INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The CtCoq System: Design and Architecture

Yves Bertot

N° 3540

Octobre 1998

THÈME 2

Rapport de recherche

ISSN 0249-6399

The CtCoq System: Design and Architecture

Yves Bertot

Thème 2 — Génie logiciel
et calcul symbolique
Projet CROAP

Rapport de recherche n° 3540 — Octobre 1998 — 25 pages

Abstract: We present issues that arose in the design of the CtCoq user-interface for proof development. Covered issues include multi-processing, data display, mouse interaction, and script management.

Key-words: Automatic proof, Interactive proof development, graphical user interfaces, CtCoq, proof by pointing.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Le système CtCoq: conception et architecture

Résumé : Nous décrivons les problèmes qui sont survenus dans la conception de l'interface homme-machine CtCoq pour le développement de démonstrations sur ordinateur. Les thèmes couverts comprennent l'utilisation de plusieurs processus, l'affichage de données, l'interaction à la souris, et la gestion de scripts

Mots-clés : Démonstration automatique, Développement interactif de preuves, Interfaces graphiques, CtCoq, preuve par sélection

1 Introduction

In [TBK92, BT98] we claim that computer aided deduction systems need powerful user-interfaces and we lay out general techniques to construct such user-interfaces, based on a multi-process architecture and tools coming from programming environments. Following these ideas, we have developed a specific user-interface for the Coq system [DFH+93], called CtCoq [BB96b, BB96a], using the programming environment generator Centaur [BCD+88]. Over the years, CtCoq has been used to perform larger and larger proof developments [BF95, Thé98, Ber98] and to experiment with new interaction ideas. For instance, we have proposed new techniques to guide the proof process using the mouse, with proof-by-pointing for predicate calculus [BKT94] and drag-and-drop for algebraic manipulations [Ber97a]. We have also described tools to help undo selectively past commands [Pon97] and, more generally use dependencies between commands or mathematical objects [PBR98]. We have accumulated a wealth of experience regarding the integration of a variety of functions, their use in practical proof development and maintenance, and the ease with which these functions can be maintained as the proof system and the context evolve.

In [Eas98], the author makes sure she clearly states what the aims of computer theorem proving may be, and gives four possible categories: *proof validity* (where the actual content of the proofs does not matter and automatic proof search is acceptable), *proof checking* (where the user provides a complete proof that only needs to be checked), *proof discovery* (where the proof organization matters and the computer brings support in working it out), and *educational purposes* (where the proof matters so much that the basic steps of reasoning have to be spelled out). In our experience, it appears that the scale factor is relevant. When the proof size increases, the three first categories tend to merge into one: to date no automatic proof tool can cope with the discovery of significant proofs, and humans rarely have the courage to spell out large proofs in such detail that proof checking will be sufficient to mechanically verify the whole proof development without automatic proof discovery.

One area that Eastaughffe did not consider is *proof maintenance*. Once you have a proof for some mathematical fact, valid under some assumptions (basic definitions or axioms), it is relevant to consider tools that help finding proofs for similar facts under some changes of the assumptions. In large scale proof development, users frequently have to start maintaining activities even before the whole proof development is over: the definitions given when starting a project have to be re-considered when addressing later parts of the proof. These aspects have also influenced our design of CtCoq, even though our maintenance tools may be less mature than the tools designed for other activities. In this domain, commands scripts are *precious* data. We have to design the working environment so that users cannot destroy their scripts by erroneous manipulation or by failure of the tools. Keeping large scale development in mind, it is also understandable that we paid less attention to beginners than in other user-interfaces, like those aimed at educational purposes. We have to maintain a balance between the powerful functions we want to provide and the over-simplification that would be required to make a beginner feel nice and cosy. As a result, the CtCoq user-interface requires a longer training than is usually required for a graphical user-interface. Still we claim that this approach made it possible to explore many interaction methods, which a more "principled" design would have ignored.

In this paper, we want to review the principles enumerated in [BT98], see how these principles are instantiated in the case of CtCoq, and describe the extensions that were devised for this experience. This paper is structured as follows. We first review related work. Then we describe the architecture of the CtCoq system. Then, we review the characteristics of data display and extensible notations. In section 5, we review the tools to guide proofs using the mouse. In section 6, we review the tools for script management. In the last section, we draw conclusions and explain possible directions of research.

2 Related work

Tools for the proof of theorems on a computer have been developed for a long time. However, the choice to put a specific emphasis on interaction and user-interface has appeared more recently, thanks mostly to the increasing use of graphical workstations. Developments for which a significant effort was put into the design of the user-interface include the Nuprl system [CAB$^+$86], the Interactive Proof Editor [Rit88]. The closest experiments to ours are the GLEF$_{\mathrm{ATINF}}$ interface [CH95], the logical framework ALF [MN94], the graphical interface TkHOL [Sym95], XBarnacle [LD97], and XIsabelle [OEC97], where the proof process and the graphical interface are implemented as separate processes.

More distant from our work are the experiments in the development of tools designed to teach basic logic. An interesting set of guidelines for such systems is described in [Sup84]. More recently educational systems have been listed in the review [GRB93]. Among these systems, it is interesting to note the Jape system [BS98, SB98], because it uses direct manipulation features that are similar to the ones presented in this work.

Very few general man-machine interaction principles have been used in the design of CtCoq. Still, several features can *a posteriori* be seen as adhering to such principles. For instance, our work on proof-by-pointing is related to the equal opportunity guideline advocated in [Thi90]. Also, our design agrees with [MH97] in asserting that a "document" oriented approach makes it easier to reduce viscosity, although our documents are not plain text as mentionned in that work.

3 Architecture

3.1 Basic principles

One debated feature is the importance given to structure directed editing. In CtCoq, the basic data manipulated by the user is a tree-like structure, even though it appears on screen as lines of characters. Gateways to textual forms have been provided, as will be seen in section 3.2.1, but important commands require structured data, and any textual data must be parsed before use. The result is a mixed blessing for beginner and expert users alike. When manipulating text fragments, users need to learn the *concrete* syntax of the language used.

This is easily done by reading correct examples. When manipulating tree-like structures, users need to learn the abstract syntax that states which sub-structures are allowed in which context. It is not enough to see correct examples to understand how they were built because display notations hide the real structure. Still, powerful notations are a must to make a usable mathematical tool. The tree structured data manipulated in CtCoq is rendered on screen using a pretty-printing procedure. This pretty-printing procedure can make use of a variety of fonts, size, and colors to render mathematical formulas more readable. Moreover, CtCoq provides ways for the user to adapt the notations to their taste. This capability is also provided in other interactive tools such as Jape [BS94] and Mathspad [BVW97] but, not surprisingly, they also use structured manipulation.

3.2 Multi-processing issues

As we already advocated in [BT98], CtCoq communicates with Coq using TCP sockets, using a different protocol in each direction. Commands sent to Coq are sent as plain text in Coq's input syntax. Data coming back from Coq is transferred as a tree-like structure, using the protocols of [DR94] to encode data.

For large scale proof development, it even proved worth the effort to implement a distributed approach, using different machines to run the various processes: in this setting the two processes do not compete for cycles and memory on the same machine. This significantly improves the user's comfort.

Several complex issues have to be considered:

- Separating parsing and logical processing,
- Maintaining the protocol procedures,
- State update and efficiency,
- Interrupting the logical engine.

.

3.2.1 Parsing and logical processing

Although the only first class data in the interface is tree-like structures, CtCoq also provides ways to input textual data. For this, a parser is required. Developing a parser from scratch or from a separate abstract description raises maintenance problems, as it is necessary to check that the independent parser works like the one in the Coq prover.

Instead we used the parser from Coq. This solution is made easier by the actual architecture of the Coq system. In Coq, there is a central data structure called `CoqAst` that is used as a pivot for communication to and from the user. From the user, this data structure is the one produced by the parser and consumed by the logical engine. To the user, this data structure is the one produced by the logical engine and consumed by the plain Coq pretty-printer. To communicate with the user-interface process, it is then enough to implement a translator from this `CoqAst` structure to the tree structure used in the user-interface. This translator still represents a significant amount of code, although we describe in section 3.2.2 the solution that we devised to make this code easier to maintain. Still, this has proved more practical than maintaining a different parser.

Our design ensures that parsing activities and logical processing can be performed concurrently. Instead of using the Coq system to handle both kinds of requests, we have reverse engineered the Coq sources to extract a smaller program that only takes care of the parsing. Thus, the CtCoq user-interface can handle parsing requests even if the Coq process is busy performing a logical processing request.

The Coq system also provides extensible syntax, in the sense that users can provide new grammar rules that are merged into the parser. This capability is not compatible with the usage model of structured editing, as it renders parsing a non context-free process. For instance, if a grammar rule definition occurs in the middle of the file, the data in the file up to the definition should be parsed using one configuration of the parser and the data in the file after the definition should be parsed using another configuration. In bare Coq, this is not an issue, as one is not supposed to parse data from the beginning of the file after the grammar rule has been merged into the parser. In the CtCoq interface, the usage model we have is that the file is a document in which users navigate randomly, so that they may edit fragments in any order: this would require that the interface keeps track of all the parser configurations used in the file and the exact areas where these configurations should be used.

To cope with this issue, we chose a weak approach: we accept syntax extensions but all data is parsed with the current configuration of the parser. This may lead to inconsistencies, but no problems will occur as long as entire proof developments are done using the same syntax. Syntax extensions may be loaded once and for all at the beginning of the session. We provide ways for the user to set up configuration files so that their preferred syntax extensions are loaded automatically when the parser starts. We also make it possible to restart the parser from scratch in the middle of a session. As the user may change the configuration file between different restarts, this gives a poor man's capability for variable syntax.

3.2.2 Protocol production in the Coq System

To send data from the Coq process or the parser process to the CtCoq interface, we use the tree transfer protocol described in [DR94]. This protocol uses a textual representation of tree-like structures based on a postfix traversal of the structures.

On the emitting side, the data to transmit is in the `CoqAst` datatype. This datatype actually provides a very simple notion of trees: it has very few operators: `Node` (with a string and a list of `CoqAst` terms as arguments) or `Var` (with a string as argument), for instance. The `CoqAst` datatype is sufficient to encode all the structures that occur in Coq commands, but it does not enforce any form of syntax discipline.

On the CtCoq side, the text received must correspond to the abstract syntax actually used in the interface. This abstract syntax is described using 350 operators and as many syntactic categories. These operators and syntactic categories make it possible for the structure-editing tools to enforce constraints that ensure that the user will only construct syntactically[1] legal commands.

As far as the communication protocol is concerned, we must ensure that the text emitted by the Coq process or the parser process correspond to a valid tree with respect to the constraints on the receiving side. The code needed is a function from the weakly constrained datatype `CoqAst` to the weakly constrained datatype `string`, making sure it respects the constraints enforced in another program. This kind of program is hard to write, hard to debug, and hard to maintain.

A solution, proposed and implemented by Healfdene Goguen, is to develop a two-pass approach that uses mechanical verification in one pass and mechanical code generation in the second pass. The idea is to introduce an intermediate ML datatype that provides a strongly typed description of the commands accepted by the Coq system. This description is a large set of mutually recursive types corresponding to the syntactic categories used in CtCoq, each type having constructors corresponding to the operators used in CtCoq. The translation is not completely straightforward, because some abstract syntax operators may belong to several syntactic categories while ML constructors may only belong to one type. We developped a program that takes the abstract syntax description from CtCoq and automatically generates the intermediate ML type and functions mapping terms in this type to valid strings for the communication protocol. These functions implement the second pass of our approach. The program is developed once and for all: it does not need to be updated when the set of commands accepted by Coq evolves from version to version. The first pass is a program that maps `CoqAst` terms to terms in the intermediary ML type. This pass must be written by hand and needs to be updated when the set of commands evolves, but the ML type-checker comes in to help detect errors, by checking that the returned data respects the type discipline.

Implementing this approach has been a real progress in the design of CtCoq. Nevertheless, the code for the first pass still remains tricky to maintain, since we do not have a declarative description of the `CoqAst` terms that may occur in a session. Until now, we have relied on reverse-engineering the code of the Coq system to discover this information. We think that the Coq process should not only provide a formal description of its concrete syntax, but also a formal description of its abstract syntax.

A drawback of this approach is that it is inefficient: the first pass constructs a term in the intermediary ML type that will only be used once when displaying it as a string. A possible extension of our approach would be to use some partial evaluation tool to merge the two passes into a program that produces the same output without producing any intermediary data, following a deforestation technique [Wad88]. Here again, the partial evaluator could easily be written once and for all, using a formal description of the caml language, as found in [RT97].

[1] By "syntactically" we mean that the constraints are only context-free constraints.

3.2.3 State update

The CtCoq user-interface must display a clear view of the Coq process' state. As a result, data must be duplicated between the logical engine and the user-interface. The naïve solution is to have the logical engine send a complete description of its state after each interaction. This inefficient solution does not fit with our objective of supporting large scale proof development.

In our design, we used two observations. First, it is often possible to foresee that modifications will only occur in some place. For instance, during goal directed proofs there may be several goals pending, but proof commands will often work on only one goal. It is possible to design the communication between the two processes in a way that only modifications on this goal are communicated between the two processes. This aspect was already advocated in [BT98].

The second observation is that a complete view of the Coq process' state is not necessary all the time. For instance, expert users spend a lot of time replaying long fragments of scripts with a low error rate. In this case, they do not need to see the intermediary states displayed: they are only interested in seeing the state at the end of execution or when an error occurs.

Taking these observations into account, we have designed a communication protocol where the logical engine only sends a summary of its state to the user-interface after executing each command. In return, the user-interface computes the data it actually needs and requests it from the logical engine. This protocol implements two modes of communication: *replay* and *regular*. In *regular* mode, the user-interface receives the information summary, computes the goal it wants to display and requests only this goal from the logical engine. In *replay* mode, the user-interface simply acknowledges that the current command has been executed and sends the next command. If an error occurs while in *replay* mode, great care is taken to stop the execution, change the mode back to *regular*, and request the relevant information.

3.2.4 Interrupting the prover

Some of the commands available in the logical engine may provoke lengthy or looping computations. The user-interface is not complete if it does not provide ways to interrupt these computations. The unix environment, in which the system runs, provides ways to interrupt "instantly" a process using a notion of signal. The user-interface must also provide this capability. In CtCoq, it is provided by a button in a specific menu.

The first issue with respect to this capability is related to multi-processing and distributed computing. Since the two processes may not run on the same machine, it may become difficult for the CtCoq process to find the Coq process identification before sending it a signal. We solved this issue by adapting the tool encapsulation tools described in [Clé90] and called *stservers* in [JMB+93]. A small process called a `stserver` is added between the user-interface and the logical engine. This process acts as a broker. It runs on the same machine as the logical engine, communicates with it through its standard input, output, and error character streams, and may occasionally send a signal. On the other side, the `stserver` process communicates with the user-interface process using two bi-directional sockets. One socket is used to transmit the standard input and output character streams, the other socket, is used to transmit the standard error stream (from `stserver` to CtCoq) and the fourth direction is used to send "signal" messages from CtCoq to the `stserver`. In normal use, the `stserver` process only copies the data from one process' output to the other process' input. When receiving a message on the signal line from CtCoq, it does not copy it, but simply sends a signal to the Coq process, whose process identifier is known since creation time.

A second issue is that we have to make sure the communication protocol is not broken by signals coming at random times. Messages sent by Coq to CtCoq all have a regular form, following the encapsulation proposed in [TBK92]. Messages have a header (used on the receiving end to decide how to parse the contents), some contents, and an end marker. If the Coq process is interrupted between the moments when it outputs the header and the end marker, the receiving end protocol procedures get stuck, expecting an end marker that never comes. The solution is to carefully protect the data emitting sections with a signal handling function, which will memorize the fact that a signal was received instead of brutally interrupting the computation. Of course, the protection should only be put in place for short periods of time and it is necessary to check whether a signal was received when exiting the protected section.

3.3 Screen real estate

Since our first experiments, we have designed interfaces that provide many windows, each adapted to displaying one kind of data or performing one kind of activity. To make sure the screen does not become overcrowded to a point that the user's productivity decreases, it is important to control the number of windows created. A good solution to this is the composite window already presented in [BT98] and shown in figure 1.

In a large proof development, users may reach situations where they discover the need for a new lemma in the middle of a large proof. In such situations, it is comfortable to be able to open a new session with the Coq process, working practically in the same context, to prove the missing lemma and then resuming with the initial proof. The CtCoq user-interface provides this capability: several composite windows may be open at the same time, each one hosting a virtual session with the Coq system. This capability improves the usability of the proof environment.

Users have the possibility to work simultaneously with several virtual sessions, sending commands randomly from several composite windows. It is important to make sure the right composite window receives the results. For this, every composite window is given a unique identifier. The commands sent to the proof engine are encapsulated with a header that also gives this identifier to the proof engine. The commands are processed in the logical engine in a regular fashion but results are also encapsulated with a header that gives the window identifier. Inside the user-interface process, the message is then routed to the right composite window.

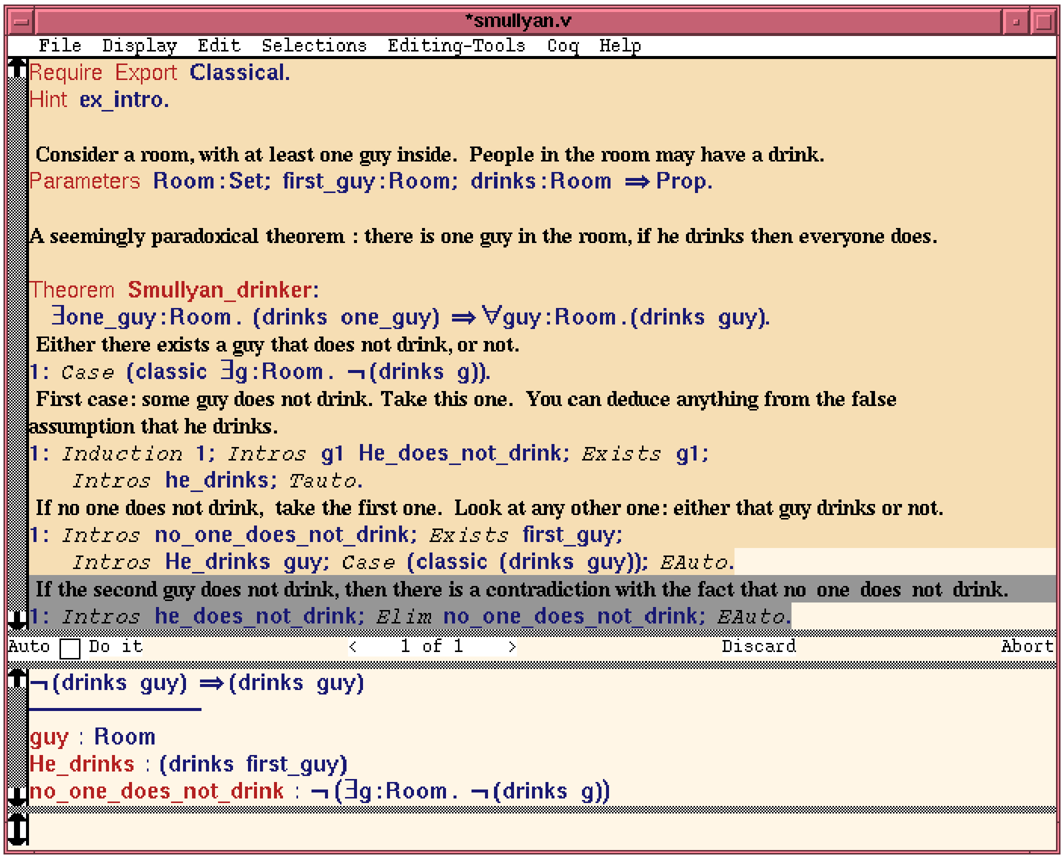

Figure 1: A composite window. The top area contains the script. Different backgrounds are used to distinguish commands that have already been executed and the "current" command. The middle area describes the current goal. The text above the short horizontal bar $(\neg(drinks\ guy) \Rightarrow (drinks\ guy))$ is the expression to prove, the text below the horizontal bar $(guy : Room \ldots)$ is a list of named premises describing known facts. The bottom area lists theorems when the user requests it. The relative size of the various subwindows can be adjusted at will by the user using the large horizontal bars.

3.4 Safety issues

Our design helps preserving scripts from erroneous manipulations, using three features.

The first feature was to separate the script window into several areas, where the area containing the commands that have already been executed is read-only. Enforcing this constraint is relevant because editing commands and sending these commands to the logical engine are two different steps and because the executed commands and the yet-to-be-executed commands are stored in the same editing area. We guess this issue is meaningless in systems like Alf [MN94], Jape [BS94] where editing and proving are merged in one single activity.

One conventional way to help protect precious data is to have the user-interface save regular backup files on disk. We have also implemented this capability, by including an object in our development that sets up regular alarms and sends a message to all composite windows to save the contents of their script sub-window. In this case, the scripts are saved in tree form rather than in regular text form, so that commands that are incomplete when the backup is performed will not prevent reloading the script in another session.

We also went one step further by setting up a memory safety. This safety provokes the termination of the user-interface when the process runs short of memory. When this condition is detected, a memory reserve is freed, the backup mechanism is triggered (assuming there is more room in memory), and the processes are killed. This safety mechanism works by checking the statistics of the garbage collector. It also provides a warning mechanism that will tell the user when memory reserves are going low, but much sooner than the more drastic termination procedure is triggered.

4 Display

The tree-like structures manipulated in CtCoq are transformed into text using three different procedures, to save in files, to communicate to the Coq process, or to display on the screen. An advantage of this situation is that the notations used on screen do not need to be the same as the ones used for storage and for communication with Coq. The objectives are different. For storage and communication with Coq, the text needs to be unambiguous but it does not matter whether the notations used are short or beautiful. On the other hand, the data displayed on screen should be short and beautiful to help readability, and some amount of ambiguity is acceptable.

In mathematics, new notations are customarily introduced to handle specific domains. For this reason, it is also interesting to let users of CtCoq adapt the notations that appear on screen. The Centaur programming environment generator [BCD+88], on which CtCoq is built, provides an abstract language, called PPML (Pretty Printing Meta Language), to describe the layout mechanism. This layout mechanism makes it possible to use a variety of fonts and colors, as could already be seen in figure 1.

The layout mechanism can be configured by the user in several ways. First, users can change colors and fonts at will using ressources. Each character is displayed with a class,

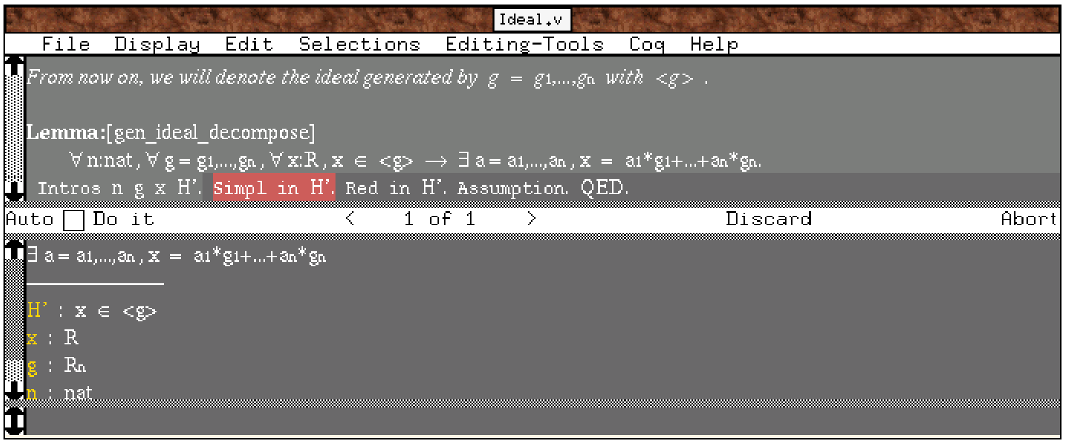

Figure 2: Notation for linear algebra calculus. Smaller fonts are used for subscripts. This example, courtesy of Loïc Pottier, shows how CtCoq is used to generate teaching material for first year students.

and the background color, foreground color, and font associated to each class can be tuned to the user's taste.

The second way to configure layout is to add pretty printing rules to the description. This is possible because Centaur provides a mechanism of layered pretty printers: the layout is not controlled by one specification file, but by several files placed in a list. Each specification file contains a collection of rules, each mapping a tree pattern to a layout pattern. For a given node in the displayed tree, if no tree pattern from the first file matches, then the layout mechanism looks up in the second file, and so on. Different lists of specification files are used for printing in files and for displaying on the screen. This way, users can have pretty but ambiguous notations to work with on screen and ugly but precise notations to save their work on disk.

Usually, the first element in the specification file list is a small file that describes the user's preferences. Figures 2 and 3 show two examples of user-prescribed notations. The layout mechanism also provides postscript output that makes it possible to produce insertions to include into paper descriptions of the mathematical developments performed using CtCoq.

The PPML machinery provides only text-like layout, with an abstraction based on layout boxes to perform indentation. The examples given in figures 2 and 3 may mislead the reader into believing that this is sufficient for general mathematical notations. It is not. The exponentiation examples in figure 3 are only possible because one of the fonts available in the window system contains a character that represents the number 2 in exponent position. Exponents other than 1,2, and 3 are not available. Similarly, the subscripts used in figure 2 are only obtained by using a smaller font.

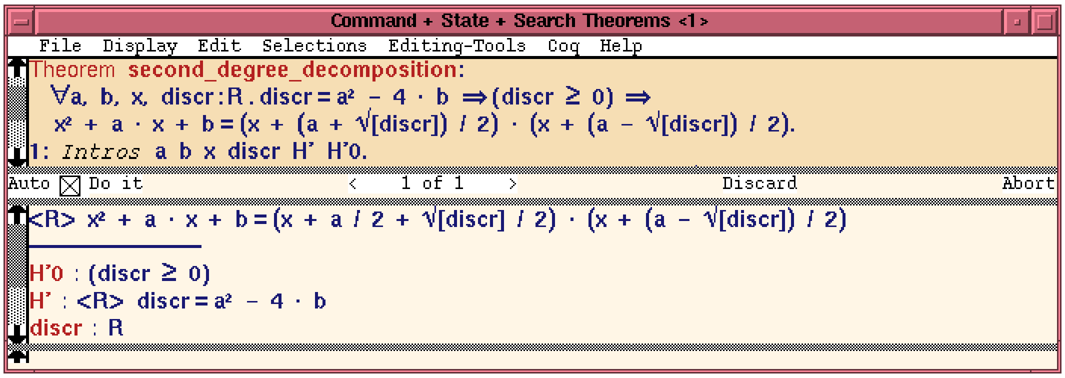

Figure 3: Notations for real number algebraic calculus.

With these easily adaptable display mechanisms, CtCoq also provides the possibility to have several views of the same data at the same time, with a different layout specification for each view. Thus, it is possible to open an auxiliary "table of contents" window, that displays only the definitions and theorem statements, but not the proof commands. The cursor can then be moved around the script by clicking in one window or the other.

5 Mouse directed proof

The most spectacular features of CtCoq revolve around its capability to let the user guide proofs using the mouse in a very efficient way. Of course, one could consider providing menus with all the Coq commands, but this would be much less efficient than usual text interaction. Instead, we have implemented proof-by-pointing [BKT94, BT98] and drag-and-drop [Ber97a].

The first facility interprets clicking actions as a language to perform proofs in predicate calculus. For instance, let us consider the following goal:

$$\forall x : nat.\ (\exists y : nat.\ x = 2 \times y) \Rightarrow even(x).$$

If the user simply clicks on the second x and chooses the option "rewrite" in the proof-by-pointing menu, then the gesture is interpreted as a command that will:

1. introduce a constant x on which to reason,
2. introduce an hypothesis $\exists y : nat.\ x = 2 \times y$,
3. introduce a constant y and a hypothesis $x = 2 \times y$,
4. produce the goal $(even\ (2 \times y))$ where x has been rewritten according to the last introduced hypothesis.

The second facility interprets dragging the mouse as a language to perform rewriting. For instance, let us consider the following goal:

$$x + ((a + b) + z).$$

Dragging the mouse from x to a will provoke a rewriting operation that permutes x and $a + b$ leading to the following expression:

$$(a + b) + (x + z).$$

In the next two sections, we study separately the implementations of these facilities in CtCoq. The important idea in the first section is that the proof-by-pointing algorithm is a compiler from gesture to proof commands and that this compiler needs an optimisation phase. The important idea in the second section is that tools to interprete mouse gestures can be extensible, with various levels of automation in the extension process.

5.1 Design issues for Proof-by-pointing

Precise descriptions of the proof-by-pointing algorithm can be found in [BKT94, BT98]. Let us only summarize the main characteristics of this algorithm. Proof-by-pointing is used when the user clicks with the mouse on a sub-expression of a goal and asks for a command to be generated. The algorithm receives as data the complete goal as a tree-like structure and the path from its root to the branch that was selected. Computation then proceeds recursively, going down the tree structure following the path. At each step, an elementary command is generated using only information on the tree operator that is traversed (a local pattern on goal sub-expression) and the next elementary step in the path. All the commands are chained together to form a large composite command that may be executed to construct new goals.

In [BKT94, BT98], we show that the algorithm for proof-by-pointing can be implemented in various ways. One of the important choices is whether the actual logical processing is performed in the logical engine or the user-interface. If the processing is performed in the logical engine, then the command that is sent to the theorem prover only needs to receive as argument a description of the path, since that process already holds a description of the goal. The same command should be recorded in the script. A drawback is that the goal does not appear in the script and it is then difficult to understand the operations that have been performed only by looking at the command. This matters a lot when considering user-friendliness and script maintenance issues.

In CtCoq, we have decided to implement the algorithm inside the user-interface and make it use plain Coq commands. The proof-by-pointing algorithm generates a large composite command that is sent to the script window. In this respect, the proof-by-pointing facility actually works as if the state window, where the goal is displayed, was simply used as a "structured" menu where the user can click to choose a command that will be inserted in the proof window. Proof-by-pointing then becomes an editing command like any other. Thus, proof-by-pointing generated commands can be edited by the user before being sent to the logical engine. Manual editing is actually often required, because generated commands often contain holes that need to be filled in before the command is sent. As a result, at least two mouse clicks are needed for every proof-by-pointing action: one to generated the command and the other to send this command to the logical engine. To avoid this annoying extra click, we also provide the possibility to set a flag that expresses that commands generated by proof-by-pointing can be sent right away to the logical engine when they contain no holes. This capability is provided by the little check box that appears between “Auto” and “Do it”, unchecked in figures 1, 2 and checked in figure 3.

If the proof-by-pointing generated commands are to be recorded in the script, it is important to make them as readable as possible. For this reason, it has also been necessary to devise a tactic simplifier [Ber97b]. The proof-by-pointing algorithm should be understood as a compiler that transforms symbolic descriptions of the user's movements into commands in another “programming” language. In this case, the tactic simplifier is the optimizer.

The output from the first phase of the proof-by-pointing algorithm, as it is described in [BKT94, BT98] is a composite tactic that follows very strictly the path from the root of the goal down to the expression selected by the user. Each step is constructed independently, with very little information about the surrounding steps. The job of the tactic simplifier is to look at command subsequences and recognize opportunities for compaction, often replacing sequences of elementary steps by one step that performs approximately the same operation. The fact that the simplified tactics are only an approximation of the original ones is acceptable as long as the main intent of the proof-by-pointing gesture is always respected.

The algorithm works by several traversals of the composite tactic from top to bottom, each traversal checking whether some collection of rewriting rules apply.

For instance, one of the first passes applies a rule for compacting introduction commands (commands that make it possible to go from a goal $A \Rightarrow B$ to a goal B with the right to use A).

$$\texttt{Intros}\ x_1\ \ldots\ x_n\texttt{;}\ \texttt{Intro}\ x_{n+1} \rightarrow \texttt{Intros}\ x_1\ \ldots\ x_{n+1}.$$

When designing this simplifier it becomes apparent that a carefully designed tactic language makes user-interface construction easier. The more uniform the tactic language is, the easier it is to provide automatic tools to manipulate composite tactics. In the case of Coq, we have shown that many tactics handle premisses of implications in a uniform way: these tactics are called *head tactics* in [Ber97b].

The proof-by-pointing facility helps making proof scripts that are more easily maintained. Typically, the generated commands describe more exactly the proof steps than the commands usually written by hand in the same situation. One sensitive issue revolves around named hypotheses. As opposed to the proof-by-pointing algorithm that only introduces hypotheses with explicit names, the commands written by hand usually leave names unspecified. Still, later commands will use names to refer to hypotheses. If no explicit names were given at introduction time, changes may occur without warning. This is a tricky issue, because name changes lead to errors that are detected long after the deed is done.

5.2 Drag-and-drop Rewrite

The proof-by-pointing algorithm only takes care of predicate calculus. A lot of mathematical reasonning is more algebraic in nature: using general equations to replace expression by equal ones. Users change the shape of the formula to prove until a simple form is recognized. Proof-by-pointing is almost useless for this process. We observed that many algebraic operations actually are used to combine data or move data around. From this observation stemmed basic notions for an interpretation of drag-and-drop movements that can be implemented in a fairly simple way. We now explain this interpretation.

5.2.1 Classes of equations

When the same symbols occur on both sides of an equation, this equation can often be given an operational intuitive meaning, stating that some symbol moves around or that two symbols combine to produce a new result. Let us enumerate a collection of possible equation forms.

1. $x + y = y + x$ or $x + (y + z) = y + (x + z)$. These equations permute two expressions.
2. $x + (y + z) = (x + y) + z$ or $x \times (y + z) = x \times y + x \times z$. These equations rearrange the relative position of two operators.
3. $-(-x) = x$. This equation combines two operators.
4. $-(1/x) = 1/(-x)$. This equation permutes two operators.
5. $x - x = 0$. This equation combines two expressions to lead to a new result.

When formulas are displayed on the screen, it is possible to convey the operation of using these equations (from left to right) with only a gesture of the mouse. In case (5), the user only needs to select one of the x's and drag it to the other. In case (1), they need to select y and drag it to x. In cases (3) or (4), they need to select the minus sign and drag it to the other operator. In case (2), they need to select the parentheses and move them to the other + sign for the first equation, and select the x and move-it to the y or z for the second equation. Note that the reverse operation can also be expressed using a gesture for cases (1), (2), or (4).

5.2.2 Implementing drag-and-drop

The drag-and-drop engine is based on a simple algorithm that takes three pieces of data as input. The first piece of data is the tree-like structure T of the goal expression displayed on the screen. The second piece of data is a pair of paths p_1, p_2 in this tree-like structure to indicate the start-point and end-point of the user's gesture. The third piece of data is an ordered list of drag-and-drop rules, where each rule combines a pattern P, a pair of paths p_f, p_l, and an incomplete command C. The output of this simple algorithm is a command that can be sent to the logical engine.

The algorithm first computes a third path p_3 from pair p_1, p_2: it is the longest common prefix to the two paths, actually this path points to the smallest expression that contains both the start-point and the end-point. Then for each rule, the algorithm performs the following steps:

1. Compute the longest common prefix p_r to the paths in the rule.
2. Compute the path p such that $p_3 = p \cdot p_r$, when such a path exists.
3. Check that $p_1 = p \cdot p_f$ and $p_2 = p \cdot p_l$.
4. Check that the subterm of T at path p is an instance of the pattern P for some substitution σ.
5. Apply σ to C.

Note that steps 2, 3, or 4 may fail. In that case, the algorithm just skips to the next rule. Also step 1 can be performed only once for each rule and pre-compiled in the rule description.

From this description, it is obvious that the order of rules matters. Actually, if two rules (P, p_f, p_l, C) and (P', p'_f, p'_l, C') are such that there exist a substitution θ and two paths q and r so that the following equations hold

$$\begin{aligned} P' &= \theta(P) \\ p'_f &= p_f \cdot q \\ p'_l &= p_l \cdot r \end{aligned}$$

then the rule (P', p'_f, p'_l, C) will never be used if it occurs behind the other in the list of rules.

To make the generated command obey precisely the user's gesture, it is necessary to add a restricting command to express where exactly the transformation is to occur. For instance, let us suppose that the goal expression is the following:

$$(x + y) + (x + y) = (y + x) + (x + y)$$

and let us suppose that the user dragged the last (rightmost) x to the last y. The command generated by the drag-and-drop algorithm is:

```
Rewrite (plus_sym x y)
```

Applied directly, this command would lead to the following expression:

$$(y + x) + (y + x) = (y + x) + (y + x).$$

In this case, the command provokes the rewriting of all three instances of the expression $x + y$, including instances that have not been designated by the user. To avoid this, the drag-and-drop engine inserts a restricting command (the `Pattern` command in Coq syntax) that makes it possible to indicate on which of the possible instances one wants the actual rewriting to occur.

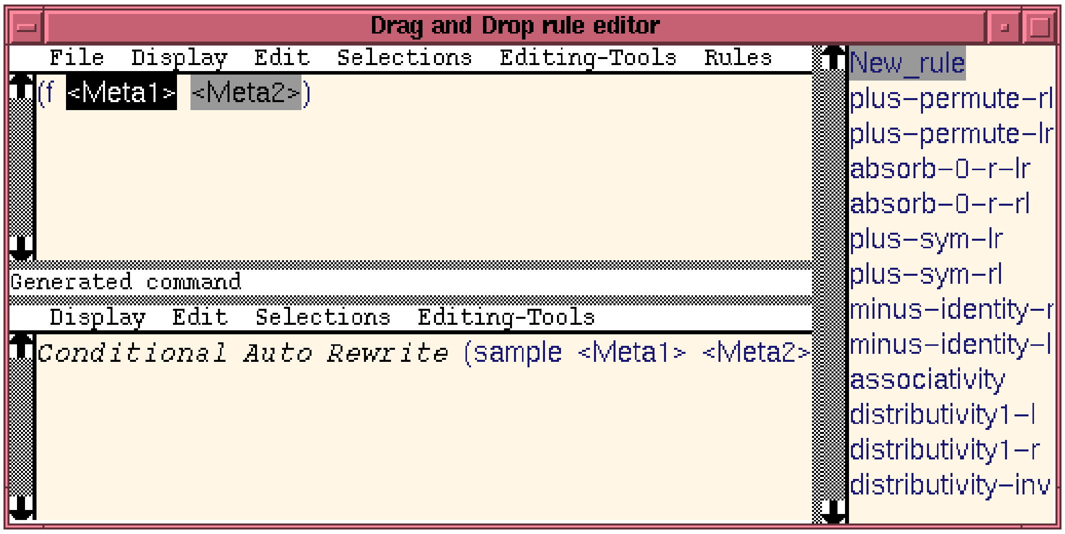

Figure 4: The drag-and-drop rule editor. The top left window contains the formula pattern and indicates the start and end positions. The bottom lef window contains the triggered rewrite command. The right window shows the ordered list of rules.

5.3 User-level extensibility

We have implemented tools to make the drag-and-drop engine easily extensible. Two directions have been studied, more or less based on automation.

5.3.1 Automatic analysis of rewriting theorems

There are several classes of theorems for which one can introduce drag-and-drop rules with systematic shapes. We have developed a procedure that analyzes the statement of theorems and constructs the corresponding drag-and-drop rules, when these theorems fall in the pre-defined classes. The various classes correspond approximately to the theorem forms described in section 5.2.1.

For each theorem, the procedure will typically construct up to four drag-and-drop rules. Basically, the equation can be used in two ways, either to produce instances of the right hand side or instances of the left hand side. Also the procedure produces the rules to interpret users' movements from left to right and from right to left.

5.3.2 Manual editing of drag-and-drop rules

It is also possible to add rules one by one, using a specialized editing tool to manipulate the pattern, the command, and the two paths. This tool is shown in figure 4. The rule editor is designed so that adding a new drag-and-drop rule is comparable to programming by example. To enter the pattern, users only need to get an instance of this pattern from the other CtCoq windows using plain copy and paste actions. To enter the command, they only need to type an instance of this command. To enter the two paths, they only need to give an example of the movement by dragging the mouse in the pattern window. Still, special attention must be paid to make sure the pattern is "generalized" by inserting "meta-variables" in the positions where the pattern can be instantiated and to make sure the corresponding meta-variables are used at the right place in the command pattern.

The rule list can also be manipulated directly in this editor, in two ways. First, one of the options proposed in the menus is a "sort" option, that will make sure that the rule list is re-arranged to avoid that general rules hide more specialized ones. Second, the user can manually change the place of a drag-and-drop rule, simply by dragging and dropping it in the right place.

6 Script management

The script management support in CtCoq encompasses a collection of commands to help visualize the structure of proof developments, return to past states, and keep coherent scripts. The main originality in this work is to have defined an insertion point in the script window, where executed commands must be inserted. This insertion point makes it easy to ensure that executed commands are recorded in the exact order of execution. The rest of the machinery takes care of making sure the contents of the script window down to the insertion point actually represent the current state of the proof engine, especially when the insertion point should be moved backward, for instance because of undone commands.

The script management facility as described in [BT98] is a broker that receives the requests from the user and forwards them to the proof engine and receives the answers from the proof engine and forwards them to the user, updating the list of executed commands as the process goes on. In CtCoq, departures from the basic principle have had to be implemented to handle specific aspects of Coq and CtCoq: varieties of undo commands and interaction between multiple windows and script management.

6.1 Varieties of undo commands

The script maintenance tool described in [BT98] only uses two kinds of undo commands, that go backward in history in a relative manner: the `Undo` command undoes the last command, while the `Abort` command goes back to the last "start" command. We have also experimented with a `Reset` command that refers to the point of return in an absolute manner. Given a theorem name, the `Reset` command undoes all the commands that occurred after the proof of this theorem, this proof included. The dialog with the proof engine is simple, as only the `Reset` command is sent. On the user-interface side, it is important to move the *executed* zone of the script to the correct place.

Another variety of `Undo` command that has been integrated in the user-interface is a *logical* undo [Pon97]. This command is available when the user-interface has enough information about the actual dependencies between all the commands that were executed after the undone command. In this case, it is possible to undo only the commands that depend on the undone command, still keeping in the executed script commands that were executed later but are independent.

Chronological undo can be performed without moving data around in the proof script. Only the end of the *executed* zone is moved from the current position to the undone position. As a result, undoing is an operation with low viscosity: after having undone a large fragment of script, it is possible to “undo the undo” at a relatively low cost by simply replaying the undone fragment (for the user this will require between one and three mouse operations). With logical undo, it is necessary to move data around, to avoid undone commands be kept between two *executed* commands. Still, the algorithm we propose for logical undo takes care of rearranging the undone commands so that they can be replayed instantly, thus keeping it cheap to “undo the undo”.

6.2 Script management with multiple windows

Script management also has strong interaction with the multiple window capability of CtCoq. If the user can open two windows and send commands from these two windows in an arbitrary order then there is no way to ensure that any of these windows will contain a proof script that can be stored in a file and replayed independently. To help with this issue, the user-interface keeps a hidden record of the order in which all commands have actually been sent to the proof engine. This record is then used to help the user move data from one window to the other, without disturbing the actual order.

For instance, let us suppose the user has two composite windows 1 and 2, where window 1 contains the commands A, C, E and window 2 contains the commands B, D, F, and that the commands have been sent in alphabetical order. Our tool makes it possible to move B from window 2 to window 1 and it enforces the constraint that B must be inserted between A and C.

Our tool could be more stringent, for instance by enforcing that saved files must contain contiguous segments of the master record. A quick reflexion on this constraint shows that it would be counter-productive: when users develop results based on facts from two domains, for instance polynomials and real numbers, they may want to have in one window their lemmas about polynoms and in the other window the lemmas about real numbers, independent of the order in which these lemmas were actually proved.

7 Conclusion

In this paper, we have toured the various features of the CtCoq user-interface. We have described the main issues in its implementation, concentrating on the following key aspects.

Multi-processing. The CtCoq user-interface is designed as a separate process that communicates with the proof engine using sockets and text-based protocols. We have described the kind of protocol used in both direction: tree-based protocol from the proof engine to the interface, text-based protocol from the user-interface to the proof engine. We have shown that the tree-based protocol has been made more robust by a using a specific methodology that relies on a formal description of the tree langage and automatic generation of protocol procedures. We have shown that the parsing capability had been implemented using yet another process, whose code had been extracted from the Coq proof assistant. We have shown how interrupts could be integrated in this framework, taking care that interruptions do not break communication protocols.

Data display. We have described the main tool for communication with the user: a composite window that groups together a window to record all commands sent by the user and windows to display the results of these commands. Because the data is stored internally as tree-like structures, the users are given the possibility to adapt the notation to their taste, or even to have several views on the same data, but with different layout strategies. We have shown how the layout strategy relies on a layout description language, PPML, that makes it possible to combine elementary specifications.

Mouse directed interaction. The CtCoq user-interface implements the proof-by-pointing algorithm described in [BKT94]. We have shown that the basic algorithm is made more useful by an optimization phase that simplifies the composite commands, which the algorithm generates. We have also shown that the tree-like structures in the user-interface made it easy to implement other strategies to interprete mouse gestures, providing an easily extensible drag-and-drop capability for algebraic manipulations.

Script management. The CtCoq user-interface also provides a variety of strategies to perform undoing commands. We have shown how these undo capabilities could be handled to support users in their task of producing coherent scripts that can be replayed in later sessions. Specific issues encompass *logical undo* that makes it possible to undo early commands without undoing commands that were executed later but are known to be unrelated, and the interaction between undoing and the possibility of interacting using several composite windows to interact with the same proof engine. We have described how one design guideline was to make it easy to "undo the undo", that is to cancel the effect of an undo command.

In all these aspects, the design of CtCoq was led mostly by the will to improve the support for large scale proof development and complete integration in the Coq user community.

Future research and development around the CtCoq user-interface follows three main directions. A first direction aims at changing the technological basis for the various interactive tools. Until now, we have been using the Centaur system that has imposed some limitations on our development: execution limited to unix platforms, restricted display capabilities, restricted memory management. We hope to re-implement the main capabilities of CtCoq on top of a toolkit that implements tree-based manipulation and full bi-dimensional layout with a portability extended outside the realm of unix platforms. A second direction aims at finding ways to integrate in one seamless capability the two strategies of mouse-directed interaction explored so far: proof-by-pointing and drag-and-drop rewrite. A third direction aims at providing more tools to help maintaining proofs once they are finished. This should include tools to visualize dependencies and use these dependencies to pilot consistency checking and recovery, as outlined in [PBR98].

Acknowledgements

Over the years, many researchers have contributed to this experience. I would especially like to thank Gilles Kahn and Laurent Théry, who first contributed many design ideas, then some development, and then became dedicated users. Janet Bertot, Francis Montagnac, Laurence Rideau, Healfdene Goguen, Yann Coscoy and Olivier Pons have also helped with the system development. The Coq team has also supported our efforts in making their system evolve to accommodate the needs of our interface.

Last but not least, the users must be thanked for their enduring support of this tool, especially Jean-François Monin from CNET and Emmanuel Ledinot from Dassault Aviation, both of whom helped with ideas and funding.

References

[BB96a] Janet Bertot and Yves Bertot. CtCoq: A system presentation. In *Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 231–234. Springer-Verlag, July 1996.

[BB96b] Janet Bertot and Yves Bertot. The ctcoq experience. In *Electronic proceedings for the conference UITP'96*, University of York, July 1996.

[BCD+88] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the system. In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).

[Ber97a] Yves Bertot. Direct manipulation of algebraic formulae in interactive proof systems. In *Electronic proceedings for the conference UITP'97*, Sophia Antipolis, September 1997.

[Ber97b] Yves Bertot. Head-tactics simplification. In *Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[Ber98] Yves Bertot. A certified compiler for an imperative language. Research Report RR-3488, INRIA, 1998.

[BF95] Yves Bertot and Ranan Fraer. Reasoning with Executable Specifications. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95)*, volume 915 of *Lecture Notes in Computer Science*, pages 531–545, 1995.

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160, 1994.

[BS94] Richard Bornat and Bernard Sufrin. Jape: a literal, lightweight, interactive proof assistant. Technical Report 641, Queen Mary and Westfield College, University of London, 1994.

[BS98] Richard Bornat and Bernard Sufrin. Using gestures to disambiguate unification. In *Informal proceedings of the Workshop on User Interfaces for Theorem Provers*, Eindhoven University of Technology, July 1998. Report 98-08.

[BT98] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25:161–194, 1998.

[BVW97] Roland C. Backhouse, Richard Verhoeven, and O. Weber. Mathspad: a system for on-line preparation of mathematical documents. *Software – Concepts and Tools*, 18:80–89, 1997.

[CAB+86] Robert Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harber, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[CH95] Ricardo Caferra and Michel Herment. A generic graphic framework for combining inference tools and editing proofs and formulae. *Journal of Symbolic Computation*, 19:217–243, 1995.

[Clé90] Dominique Clément. A distributed architecture for programming environments. *Software Engineering Notes*, 15(5), 1990. *Proceedings of the 4th Symposium on Software Development Environments*.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. *The Coq Proof Assistant User's Guide*. INRIA, May 1993. Version 5.8.

[DR94] Anne-Marie Déry and Laurence Rideau. Distributed programming environments: an example of message protocol. Rapport Technique 165, INRIA, 1994.

[Eas98] Katherine Eastaughffe. Support for interactive theorem proving: Some design principles and their application. In *Informal Proceedings of "User Interfaces for Theorem Provers 1998*, Eindhoven University of Technology, 1998.

[GRB93] Doug Goldson, Steve Reeves, and Richard Bornat. A review of several programs for the teaching of logic. *The Computer Journal*, 36(4):374–386, 1993.

[JMB+93] Ian Jacobs, Francis Montagnac, Janet Bertot, Dominique Clément, and Vincent Prunet. the Sophtalk reference manual. Rapport Technique 150, INRIA, 1993.

[LD97] Helen Lowe and D. Duncan. Xbarnacle: making theorem provers more accessible. In *Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 404–407. Springer-Verlag, July 1997.

[MH97] Nicholas A. Merriam and Michael D. Harrison. What is wrong with guis for theorem provers? In *Electronic proceedings for the conference UITP'97*, Sophia Antipolis, September 1997.

[MN94] Lena Magnusson and Bengt 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.

[OEC97] Maris A. Ozols, Katherine A. Eastaughffe, and Antony Cant. Xisabelle: A system description. In *Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 400–403. Springer-Verlag, July 1997.

[PBR98] Olivier Pons, Yves Bertot, and Laurence Rideau. Notions of dependency in proof assistants. In *User Interfaces for Theorem Provers 1998*, Eindhoven University of Technology, 1998.

[Pon97] Olivier Pons. Undoing and managing a proof. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1997"*, Sophia-Antipolis, France, 1997.

[Rit88] Brian Ritchie. *The design and implementation of an interactive proof editor*. PhD thesis, University of Edinburgh, 1988.

[RT97] Laurence Rideau and Laurent Théry. Interactive programming environment for ml. Rapport de Recherche RR-3139, INRIA, March 1997.

[SB98] Bernard Sufrin and Richard Bornat. User interfaces for generic proof assistants part ii: Displying proofs. In *Informal proceedings of the Workshop on User Interfaces for Theorem Provers*, Eindhoven University of Technology, July 1998. Report 98-08.

[Sup84] Patrick Suppes. The next generation of interactive theorem provers. In *Automated Deduction (CADE-7)*, volume 170 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1984.

[Sym95] Donald 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, September 1995.

[TBK92] Laurent Théry, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5), 1992. Proceedings of the 5th Symposium on Software Development Environments.

[Thé98] Laurent Théry. A certified version of Buchberger's algorithm. In *Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, July 1998.

[Thi90] Harold Thimbleby. *User Interface Design*. Frontier Series. ACM Press, 1990.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In Harald Ganzinger, editor, *European Symposium on Programming (ESOP'88)*, volume 300 of *LNCS*, pages 344–358. Springer Verlag, March 1988.

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