The CtCoq experience

Janet Bertot and Yves Bertot

INRIA, 2004 Route des Lucioles, 06902 Sophia Antipolis, France
{jmi,bertot}@sophia.inria.fr

Abstract. The CtCoq system is a graphical user-interface using a distributed architecture adapted to the Coq proof system [4]. Basic features are direct manipulation of formulas and commands using the mouse, mathematical notations with an extended character set and colors, menus for guiding users in their manipulations of commands and formulas. More advanced features also include proof-by-pointing, script management, and textual explanation of proofs.

1 Introduction

Integrating proof systems in user-friendly environments is a crucial issue if software engineers are to use these tools on a daily basis. We have concentrated on the user-interface part of proof engine development, by developing this user-interface as a different process, communicating with a proof engine using a simple protocol. The user-interface itself has been developed using a program environment generator, the Centaur system [2, 3]. The Coq proof engine is developed in an independent team. This paradigm of independent development of programs integrated in a multi-process, possibly distributed, system has an impact on the future directions of research in the domain of interactive proof systems, as it allows for the fast integration of progress in both the automated deduction and man-machine interface domains. This paradigm is also used in works such as Alf [5] and TkHOL [6].

The design principles are to make tedious tasks easier, regardless of their relevance to proof theory (see for example script management) and to provide customization whenever possible (see for example the work on pretty-printing and notations).

2 Basic features

The basic elements of the graphical interface are editing windows where the commands and formulas are displayed and edited in a structured way. This means that the data is almost always considered as trees, where the tree structure reflects the syntax of the language. From a purely visual point of view, the data still looks like enriched text. But by comparison with plain textual editing, the structured approach makes it possible to have more contextual help during the editing process and more accurate notations, adapted to usual mathematical practice, with a wide variety of fonts and colors.

Instead of a notion of current position in a document represented by a single character cursor, the editing windows make it possible to have a whole sub-structure as a *current selection.* Editing operations are then related to this current selection. For instance, the position of the current selection is used to provide menu-guided context help. For this tool,

in: *User Interfaces for Theorem Provers (UITP),*
York, United Kingdom, July, 1996.

a menu is provided to the user, whose content is updated every time the current selection moves, to propose syntactic constructs that are adapted to the syntactic context of this current selection. This menu can even be customized by the user who can add his own constructs and transformations.

The CtCoq user-interface also accommodates more traditional text editing. At any time, the user can select a structure and edit it as a text fragment, meanwhile forgetting the structure information. This makes it possible to go through states where the structure of the document is incoherent. Coming back to a structured representation is then necessary, and can be done through a parse operation. This integration of textual editing also makes it possible to communicate with other X-Window applications through the usual textual selection mechanism.

2.1 Window organization

When using an alpha numeric terminal, the communication between a user and an interactive system is very constrained and can be described by the pair of a simple input flow and a simple output flow. Using a graphical user interface, possibly with multiple windows, releases this constraint and there are many ways to handle input and output in this context, in fact, too many.

The first intuitive idea is to open a different window for every kind of data. There would be one window for the script of commands, one window for answers of one kind, another window for answers of some other kind, and so on. The pitfall is that users complain about the large number of windows. Important data is spread all over the screen and manipulating this data is slowed down by too many moves with the mouse and too many changes of visual context.

Based on our users' experience, we have designed an organization that makes it possible to concentrate important data in the middle of the screen, with geographical proximity between related concepts. This design is based on the ability to build a multi-paned window, which rigidly binds several windows, menubars, and buttons together.

This multi-paned window is shown in Figure 1. From top to bottom, it contains a *Command* window that records the script of commands sent to the proof engine, a *State* window that contains the current goals to prove, and a *Theorems* window that contains the result of queries into the proof engine's theorem database. The goals appearing in the State window are composed of two parts separated by a horizontal line, a *conclusion* that expresses the statement to prove and a *local context* that contains named statements that can be used for this goal only.

Obviously, the most important data in the window is the conclusion of the current goal. This data appears in the middle of the window. Equally obvious, the most important command in the *Commands* window is the last command in the script, either the one that just produced the current goal or the one that will be applied next (as in Figure 1). This command appears in the bottom of the *Commands* window, just above the goal. Between the *Commands* and *State* window, there is a "button bar" with buttons to perform some of the more frequent actions. So, the most important data is concentrated in the middle of the screen and frequent operations can also be triggered using the mouse in this area. Another characteristic of this organization is that the contents of the *Theorems* window are geographically close to the local context of the goal. This works well since the *Theorems* window

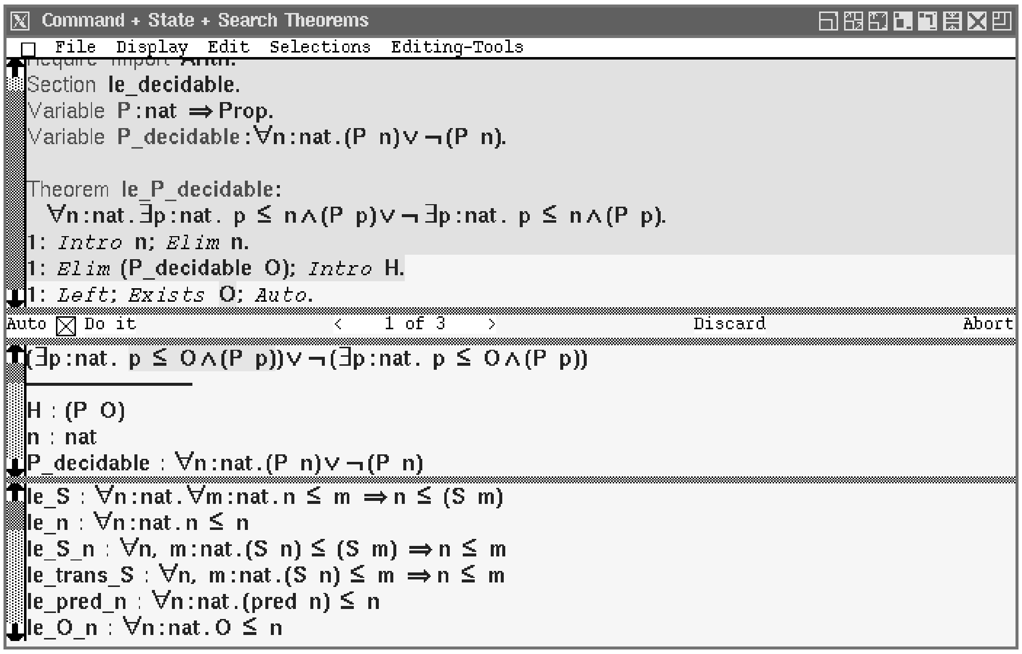

Figure 1. The CtCoq main window

contains objects from the global context, and a lot of operations are common to both kinds of context.

Beside the physical placement, there is also a behavior association. Each of the three windows can be used as a menu to build and edit commands for the *Commands* window, for example with the proof-by-pointing facility described below.

2.2 Pretty printing and notations

For pretty-printing matters, the CtCoq system relies on the Centaur display machinery, which has three important features. First, the notations are chosen through a tree pattern-matching mechanism, which is more powerful than what can be done with text editors as in the TkHOL or Emacs environments.

Second, the layout is computed incrementally and it is sensitive to the mouse, in the sense that it is possible to select sub-expressions of a formula, and to copy, paste, or trigger commands with these sub-expressions. This makes it more practical to use than layout approaches that use the LaTeX-xdvi chain, where the output is computed in a batch mode and is, thus, inert.

Mechanisms for extending notations and adapting these to specific mathematical domains are also provided, thanks to generic capabilities present in the Centaur system. A crucial aspect is that a pretty printer can be composed of several parts working together. To add a new notation, it is only necessary to add a new part and to express how it is composed with the existing ones. This operation consists of a few quick steps.

- Edit in the CtCoq interface a ten to twenty line specification describing the new notation pattern,
- Press a button to compile this file (because the specification is so small, this compilation is typically very fast),
- Tell the windows to refresh so that they use the new notation.

This cycle is very short, and a typical user will fine-tune his notation by repeatedly adding information for the line-breaking policy and the choice of fonts and colors in his notation pattern and trying the specification along the way. Still, adding a new notation requires some skill, as there is a particular specification formalism to learn. Figure 2 gives an example of the kind of notations a user can add. The two windows contain the same formula, with default notations in the bottom and special notations on top.

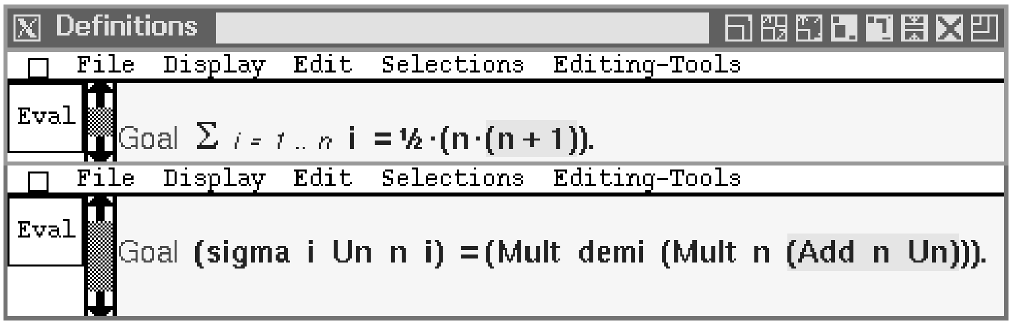

Figure 2. An example of user-defined notations

3 Advanced features

The CtCoq user-interface also provides elaborate solutions to problems often encountered in the usage of interactive proof systems. *Proof-by-pointing* is concerned with inputting the basic commands. It shows that in the domain of proof manipulation, the mouse can be used to guide a symbolic system in a much more clever way than a simple push-button user-interface. *Script management* is concerned with the support that can be given to a user who tries different solutions to a problem and wants to record his successful attempts. *Textual explanation of proof* provides support to a user who wants to produce documents from his proofs that any mathematician can read, even a mathematician who does not know the Coq system.

3.1 Proof-by-pointing

In the Coq system, most proofs are done interactively in a goal directed style. In this mode of operation, the user first states a goal to prove and then applies commands that reduce this goal to a number of simpler subgoals. Some of the commands have a very elaborate behavior, while others perform very simple logical bookkeeping. *Proof-by-pointing* is a method to ensure that the bookkeeping commands can be very easily triggered and composed, to make their use less tedious.

The idea of proof-by-pointing [1] is that selecting a position in a goal formula can be interpreted as a command to bring the selected sub-formula to the surface of the goal. In practice, the behavior of proof-by-pointing relies heavily on the ability of the graphical interface to construct graphical representations of logical formulas that are structured and sensitive to the mouse. With this algorithm, the user of the CtCoq environment ends up performing a lot of proofs using mostly the mouse to select relevant parts of goals, and sometimes choosing a more elaborate command from a menu.

3.2 Script management

The CtCoq proof environment supports the activity of interactive proof search. In this mode, the user may try several possibilities before finding the right formalization and the right sequence of inference steps. His work is then a succession of steps forward and backward. Script management is a tool to keep only the successful attempts, discarding the failed ones.

Script management works by separating the window containing the commands sent to the proof system into several areas. One area (called the final area) contains the commands that have already been sent to the proof system, another (the buffer area) contains the command being executed, a third (the queue area) contains the commands, the execution of which have already been requested but not yet started, and a fourth (the scratch area) contains the commands being edited by the user. When a command is undone, it is removed from the final area and returned to the scratch area.

The various areas are made visible by different colors. This way the user can see at a glance the current state of his work. This is especially useful when modifying the formalization of a problem and trying to recuperate large amounts of work that were valid in the formalization and that need to be checked against the new formalization. The final area acts like an "iron curtain": all commands in this area are *read-only* and can no longer be edited. In fact, to avoid race conditions between the user and the proof engine, it is also necessary that the buffer area and the queue area be read-only.

The interface provides a way to bring the iron curtain down, so it should also provide a way to lift it back up. This is done by undoing commands, which correspond to the undoing commands provided by the Coq proof engine. There are two kinds of undoing commands, depending on whether the target position is relative or absolute. The *Discard* button provides relative movement: it always undoes the last requested command (or cancels the last requested command when it has not been executed yet). Reset commands provide absolute movement: they make it possible to go back to a precise position in the script, regardless of the number of commands that need to be discarded. In the graphical user-interface, these reset commands are easy to trigger, as the user only needs to select the command to go back to with the mouse and type a keystroke.

A nice feature of this script management paradigm is that it makes replaying very easy. The user can simply select a collection of commands by dragging the mouse in the script window and request for these commands to be executed by simply pressing on a button. This operation simply lines up all these commands in the queue area. It is also possible to request for all the commands in the scratch area to be executed, with a single keystroke, by simply putting them in the queue area. This replaying facility is very useful when one has already put a large amount of work into a theory before discovering that one of the basic axioms is wrong or unproperly stated (this happens a lot in practice: *errare humanum est*). When this happens, the user need only click on the defective axiom, correct the formulation, and then request for everything to be executed again.

Future work in the area of script management aims at reducing the dependence between commands in the script, so that fewer commands need be undone when going backward and fewer commands need to be re-executed when replaying. This implies recovering dependencies between theorems and commands and developing a set of facilities that exploit these dependencies accurately and efficiently.

3.3 Textual Explanation of Proofs

The proof scripts written to check a proof under Coq actually represent the proof but they are unreadable to people who don't have a good working knowledge of the proof system's behavior. Fortunately, the Coq system also constructs a typed λ-term that represents the basic inferences used to prove a mathematical result. We are working on pretty-printing facilities that produce formalized English text from these proof terms. With a simple keystroke, it is thus possible to open a window containing the English text describing the proof of a theorem, or even a proof currently in progress.

The text production mechanism consists of two parts, with the first part embedded inside the Coq proof engine and the second part embedded in the CtCoq user-interface. The first part performs a careful analysis of the λ-term representing the proof we want to display to detect mainly three kinds of information. The first kind of information is information about type: the sentences are not worded the same depending on whether the objects are are plain data objects (like integers), sets (like the set of all integers), properties over objects (like being a prime number), or properties over sets (like being finite). The second kind of information is information about occurrence. Names are not given to objects in the same manner depending on whether these names will be used often or not in the following text. The third kind of information describes the usage of introduction and elimination theorems for inductive definitions. Typically, inductive definitions with only one introduction theorem (or one constructor), can all be treated the same, where a usage of the constructor or the elimination theorem corresponds to a usage of the definition.

In the CtCoq interface, all this information is used to display the λ-term with relevant notations. This also uses the Centaur display machinery. The versatility of this machinery makes that it is possible to have several views of the same proof, where one is the plain unsugared λ-term, the second is a proof text in English, and a third is the proof text in French.

In this framework, incomplete proofs simply appear as λ-terms where some sub-term is replaced by a place-holder that remains to be instantiated. These place-holders are decorated with the type expected for their replacement (in other words, the place holder are decorated with the statement that remains to be proved). In the proof text, these place holders are replaced by a sentence indicating that some proof still needs to be found. In Figure 3, readers can see a instance of incomplete proof text. The incomplete proof appears in the sentence "Imagine a proof of $\langle nat \rangle 2 \cdot (m \cdot m) = 2 \cdot x \cdot (x \cdot 2)$".

4 Conclusion

This paper describes an experience of user-interface for a proof system that is developed independently from the proof engine. In the long run, the aim is to construct a proof environment around the proof system, providing tools for expert users engaged in intensive work.

The first design principle is to make tedious tasks easier, regardless of their relevance to proof theory (although some features like proof-by-pointing and textual explanation of proofs do use basic concepts from proof theory). In this respect, a good example is the set of capabilities we call script management. The facilities present in the current version use very little knowledge of the way the proof engine operates. Still, these basic facilities provide a good added value because they make the proof engine easier to use in situations that are very frequent for expert users.

A second design principle is to provide customization whenever possible. The possibility to customize notations described in this paper is parallelled by a possibility to customize template menus for structure editing (not described in detailed in the paper, due to a lack of space).

5 Availability

The CtCoq system is available on Sun and Dec Alpha workstations. Information on how to retrieve this system is available at the following WWW address:

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

References

1. Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160, 1994.
2. P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).
3. D. Clément. A distributed architecture for programming environments. *Software Engineering Notes*, 15(5), 1990. *Proceedings of the 4th Symposium on Software Development Environments*.
4. C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant User's Guide*. INRIA, May 1995. Version 5.10.
5. L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes Computer Science*, pages 213–237, Nijmegen, 1994. Springer-Verlag.
6. D. Syme. A new interface for hol - ideas, issues, and implementation. In *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 324–339. Springer-Verlag, Sept. 1995.

Figure 3. A typical session with CtCoq