawn its `Coq` process on a machine other than the one that will run the interface with the following resource:

```
Centaur.Start-data.Host: <internet name of a machine>
```

You can also change the directory where the command will be found by using the following resource:

```
Centaur.Start-data.Root: abs
Centaur.Start-data.Location: <some file name>
```

The `ctcoq` system also allows the users to spawn its `vernac` parser on a machine other than the one that will run the interface with the following resource:

```
Centaur.vernac.SyntaxManager.vernacSyntax.Host: <internet name of a machine>
```

Please make sure that `$CtcoqDir` is visible on the other machine.

Checking the fonts

One auxilliary command is also available:

```
caff
```

The `caff` command can help you in setting up CtCoq to use different fonts in case your X server doesn't have our default fonts. Call `caff -h` for help.

Documentation is available in the files

```
$CtcoqDir/docs/contrib/vernac/user-guide.ps.gz
$CtcoqDir/docs/contrib/vernac/user-guide.txt
```

Have fun!

B Quick reference to all key bindings

Bindings valid in all CtCoq windows (structure editing)

`^O`	Inserts a place-holder before the current selection.
`^B`	Moves the current selection to the left.
`^F`	Moves the current selection to the right.
`Esc ^U`	Moves the current selection towards the top.
`Esc ^D`	Moves the current selection on the first son.
`Esc f`	Moves the current selection to the first atomic tree on the left.
`Esc b`	Moves the current selection to the first atomic tree on the right.
`Esc p`	Moves the current selection to the first place-holder on the left.
`Esc n`	Moves the current selection to the first place-holder on the right.
`Esc m 1`	Turns on the mode for automatic jump to the next place-holder.
`Esc m 0`	Turns off the mode for automatic jump to the next place holder.
`return`	In general, inserts a place holder after the current selection.
`^S`	Searches forward for the current search string.
`^R`	Searches backward for the current search string.
`Esc s`	Sets the search string and searches forward.
`Esc r`	Sets the search string and searches backward.
`Esc t`	Starts textual editing.
`^X m`	Shows or hides the template menu for structure guided editing.
`^X w`	Expands the current selection, to cancel the effect of ellision.

`^W`	Cut.
`Esc w`	Copy.
`^Y .`	Replace.
`^Y <`	Inserts before the current selection.
`^Y >`	Inserts after the current selection.
`^L`	Scrolls to the current selection.
`^X 4 f`	Loads a new file in another window.
`^X ^F`	Loads a new file.
`^X ^W`	Saves in a new file.
`^X ^S`	Saves the file.
`^X +`	Increments the print level.
`^X -`	Decrements the print level
`^X # 1`	Sets the print level to 1
`^X # #`	Sets the print level to a very high value
`^Z >`	Scrolls one line up.
`^Z <`	Scrolls one line down.
`^X <`	Scrolls left.
`^X >`	Scrolls right.
`Esc g`	Goto line
`^X 2`	Opens a second view on the same data.
`^V` `space`	Scrolls one page up.
`Esc v`	Scrolls one page down.
`Esc <`	Scrolls to the beginning.
`Esc >`	Scrolls to the end.
`Esc 2` `Esc 3` `Esc 4` `Esc 5` `Esc 6` `Esc 7` `Esc 8` `Esc 9`	Repeats the next command n times.
`Esc x`	Executes a function provided by the user.

Bindings valid in all CtCoq windows (text editing)

`return`	New line.
`Esc`	Parses the text area.
`back-space` `delete`	Deletes backward.

^B	Move left.
^F	Moves right.
^P	Moves up one line.
^N	Moves down one line.
^K	Removes the rest of the line.
^E	Goes to end of line.
^A	Goes to beginning of line.
^D	Delete one character forward.
bell	Cancels text editing.
^C g	Do it.

Bindings valid in all Coq related windows

^C s	Searches for theorems proving the selected predicate (Coq's `Search`).
^C c	Type-checks the selected expression (Coq's `Check`).
^C e	Evaluates the selected expression (Coq's `Eval`).
^C C	Computes the selected expression (Coq's `Compute`).
^C ^C e	Eval, using the current goal's context.
^C ^C c	Check, using the current goal's context.
^C ^C C	Compute, using the current goal's context.
^C p	Prints the natural language proof of the selected name.
^C t	Toggle the Auto do it check box.
^C g	Do it.
^C _	Discards the last command in Coq.
^C k	Aborts the current proof.

Bindings valid in all windows that contains premises

a	Point and Apply.
e	Point and Elim.
f	Point and Auto.
r	Point and Red.
w	Point and Rewrite.
i	Point and Injection.

`g`	Point and Generalize.
`l`	Point and Simpl.
`m`	Point and Assumption (Proof by pointing)
`!`	Duplicates the hypothesis.
`z`	Generates a Specialize command.

Bindings valid in Theorems and Library Fragment windows

`^C d`	Creates a Library Fragment window.

Bindings valid only in State windows

`^C u`	Unfolds.
`^C w`	Clears hypotheses.
`^C ^`	Creates a Coq Pattern command.
`^N`	Goes to next goal.
`^P`	Goes to previous goal.

Bindings valid in windows used for editing Coq commands

`^C a`	Executes all the commands in the window.
`^C d`	Opens a new multiple window.
`^C f`	Adds an Auto tactic to the current command.
`^C h`	Shows the goal on which the selected command was executed.
`^C q`	Prints the natural language explanation of the current proof.
`^C r`	Resets Coq to a state where the selected name is forgotten.
`^C v`	Insert a `Qed.` command.
`^C B`	Moves the current selection to an auxiliary window and file.

Index

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399