J. Symbolic Computation (1998) **25**, 161–194

A Generic Approach to Building User Interfaces for Theorem Provers

YVES BERTOT[†] AND LAURENT THÉRY

INRIA Sophia–Antipolis 2604 Route Des Lucioles, 06902 Sophia Antipolis Cedex, France

In this paper, we present the results of an ongoing effort in building user interfaces for proof systems. Our approach is generic: we are not constructing a user interface for a particular proof system, rather we have developed techniques and tools that have been applied to several proof systems. We first propose and motivate a distributed architecture, where the proof system and the interface are two separate processes communicating through a protocol. Then we describe three high-level features: *proof-by-pointing*, *script management*, and *textual explanation*. Altogether, they take advantage of the underlying architecture and yield a more user-friendly proof environment.

1. Introduction

It is our belief that proof systems will become an important component of future software and hardware system developments. So far, theorem proving has been limited to experts, who are ready to make the effort to learn the behavior of proof systems and their arcane notations. Integrating such systems in user-friendly environments is crucial if run-of-the-mill software engineers are to use these tools on a daily basis.

Theorem proving is a highly interactive activity for which building a complete interface is not a trivial task. The dialog between the user and the system deals with high level objects such as tactics and proofs. It should benefit from all the modern user interface technologies (windowing, mouse, pen, voice).

Our point is not to build a new proof system. Rather, software reuse should be optimized and our user interface work is designed to adapt from one proof system to another. Software reuse can be achieved in one of the following ways:

- by linking together two components; one that is specialized in proof management and one that is specialized in user interface,
- by having two programs dialogue using a protocol where one program manages the proof and the other manages the user interface.

The first solution results in a big monolithic program. If the whole program is written

E-mail:bertot,thery@sophia.inria.fr

0747–7171/98/020161 + 34 $25.00/0 sy970171

in a general purpose programming language, then the language may not be suited for one of the two components. For example in Jones et al. (1991), the authors justify the use of the language SmallTalk for implementing the system Mural only in terms of user interface capabilities. Alternatively, different programming languages are linked together and, in this case, the resulting system is difficult to port and maintain.

The results we are going to present in this paper will instead be based on communicating programs. This technique enables us to reuse existing software, as can be shown by the number of proof systems to which we have been able to adapt user interfaces (some of these interfaces are no longer maintained): Amy Felty's theorem prover (Felty (1989)), HOL (Gordon & Melham (1993)), Isabelle (Paulson & Nipkow (1994)), LEGO (Luo & Pollack (1992)), and Coq (Dowek et al. (1993)).

We take advantage of the similarity between constructing proofs and constructing programs. We have used the Centaur system (Borras et al. (1988)), a programming environment generator, for designing the user interface. The basic function of this system is interactive structure manipulation. The main data structures are *trees*. The system provides ways to specify languages as sets of trees, methods for constructing such trees (parsing), methods for displaying them (formatting), and methods for their semantic manipulation. On the interactive side, the Centaur system provides graphical windows where the user can manipulate trees, printed as formulas or expressions, by pointing at them with the mouse and applying editing commands on the selected expressions.

Aside from using separate processes for proof manipulation and the user interface, the main idea is that structure manipulation is a *must* in user interfaces. In the remainder of this paper, we first describe the basic components of user interfaces: communication protocols, graphical displaying and formatting tools, and structure editing capabilities. Then, we describe advanced concepts for user-friendly interfaces. These advanced concepts address three common issues for users of interactive proof assistants. The first concept, *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 with a simple push-button user interface. The second concept, *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. The third concept, *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 specific proof system used to mechanically verify the proofs.

1.1. An Example

To give a feel for the functionalities that can be provided in the user interfaces we envision, we are going to follow a sample session with the CtCoq system, which is built on top of the Coq system (Dowek et al. (1993)). This proof system enables one to state definitions and to perform goal directed proofs where one states an initial goal that can be transformed into simpler subgoals using commands called *tactics*. This proof system is based on type theory and the proofs themselves are represented by typed λ-terms.

When the system is started, a window appears on the screen. This window is composed of five areas that can be seen in figure 1:

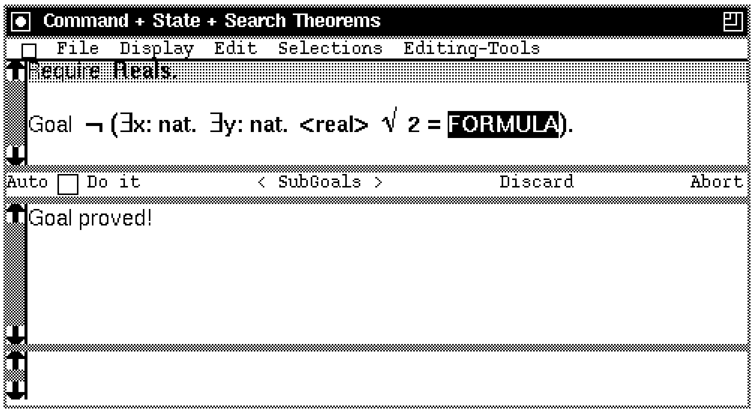

Figure 1. The CtCoq main window

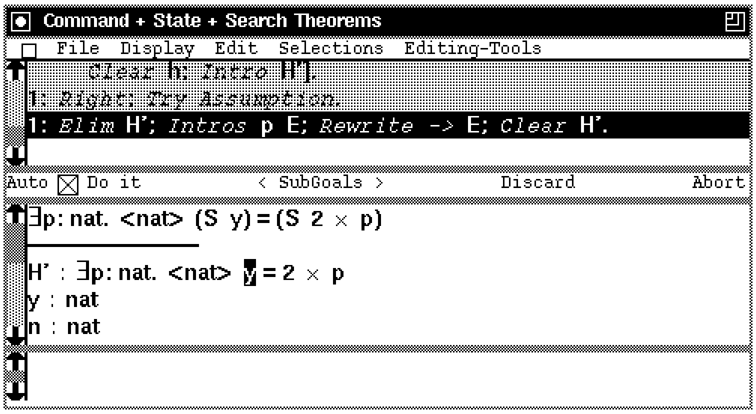

Figure 2. Selecting an expression for *proof-by-pointing*

1. the topmost area contains a menubar with access to a variety of editing and visualizing commands,
2. the next area is a *command* area, where the user can edit and visualize commands,
3. the next area is a row of tools that control the interaction with the proof system,
4. the next area is a *state* area, where the formulae that remain to be proved are displayed during goal directed proofs,
5. the lowest area is a *context* area, where the results of searches in the proof system's database are displayed.

Figure 1 gives an example of the content of this window when the user is in the middle of entering a formula to prove. The command `Require Reals` has already been sent to the proof system, and this is made visible to the user by a darker background. The second command, which starts with `Goal`, is currently being edited. The editing is structured: whenever the user clicks somewhere, it is not a single character that is selected, but an entire expression. Currently, the user selected a *place-holder*: `FORMULA`. This represents a hole in the formula. There are two ways to edit a hole. In *text-editing*

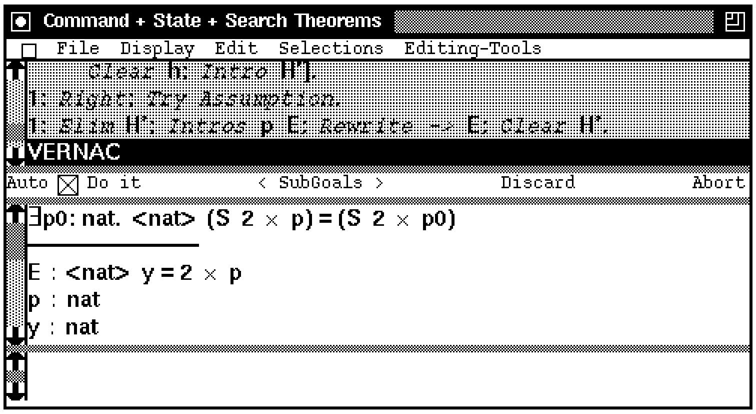

Figure 3. Result of the proof-by-pointing action of figure 2

mode the formula can be input character by character. In *structured-editing* mode one can use tree manipulating commands that are available in the `Edit` and `Editing-Tools` menus provided in the menu bar. When an expression is edited as text, it is presented with only plain ascii characters. However, when an expression is manipulated as a structure, the display machinery can pretty-print it using a rich character set. For instance, symbols for negation, $\neg$, existential quantification, $\exists$, and square root, $\surd$, appear in our example.

After sending the completed `Goal` command to the proof system, the user enters a *goal directed proof* mode, where the proof system maintains a list of *current* goals, on which the user can act with a set of predefined commands. Obviously, these commands can be entered by structure and text editing exactly like the first command. But the interaction can be more effective using the functionality we call *proof-by-pointing*, where the user selects important expressions in the goal formula to guide the proof process. For example, figure 2 describes the case where one wants to use the assumption that there exists a p such that $y = 2 \times p$ (this assumption is named H' in our example where assumptions are listed under a horizontal line) to replace the occurrences of y by the corresponding $2 \times p$ in the conclusion of the same goal (conclusions of goals appear above the horizontal line). This operation is simply requested by clicking with the mouse on the y that appears in reverse video in the *State* area in figure 2. The result of this operation is shown in figure 3: a new constant p has been added, along with an assumption named E that states that $y = 2 \times p$ and rewriting is performed in the goal formula. Notice that the bound variable "p" in the conclusion has been renamed $p0$.

When a proof is complete, the Coq system builds a typed λ-term that represents the complete derivation of the statement. This λ-term represents all the basic inferences used in the proof and it is quite hard to read, mainly because these inferences are encoded in the constructs of λ-calculus. However, the structure-oriented interface makes it possible to annotate the λ-term and recover the intuitive meaning of the term. Figure 4 gives an example of such a decoration. In this figure, the two windows show different presentations of the same proof object. The topmost window uses a textual form, where reasoning constructs are described by sentences in "pseudo natural language" form. The bottom window uses a mathematical form which is closer to the usual λ-calculus. In both windows, the same sub-expression is selected and displayed with a darker background.

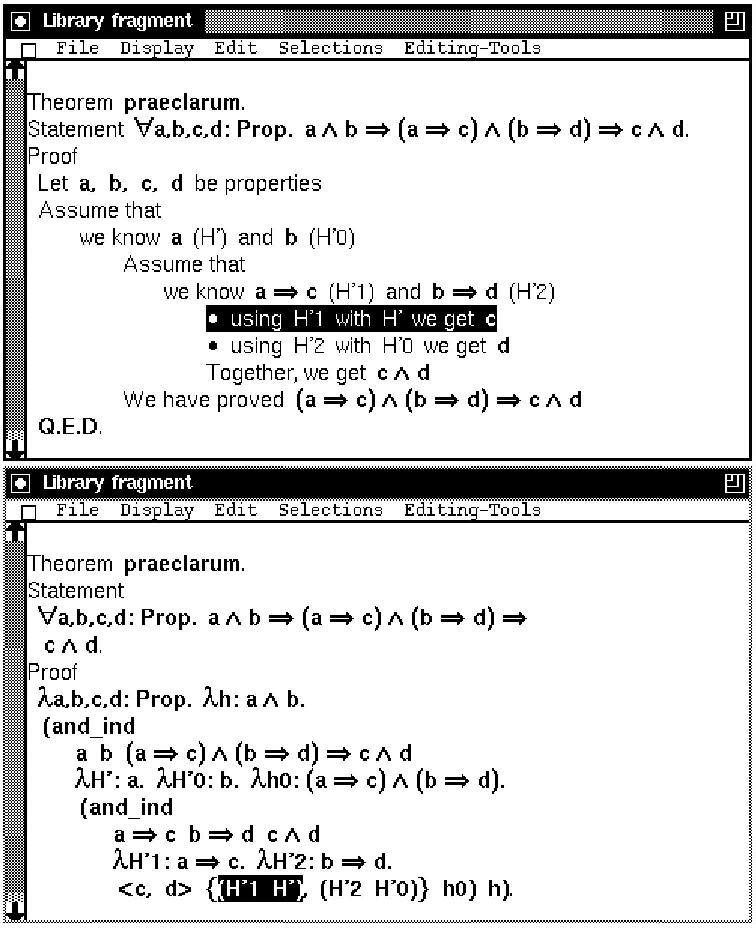

Figure 4. Textual presentation of a proof

1.2. Related Work

This paper sums up the work realized by the authors and members of their team over several years. In particular, Théry et al. (1992) describe the basic principles of this research: we do not develop a new proof system but we provide a separate user interface, built with tools provided in programming environments. This approach is significantly different from the approaches found in many other implementations of graphical user interfaces for proof systems where the user interface and the logical engine are integrated together. Examples of such systems are Constable et al. (1986), Ritchie (1988), Griffin (1988) Jones et al. (1991) and Caferra & Herment (1995). In this respect, the closest experiments to ours may be the logical framework ALF described in Magnusson & Nordström (1994), the lightweight proof assistant Jape described in Bornat & Sufrin (1994), and the graphical interface TkHOL for the proof system HOL described in Syme (1995), where the proof engine and the user interface are separate processes.

Other theorem provers only provide crude user interfaces, rarely going beyond extensible pretty-printing and parsing. Users are then left on their own to find or implement some support. A significant advance in this respect is provided by the extensible and programmable text editor `Emacs` (Cameron & Rosenblatt (1991)), with its `shell-buffer` functionality, that makes it possible to record all traces of interaction with the proof system in a file. Also, shortcuts for frequent interaction patterns can be implemented as macros or menu options. For example, the PVS system (Shankar et al. (1993)) uses the Emacs approach for its user interface. However, this approach lacks support for manipulating the inherent structure of logical formulae and commands, which is pervasive in our work. Still, our approach is close in spirit to the `Emacs` approach in the sense that we attempt to be generic and develop tools that will apply to many proof systems.

2. Architecture

In this section we describe the main components that must be present to provide powerful user interfaces. We concentrate on communication protocols, graphical toolkits, layout mechanisms, and structure manipulation. While these components were developed independently from the issue of implementing user interfaces for proof systems, we highlight those features of these components that are essential in our experiments.

Our method for building user interfaces follows the ideas presented in Clément (1990) and Clément et al. (1991). An interface is a *network* of cooperating components that communicate by *broadcasting* messages. This model promotes modularity in the design of the user interface. When adding a new interactive object, all that matters is the messages that the object can emit or receive. In this model, the proof engine is just one of the components of the network even though it is an important provider and receiver of messages.

We use SOPHTALK (Jacobs et al. (1993)) to implement the network descriptions and the protocols devised in Déry & Rideau (1994) to give us a simple mechanism to encapsulate any process as an object of the network. A central aspect of the protocols is to make it possible to transmit structured data. This is necessary if we want the user interface to manipulate objects such as formulae, sequents, or proof trees. Structured objects are sent as typed trees, that is, trees belonging to a particular abstract syntax.

The link between the prover and its interface uses Transmission Control Protocol (TCP) sockets. From the prover side, establishing this link is made easier by simply encapsulating this process in a service provider that just takes the process' standard input and output and provides TCP sockets to all the clients. Then in all our experiments, we had only to modify slightly the toplevel loop of the prover so that the output follows a simple ascii protocol. In particular, data output is performed by a postfix traversal of the data structure that we implemented.

Protocols of this kind are also used for computer algebra systems (Kajler (1992)) to communicate data between a computer algebra system and a plotter, a code generator, or between several computer algebra systems, in order to use best the capabilities provided by the various systems. In this area, authors like Gray et al. (1994) have proposed optimized protocols to exchange data. In order to reduce the size of the messages, one of the main optimizations is to express the sharing of subexpressions in the protocols. In our application, it appears that the largest data that we need to transmit is mainly the data that is built interactively. So the size of messages can be reduced by sending *incremental* updates.

To achieve incrementality, it is necessary to compute the difference between formulae before and after each proof step. Ideally, the proof engine could compute this difference simultaneously with the new data and communicate only the difference. This kind of communication using changes exists in the ALF system. Most of the time, however, proof systems are not so cooperative and difference information has to be recovered *a posteriori* using a tree differentiation algorithm. The complexity of general differentiation algorithms is at least quadratic in the size of the compared data. However, most commands act on the proof system's state in a very systematic way: these commands work on goal directed proofs and it is usually trivial to detect that these commands act on only one goal, although the state of the proof system may be represented by a large list of goals. The proof system may also give information that can be used as a summary to avoid comparing data that is predictably different. For example, in Coq (Dowek et al. (1993)), Lego (Luo & Pollack (1992)), and PVS (Shankar et al. (1993)) assumptions are given names which do not change between successive states, so that it is usually relevant to drop the comparison between two assumptions that have a different name. In our experiments, these properties have been sufficient to reduce the complexity of the differentiation algorithm making incremental updates worth using.

2.1. Multiple Aspects for Multiple Objects

Proof systems are manipulating different kinds of objects. Typically, a prover will have objects that represent knowledge (the set of known definitions and theorems), proofs, and commands.

Because these objects are inherently different, it is natural to reflect the diversity of objects in the interface by means of dedicated windows. For example, the knowledge of the system is organized in theories that can be viewed as a directed graph. Also commands are usually constructed and kept in a persistent script, so textual editors are needed.

For a single object, there may also exist several possible representations, that may or may not co-exist simultaneously in the interface. For example, in our experiment different presentations of proof objects have been proposed: as a λterm to be inserted in the command language, as a two dimensional proof-tree, or as a natural language text.

Design guidelines have to be followed to avoid the anarchy that can result from such composite interfaces. One needs to *structure* the interface by grouping objects related to the same activity, and to keep some *uniform* representation for the same object in different views. For example, we can describe the construction panel in Figure 5 presented in the section 1.1. The panel is composed of five different components, listed from top to bottom:

1. a menubar,
2. a script editor,
3. a row of control buttons,
4. a subgoal view,
5. a view that displays result to theorem database queries.

The composition of this panel follows directly from the activity of this window which is to build proofs. First of all, the script is a linear structure where every new command is added at the end of the script. So user attention will be concentrated around the bottom part of the editor. It is then natural to put buttons, such as `Quit` or `Abort` below the script rather than in the menubar or on the side of the window. Similarly, the goal

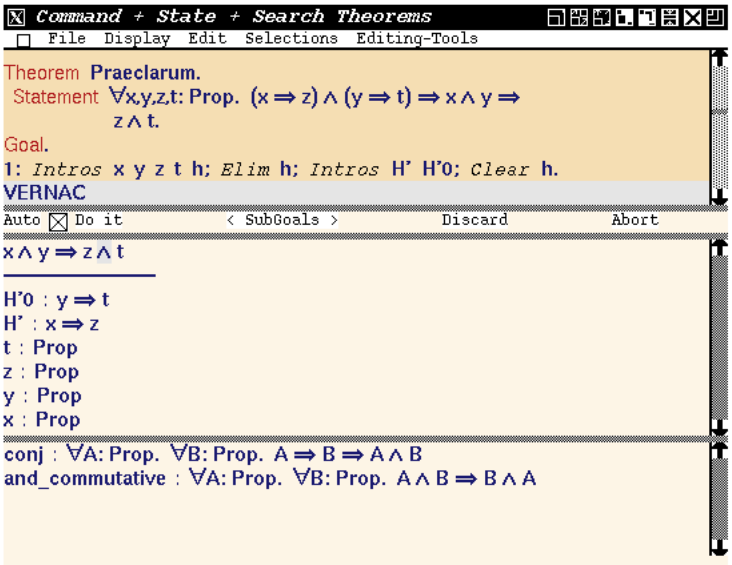

Figure 5. A composite window

window that can be used for constructing new commands in the script is added after the buttons.

In the same spirit, subgoals are presented with the conclusion that has to be proved first and below the list of local assumptions. As theorems in Coq can be used on a subgoal like any local assumptions, it is also natural to put the view containing theorems below the assumptions and to present theorems *like* assumptions.

Uniformity must also appear in the use of the mouse and keyboard inputs. For instance, the behavior of the mouse buttons is the same throughout the various components of this composite window.

3. Display and Manipulation

3.1. Display Mechanism

Displaying objects in the context of a theorem prover is a challenging task. Proofs manipulate mathematical objects that have well established and demanding typographic conventions. The display mechanism also needs to be *customizable*, so that users can adapt fonts and colors to their taste, and *extensible*, so that users who create new objects can associate their own graphical representation. Also, some operations such as navigating inside large proof trees or designating a subgoal or the portion of the goal, on which a command has to be applied, are naturally expressed by pointing with the finger, the pen, or the mouse on a particular region of the displayed objects. Providing such capabilities requires not only displaying objects but also keeping links between the graphical representation and the structure of the objects. Finally, proofs and theories are objects that are progressively refined in the interaction process between the prover and the user. From our experiments, the naive approach of recomputing the layout after each step leads to an interface that becomes slower and slower as the proof progresses. So, some *incrementality* has to be integrated to the display mechanism in order to avoid recomputing the layout of unchanged parts of the modified object.

Research in the generation of programming environments (Borras et al. (1988), Reps & Teitelbaum (1988)) has proposed solutions to similar problems. One of the contributions of our work has been to show that these solutions are directly applicable to the generation of interfaces for proof systems. In our approach, we follow Borras et al. (1988) and use the formalism called PPML to express the layout of structured data. A PPML description defines a mapping between data structures and their layout structures. It is composed of a set of rules. Each rule associates a pattern that filters the data with a layout description. For instance, the following rule describes the layout of a binary operator. The text between brackets < and > describes the line-breaking strategy and the class mechanism is used to describe a change of font or color.

```
binop(op "plus", *x, *y) ⟶ [<hov 1,1,0> *x in class = operator :"+" *y];
```

A rule is triggered if it is the first in the list to match the current data. The layout description is given in a box style. The benefits of this approach are immediate:

- The engine that interprets PPML descriptions is a reusable component; so a lot of effort has been put into making it robust and efficient.
- Customization is provided by hooking attributes on the boxes of the PPML description. Attribute names are then used to retrieve specific graphical resources such as font or background and foreground colors from a resource file.
- Extensibility is provided through easy ways to compose several pretty-printing specifications, to modify a sub-specification, and to recompile it. In practice, when users want to add a new notation they have to edit a few lines of code and they can interactively test their modifications.
- Selections with the mouse are automatically translated from coordinates in the window to a subterm of the data structure.
- Incrementality is free. Any modification on the data is automatically propagated in terms of updates in the box structure so that the graphical representation can be incrementally updated.

3.2. Manipulation

In this section, we describe tools that have been implemented around the notion of abstract syntax. These tools encompass various ways of manipulating data: top-down construction of commands by templates, menu-directed interactive transformations based on rewriting, textual editing, and we show how these different points of view can collaborate efficiently. At the end of the section, we discuss design decisions when designing an abstract syntax to make these tools easy to implement and to use, and we present the interaction of abstract syntax and type discipline.

The user interface process is based on editing windows where manipulations are performed at the tree or structured data level. An abstract syntax describes how to construct structured data for commands and logical formulae. It consists of *sorts* and *operators*. Operators represent primitive tree patterns, while sorts represent tree categories, like commands or logical formulae. Each sort is defined as the set of head operators accepted for trees in this sort. Each operator is defined by its name and an arity, a function that maps the rank of the children to the sort allowed for each child. An extra operator that belongs to all the sorts is automatically added to represent holes in incomplete trees.

Editing windows restrict the user operations to only construct syntactically correct trees.For any position in the tree, computing syntactic constraints is done by looking at the operator above that position, and looking up the sort associated to that position's rank in the operator's arity.

Usual operations such as copying a tree, inserting an element in a list, and replacement are provided by the Centaur system. All editing operations are performed with respect to the *currently selected expression*. We also experimented with a less commonplace operation called *auto-clip* that performs tree substitution using two mouse buttons. One button enables users to move the current selection around the window and the other enables them to choose a value to replace the current selection. When connected with an automatic jump to the next hole to fill in, this capability has proven to be a very efficient tool for quickly constructing a command by using parts from previous commands in the script. Notice that this capability is safer than the standard X copy-and-paste because most of the wrong manipulations are automatically ruled out by the requirement that the two expressions that are selected have to be of the same sort. This kind of control leads to quicker interaction as syntactic errors are detected earlier.

Every time the current selection is moved, it also updates the *current sort* for that position and various tools can use this information. With the current sort mechanism, Centaur provides generic menus that propose tree construction and transformation patterns to the user. The user can select a pattern either with the mouse or with a keyboard sequence. Designing the set of available patterns has been an important part of our design activity. These menus can be tuned to specific classes of users. To define new patterns, we use a language of rewrite rules. For instance, the following two rules describe basic transformations.

```
'Intro' : {TACTIC}
    *x --> intro(*id) ;

'and1' : {FORMULA}
    *x --> and(*x,*y) ;
```

Identifiers starting with a star denote variable components. The first rule performs simple template directed editing while the second rule performs a transformation that adds an `and` construct around the selected formula. These rules are available to the user only when pertinent, that is, when the currently selected expression belongs in the TACTIC or FORMULA sorts.

Beginner users will need menus that describe the complete set of commands available in the system. Expert users will prefer complex menus adapted to specific mathematical domains, like group theory or program verification, or to specific modes of activity, like proof search or theory re-engineering. To make this variety of menus possible, it is important to have a formalism to describe menus and to have the possibility to have several menus at a time on the screen (one for general use and one for specific tasks). This multiplicity of context help is possible thanks to the use of the broadcast approach of the SOPHTALK system (Jacobs et al. (1993)) that underlies our architecture.

When defining the formal languages to represent logical formulae and commands, there is a tradeoff between the power of this system of menus and the simplicity of the formal languages. Formal languages can be made very simple by *externalizing* the operators: every construct of the language is represented by the application of a function to a list of arguments. With this kind of encoding, the language is very simple and easily extensible. But it also contains very few sorts and therefore menus will be able to provide very little context help. On the other hand, the formal language can be made very sophisticated by using different operators for all the constructs of the language. In this case, the language becomes more rigid and harder to maintain, but the assistance given to users through the sort driven menus is more efficient, since the language provides a wider variety of sorts. Note that a way to recapture pertinent menus with extensible grammar is to use type information. For example Magnusson & Nordström (1994) describe the implementation of type-driven menus. While we have developed similar menus for specific user interfaces like the one for the logical framework ISABELLE, more experiments are needed in order to understand to what extent this capability could be provided in a generic way.

Menu driven editing should only be one of several ways to enter data, and users should have the possibility of simply typing in text. More generally, users should be able to select any subexpression, modify its textual representation, parse the new version, and replace the subexpression by the new value, provided it still makes sense from a syntactical point of view. In our experiments, we have included this functionality with little effort, the only component to provide being the parser for subexpressions. As a side effect, we also inherit the communication with other applications through the X copy-and-paste mechanism. Users are then free to mix structured editing with character based editing. This is a significant improvement compared to related works where either only structured editing is available (such as in the Nuprl interface) or only textual editing is available (the most common situation).

4. Proof-by-Pointing

In goal-directed proof engines, the user can attack a new theorem by giving its statement as a goal and applying commands to break down the goal into simpler subgoals, until all the subgoals are solved. In the function we call *proof-by-pointing* (see Bertot et al. (1994)), the goals output by the proof engine are used to generate new commands to reduce these goals. While the menus presented in section 3.2 use a very limited notion of context, restricted to the sort of the current selection in the main window, the guidance provided by proof-by-pointing is more sensitive to the actual state of the proof system. Proof-by-pointing is only one way to interact with the proof engine, and one should keep in mind that users may still use other kinds of commands, possibly very powerful ones like decision procedures. Proof-by-pointing is only there to ensure that the basic inference steps that users must do to prepare the ground for a decision procedure can easily be made.

The basic idea of proof-by-pointing is that selecting a position in a goal formula can be interpreted as a command to bring the selected subformula to the surface of the goal. To describe this algorithm we present it as an operational interpretation of Gentzen's rules for natural deduction, based on an annotated version of these rules. In this section, we give an example of the benefits that can be obtained from this functionality and we give a short presentation of the proof-by-pointing algorithm. Then we discuss two important issues: controlling the number of assumptions and extending the algorithm to new logical connectives.

4.1. An example

Consider the following first order logic goal formula, G_0 , where a and b are individuals and p and q are predicate symbols:

$$(G_0) \qquad (p(a) \vee q(b)) \wedge (\forall x\ p(x) \supset q(x)) \supset (\exists x\ q(x))$$

Formula G_0 can be paraphrased in English as: *if we know that either p is verified for a or q is verified for b, and that for all x, if p is verified for x then q is verified for the same x, then there exists an x for which property q is verified.*

The proof of this fact examines the two cases involved in the formula $p(a) \vee q(b)$. In the case where $p(a)$ holds, we use the fact $\forall x\ p(x) \supset q(x)$ to deduce $q(a)$. Then a is a witness to prove $\exists x\ q(x)$ in that case. In the second case $q(b)$ holds, so the witness b is directly available.

To steer the computer toward the proof, the user points to subformula $p(a)$ with the mouse. As it occurs within expression $p(a) \vee q(b)$, this indicates interest in a case analysis. The proof state changes to include two new subgoals G_1 and G_2 :

$$(G_1) \qquad p(a),\ p(a) \vee q(b),\ (p(a) \vee q(b)) \wedge (\forall x\ p(x) \supset q(x)) \vdash \exists x\ q(x)$$

$$(G_2) \qquad q(b),\ p(a) \vee q(b),\ (p(a) \vee q(b)) \wedge (\forall x\ p(x) \supset q(x)) \vdash \exists x\ q(x)$$

In our notation, the turnstile symbol $\vdash$ separates the local assumptions from the conclusion in a subgoal, and assumptions are separated by commas. Naturally, assumptions that are local to a subgoal can only be used to prove this subgoal's conclusion.

The user is free to carry on working with subgoal G_1 or G_2 , although G_1 should be emphasized since $p(a)$ rather than $q(b)$ was pointed at initially. In G_1 , since $p(a)$ and $\forall x\ p(x) \supset q(x)$ hold one can deduce $q(a)$. This inference step is requested by pointing at subexpression $p(x)$ in G_1, meaning, *prove an instance of $p(x)$ and deduce the corresponding instance of $q(x)$*. In the proof state, subgoal G_1 is replaced by G_3 :

$$(G_3) \qquad q(a),\ p(a),\ p(a) \vee q(b),\ (p(a) \vee q(b)) \wedge (\forall x\ p(x) \supset q(x)) \vdash \exists x\ q(x)$$

Now subgoal G_3 can easily be dealt with. The fact $q(a)$ appears in the assumptions and we need to prove $\exists x\ q(x)$. The user simply selects $q(x)$ behind the existential quantifier in G_3 with the intended meaning *there is a witness for x in the assumptions of this goal that allows one to prove $q(x)$*. Subgoal G_3 vanishes and only G_2 remains. Subgoal G_2 is handled in an identical fashion and vanishes as well. As no subgoals remain to be proved, the result is established.

Throughout this example, the meaning of mouse designation is not *ad hoc*. In goal G_0, the first mouse click designates the expression $p(a)$. The location of this expression can be described in three steps, beginning in G_0 :

i) to the left of an implication symbol (denoted by $\supset$),
ii) to the left of a conjunction symbol (denoted by $\wedge$),
iii) to the left of a disjunction symbol (denoted by $\vee$).

When pointing at $p(a)$, each one of these facts is exploited in turn:

i) the antecedent of the implication is added as an assumption,
ii) the left part of the conjunction is extracted and added as an assumption,

iii) two subgoals corresponding to the two cases in the disjunction are created, with either disjunct as additional assumption; the goal created by the left disjunct is emphasized.

The second mouse click is simpler to explain. In goal G_1 , the second mouse click points at expression $p(x)$. This expression occurs in an assumption and:

i) to the right of a conjunction symbol,
ii) within a universally quantified expression,
iii) to the left of an implication symbol.

As a consequence, pointing at $p(x)$ directs the computer to:

i) extract the right conjunct,
ii) find a proof of $p(x)$ for some x,
iii) add a new assumption $q(x)$ for the same x, creating G_3.

The last two mouse clicks are even simpler. In goals G_3 , and similarly in goal G_2, the user points at $q(x)$ in the conclusion of the goal, within the existentially quantified formula. In both cases, the system looks through the assumptions to see if an instance of $q(x)$ is directly provable. In both cases it is successful, so the goals are eliminated.

This finishes our informal presentation of the proof-by-pointing algorithm, which can be summarized by the rules given in figure 6. These rules follow Gentzen's presentation of logical deduction (Szabo (1969)). Each rule has two parts separated by an horizontal bar: a list of sequents, the *premises*, and a single sequent, the *conclusion*. The intuitive meaning is that a goal that matches the conclusion of the rule can be proved if the corresponding premises can be proved. Thus, rules can be used to reduce goals to simpler goals. Any goal can be *closed* (i.e., considered solved) if its conclusion (the formula at the right of the $\vdash$ symbol) occurs in the list of assumptions (the formulae at the left). To describe proof-by-pointing, these rules have been annotated with boxes that indicate the part of the goal that contains the currently selected expression. When this expression is a complete assumption or the complete conclusion of a goal, then the algorithm constructs a command that attempts to close the goal using the selected assumption when one is selected or any assumption that matches when the conclusion is selected. When the currently selected expression is a proper subexpression of an assumption or the conclusion, the algorithms applies the appropriate rule from figure 6. New goals corresponding to the premises of the rule will be created and there will be a *residual* of the currently selected expression in one of these goals. The algorithm repeats, recursively, its operation with this new goal and this new currently selected expression. The termination of the algorithm is ensured by the fact that the depth of the currently selected expression decreases at each recursive call of the algorithm.

From a practical point of view, the context also contains all the theorems that have already been proved, and proof-by-pointing can also be used on the statements of these theorems, as if they were plain assumptions. Note that this is reflected in the organization of windows in figure 1 where the window containing the results of lookups in the proof system theorem database is placed next to the goal's list of assumption.

This presentation of the proof-by-pointing algorithm as an annotation of Gentzen's

$$\wedge\, \mathit{left}_1 : \frac{\boxed{A}, B, A \wedge B, \Gamma \vdash C}{\boxed{A} \wedge B, \Gamma \vdash C} \qquad \wedge\, \mathit{right}_1 : \frac{\Gamma \vdash \boxed{A} \qquad \Gamma \vdash B}{\Gamma \vdash \boxed{A} \wedge B}$$

$$\wedge\, \mathit{left}_2 : \frac{A, \boxed{B}, A \wedge B, \Gamma \vdash C}{A \wedge \boxed{B}, \Gamma \vdash C} \qquad \wedge\, \mathit{right}_2 : \frac{\Gamma \vdash A \qquad \Gamma \vdash \boxed{B}}{\Gamma \vdash A \wedge \boxed{B}}$$

$$\vee\, \mathit{left}_1 : \frac{\boxed{A}, A \vee B, \Gamma \vdash C \qquad B, A \vee B, \Gamma \vdash C}{\boxed{A} \vee B, \Gamma \vdash C} \qquad \vee\, \mathit{right}_1 : \frac{\Gamma \vdash \boxed{A}}{\Gamma \vdash \boxed{A} \vee B}$$

$$\vee\, \mathit{left}_2 : \frac{A, A \vee B, \Gamma \vdash C \qquad \boxed{B}, A \vee B, \Gamma \vdash C}{A \vee \boxed{B}, \Gamma \vdash C} \qquad \vee\, \mathit{right}_2 : \frac{\Gamma \vdash \boxed{B}}{\Gamma \vdash A \vee \boxed{B}}$$

$$\supset \mathit{left}_1 : \frac{A \supset B, \Gamma \vdash \boxed{A} \qquad B, A \supset B, \Gamma \vdash C}{\boxed{A} \supset B, \Gamma \vdash C} \qquad \supset \mathit{right}_1 : \frac{\boxed{A}, \Gamma \vdash B}{\Gamma \vdash \boxed{A} \supset B}$$

$$\supset \mathit{left}_2 : \frac{A \supset B, \Gamma \vdash A \qquad \boxed{B}, A \supset B, \Gamma \vdash C}{A \supset \boxed{B}, \Gamma \vdash C} \qquad \supset \mathit{right}_2 : \frac{A, \Gamma \vdash \boxed{B}}{\Gamma \vdash A \supset \boxed{B}}$$

$$\forall\, \mathit{left} : \frac{\boxed{A[x\backslash e]}, \forall x\, A, \Gamma \vdash C}{\forall x \boxed{A}, \Gamma \vdash C} \qquad \forall\, \mathit{right} : \frac{\Gamma \vdash \boxed{A[x\backslash c]}}{\Gamma \vdash \forall x \boxed{A}}$$

$$\exists\, \mathit{left} : \frac{\boxed{A[x\backslash c]}, \exists x\, A, \Gamma \vdash C}{\exists x \boxed{A}, \Gamma \vdash C} \qquad \exists\, \mathit{right} : \frac{\Gamma \vdash \boxed{A[x\backslash e]}}{\Gamma \vdash \exists x \boxed{A}}$$

Figure 6. Rules of proof-by-pointing

rules for LJ, the rules for intuitionistic logic, seems to imply that the algorithm is restricted to this kind of logic. Extension to classical logic is simply done by considering the principle of the excluded middle as a theorem, whose statement is $\forall P.P \vee \neg P$. This theorem can be used at any time, like any other theorem.

What appears directly from this example is that any efficient use of this algorithm relies on the possibility of displaying logical formulae in windows that are sensitive to mouse clicks and to make it possible to know the selected expressions. These windows are used like menus, except that they do not provide a fixed number of options. In fact, almost every single node in the tree representation of the goals corresponds to a different option in the menu.

4.2. Controlling the growth of local contexts

The algorithm given in the previous section has the flaw of producing a large number of assumptions at each proof step. This comes from the fact that no rule actually removes assumptions from the context. In Bertot et al. (1994), we propose a "linear" form of the algorithm that destroys assumptions that are used during the development of the algorithm. This linear algorithm is an overkill: some proofs are no longer possible because assumptions that would have been used twice are destroyed at the first occasion. The conclusion in Bertot et al. (1994) is that this linear algorithm should be used only as a

$$\wedge\, \mathit{left}_1 : \frac{\boxed{A}, B, \Gamma \vdash C}{\boxed{A} \wedge B, \Gamma \vdash C} \qquad \wedge\, \mathit{right}_1 : \frac{\Gamma \vdash \boxed{A} \qquad \Gamma \vdash B}{\Gamma \vdash \boxed{A} \wedge B}$$

$$\wedge\, \mathit{left}_2 : \frac{A, \boxed{B}, \Gamma \vdash C}{A \wedge \boxed{B}, \Gamma \vdash C} \qquad \wedge\, \mathit{right}_2 : \frac{\Gamma \vdash A \qquad \Gamma \vdash \boxed{B}}{\Gamma \vdash A \wedge \boxed{B}}$$

$$\vee\, \mathit{left}_1 : \frac{\boxed{A}, \Gamma \vdash C \qquad B, \Gamma \vdash C}{\boxed{A} \vee B, \Gamma \vdash C} \qquad \vee\, \mathit{right}_1 : \frac{\Gamma \vdash \boxed{A}}{\Gamma \vdash \boxed{A} \vee B}$$

$$\vee\, \mathit{left}_2 : \frac{A, \Gamma \vdash C \qquad \boxed{B}, \Gamma \vdash C}{A \vee \boxed{B}, \Gamma \vdash C} \qquad \vee\, \mathit{right}_2 : \frac{\Gamma \vdash \boxed{B}}{\Gamma \vdash A \vee \boxed{B}}$$

$$\supset \mathit{left}_1 : \frac{A \supset B, \Gamma \vdash \boxed{A} \qquad B, \Gamma \vdash C}{\boxed{A} \supset B, \Gamma \vdash C} \qquad \supset \mathit{right}_1 : \frac{\boxed{A}, \Gamma \vdash B}{\Gamma \vdash \boxed{A} \supset B}$$

$$\supset \mathit{left}_2 : \frac{A \supset B, \Gamma \vdash A \qquad \boxed{B}, \Gamma \vdash C}{A \supset \boxed{B}, \Gamma \vdash C} \qquad \supset \mathit{right}_2 : \frac{A, \Gamma \vdash \boxed{B}}{\Gamma \vdash A \supset \boxed{B}}$$

$$\forall\, \mathit{left} : \frac{\boxed{A[x\backslash e]}^{\star}, \forall x\, A, \Gamma \vdash C}{\forall x \boxed{A}, \Gamma \vdash C} \qquad \forall\, \mathit{right} : \frac{\Gamma \vdash \boxed{A[x\backslash c]}}{\Gamma \vdash \forall x \boxed{A}}$$

$$\exists\, \mathit{left} : \frac{\boxed{A[x\backslash c]}, \Gamma \vdash C}{\exists x \boxed{A}, \Gamma \vdash C} \qquad \exists\, \mathit{right} : \frac{\Gamma \vdash \boxed{A[x\backslash e]}}{\Gamma \vdash \exists x \boxed{A}}$$

Figure 7. Rules with consumption marks

complement to the regular algorithm, leaving the user with the difficult task of choosing the right behavior at the right moment.

A simple improvement is to destroy any assumptions that are not absolutely needed in further reasoning. In some rules, the assumption that is used should always be destroyed, because newly created assumptions contain more information. In other rules newly created assumptions should be marked as consumable, also because some other assumption contains more information.

To represent formally this new variant of the algorithm, we can take the rules of figure 6 and modify them either by removing unnecessary assumptions, or by adding an annotation indicating when assumptions will be amenable for destruction. This annotation, a star ($\star$), can be interpreted this way: if the proof-by-pointing algorithm terminates directly after using the rule that introduces the star, the assumption should be kept in the context. Otherwise, the next recursive step of the algorithm should destroy this assumption. We give the rules for this variant of the algorithm in figure 7, where rules that are consumable are annotated with a star.

These rules deserve a few comments. First, there is an alternative form for the $\wedge \mathit{left}$ rules:

$$\wedge\, \mathit{left}'_1 : \frac{\boxed{A}^{\star}, A \wedge B, \Gamma \vdash C}{\boxed{A} \wedge B, \Gamma \vdash C} \qquad \wedge\, \mathit{left}'_2 : \frac{\boxed{B}^{\star}, A \wedge B, \Gamma \vdash C}{A \wedge \boxed{B}, \Gamma \vdash C}$$

In this variant, the non-boxed part of the used conjunct is not added to the context in the premise. Since it is not possible to re-generate the original conjunct from the other assumptions, then this conjunct must be in the new context. But now, the new assumption can be re-generated from the conjunct and it can be marked as consumable. Experience with many users has shown that the variant given in figure 7 is more natural: once a conjunction has been broken into pieces to get one component, it does not feel natural to have to redo the operation to get the other component. However, this variant breaks the usual similarities between conjunction and "for all" quantification on one side and disjunction and "there exists" quantification on the other. The rules $\wedge left'_1$ and $\wedge left'_2$ would have retained these similarities better.

Another comment is related to the implementation of proof-by-pointing when the proof system gives names to assumptions and prevents the user from using the same name twice in the local context, as in Coq (Dowek et al. (1993)). The proof-by-pointing algorithm creates assumptions and destroys them so that some of these assumptions may never be shown to the user. It also occurs that some commands generated by proof-by-pointing are applicable at several places during a proof. Users may find it useful to just copy and paste these commands to re-use them. When this happens, the names of assumptions sometimes clash with the names already existing in the context. The problem can be reduced (but not completely resolved), if the algorithm makes sure that assumptions, which are created and directly destroyed, are named in a different name space. For example, the implementation of the proof-by-pointing algorithm that we devised for the Coq proof system generates names for hypotheses by concatenating a radix and an index. For those assumptions that are known to be consumed, the algorithm uses a different radix.

4.3. Extending the algorithm to new logical operators

Most proof systems have a capability to define new logical connectors. This functionality must be matched by a capability to add rules for proof-by-pointing. We propose a table driven algorithm. Obviously, the elements in the table should correspond to the rules of figure 7. Each rule in the table must provide the following data:

- a flag b indicating whether the rule is a *left* or a *right* rule, that is, whether the rule works in the conclusion of a goal or in its context.
- a pattern P that must match the head of the goal assumption or conclusion for the rule to be applicable,
- a path p that must match the head of the path to the currently selected expression,
- the command pattern C that will perform the rule, that is, the command pattern that will provoke the replacement of the goal by the goals corresponding to the premises of the rule,
- a path p_c in C indicating where the command generated by the recursive call of the proof-by-pointing algorithm should be inserted.
- a tuple (F_r, p_r, b_r) giving the pattern of the premise that contains the box, the path to the box in that premise, and a flag indicating whether the formula is an assumption or the conclusion (r stands for *recursive call*).

Aside from the table containing these rules, the proof-by-pointing algorithm receives three arguments: the formula f where the selection occurred, the path to the currently selected position in that formula p_f, and a flag indicating whether this position is in an assumption or not b_f. The algorithm works as follows:

1. Find the entry in the table that will be applied. This entry is such that $f = \sigma(P)$ for some substitution σ and p is the head of the path p_f. Compute the path p'_f such that $p_f = p + p'_f$ where '+' is the natural concatenation operation on paths.
2. Call recursively the algorithm on the formula $\sigma(F_r)$ the path $p_r + p'_f$, and the flag b_r. This returns a command C'.
3. Return the command $C[p_c \leftarrow C']$.

Each proof-by-pointing rule $\mathcal{R}$ can be implemented as a partial function $f_{\mathcal{R}}$ on tuples *(formula, path, function)*, where *formula* and *path* describe the position that was selected, and where *function* is a partial function that is called on the new premise with the new path and that returns a command. In case of success, the $f_{\mathcal{R}}$ returns the complete command. The parameter *function* acts as a continuation. In all generality, this function does not need to correspond to the recursive call of the proof-by-pointing algorithm, so that it is possible to replace it by any other function capable of creating a command from a formula. This remark is the basis for another extension to proof-by-pointing that we call *point and shoot* (see section 4.4).

Thus, the whole proof-by-pointing algorithm is implemented by a function f_{pbp} that repeatedly calls all the functions $f_{\mathcal{R}}$ with itself (f_{pbp}) as third argument. The functions $f_{\mathcal{R}}$ check whether the rule matches and call recursively f_{pbp} with a new formula and a new path. The termination of the proof-by-pointing algorithm, as presented in Bertot et al. (1994), is ensured by the fact that the path length decreases at each recursive call. For the extensible algorithm, termination is no longer ensured, because the new path $p_r + p'_f$ may be longer than the old one p_f.

In the current implementation, we have decided to trust the users who add new proof-by-pointing rules to make sure that these rules actually perform recursive calls on arguments that decrease in some way.

To implement the separation of name spaces described in the previous section, it is also necessary to run a second pass on the produced command. This pass detects the names that are both created and destroyed in the command and performs an α-conversion on these names.

4.4. Extending the algorithm to new commands

The proof-by-pointing algorithm terminates when no rule matches the formula and path given as arguments, most often because the path has been reduced to length 0. When this happens, the algorithm must provide a command that will be called on the last produced subgoal. To generate this command there are two cases:

- the box may reappear in an assumption. In this case, the command should check whether this assumption matches the conclusion so that the goal might be closed,
- the box may reappear in the conclusion. In this case, the command should look for an assumption that makes it possible to close the goal.

Actually, this is only the most basic behavior. It is possible to provide a range of choices to users, so that they also decide the function that is called at the termination of the proof-by-pointing algorithm. This remark is the basis for *point and shoot.*

When using the point and shoot behavior, users must provide both a position in a goal and the function that will be called when the proof-by-pointing algorithm reaches that position. Such functions can be proposed to users through a menu, or, as we did in our proof environment for Coq, through a variety of key bindings.

The set of available *shoot* functions is extensible. Three examples of such functions are the functions for *point-and-reduce*, *point-and-apply*, and *point-and-rewrite*, that can be implemented in the proof environment for Coq:

- the shoot function for *point-and-reduce* is used to unfold the definition of a function. It returns the command '`Red`' if the box is in a goal's conclusion and the command '`Red in H`' if the command is in an assumption named `H`,
- the shoot function for *point-and-apply* is used to apply a universally quantified formula. It fails if the box is in a goal's conclusion and returns the command '`Apply H with value_for_y`$_1$ ... `value_for_y`$_p$' if the the box is in an assumption of the form

$$\mathtt{H} : \forall \mathtt{x_1} \ldots \forall \mathtt{x_n}.\mathtt{P1} \Rightarrow \cdots \Rightarrow \mathtt{Pn} \Rightarrow \mathtt{Q},$$

  where $\mathtt{y}_1, \ldots, \mathtt{y}_p$ are the variables among $\mathtt{x}_1, \ldots, \mathtt{x}_n$ that do not occur in `Q`,
- the shoot function $f_{rewrite}$ for *point-and-rewrite* is used to perform a rewriting with an equation $\mathtt{t_k} = \mathtt{t'_k}$ when the available assumption has the form $C[t_k] = C[t'_k]$ where the context $C[\ ]$ is known to be injective. The function $f_{rewrite}$ fails if the box is in a goal's conclusion. When the box is in an assumption of the form

$$\mathtt{H} : \mathtt{P} = \mathtt{Q}$$

  there are three possible cases:

  1. if the box is on `P` then the returned command is `Rewrite -> H`,
  2. if the box is on `Q` then the returned command is `Rewrite <- H`,
  3. if the assumption actually is of the form

  $$\mathtt{H} : \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_k}, \ldots, \mathtt{t_n}) = \mathtt{f}(\mathtt{t'_1}, \ldots, \mathtt{t'_k}, \ldots, \mathtt{t'_n})$$

  and the box is either in $\mathtt{t_k}$ (path of the form $1.k.p$) or in $\mathtt{t'_k}$ (path of the form $2.k.p$). Then the function looks up in a table to find a function $\mathtt{proj}_k$ such that for any $\mathtt{t_1}, \ldots, \mathtt{t_n}$ one has $\mathtt{proj_k}(\mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_k}, \ldots \mathtt{t_n})) = \mathtt{t_k}$ and it returns the command

  `Specialize f_equal with f := proj`$_k$`; Intro H'`;*command*$_{rec}$

  where *command*$_{rec}$ is the result of recursively calling $f_{rewrite}$ on an assumption of the form

  $$\mathtt{H'} : \mathtt{t_k} = \mathtt{t'_k}$$

  with a path of the form $1.p$ or $2.p$.

As can be seen from this example, shoot functions can be arbitrarily complex. Also, the notion of procedures that use pattern matching and recursive path walk is more general than just proof-by-pointing.

5. Script Management

The purpose of script management is to record a clean script of the commands sent to the proof system during a working session. Ideally, it should be possible to replay this clean script from the initial state of the proof system and obtain the same final state and this replay should not provoke any errors. Most proof assistants also provide undo and abort commands that enable users to backtrack and to discard commands that led in a wrong direction. The clean script should also not contain commands that have been undone. In our experiments, we have designed a script management tool that also makes the clean script visible to the user, with undo and redo easily available by simple clicks of the mouse.

Technologies for providing user-friendly undo have already been studied, even in multi-user frameworks for simple text and graphics editors (Vitter (1984), Thimbleby (1990)). Leeman (1985) also studied techniques to provide undo operations in a generic way at the level of the programming language. Our problem is slightly different since the history of commands sent to the proof system is more important than the final state of that system (while in an editor, the only thing that matters is the final state of the document). Also, errors play an important role in our study, while erroneous commands are simply not considered in usual presentations of undo mechanisms for editors. For example, we consider cases where even undo commands may provoke an error. This difference is related to the apparent complexity of commands. In our work, commands can be arbitrarily large expressions that may take a long time to prepare, while in editors the commands considered in undo mechanisms are extremely simple: character insertions or deletions, moves of the cursor, etc. The use of two separate processes also introduces a delay between requests and their effects that must be taken into account. In this respect, our problem is also related to the issue of transaction integrity in distributed databases.

In this section, we give an example of the behavior of the user interface with respect to recording commands sent to the proof system. Then, we describe formally the properties expected from the proof system for this behavior to make sense and we describe the corresponding algorithm.

5.1. An example

The style of user interface we implement provides a *command* window where users can edit and record the commands sent to the proof system. Inside this window, there are four different regions, that can be distinguished by different background colors. These regions can be named as follows:

- a *final stack* region, which contains the commands that have already been executed by the proof system (the region with a grey background in figures 1, 2, and 3).
- a *buffer* region. This region contains the command that is currently executed by the proof system.
- a *queue* region. This region contains commands that are waiting their turn to be sent to the proof system.
- a *normal* region. This region contains commands that are being edited, but that have not been queued for execution yet.

When the user requests that a command be executed, it is stored in the *buffer* region until reception of an answer from the proof engine. Depending on whether the answer is a result or an error, the command is taken from the *buffer* region to the *final stack* region or back to the *normal* region, respectively. If there already is a command in the *buffer* region and the user requests the execution of another command, then this command is stored in the *queue* region. When the proof engine terminates its processing of the command in the *buffer* region with a success, the next command in the *queue* region takes its place. However, if an error occurs, all the *queue* region vanishes and its commands return to the *normal* region. Giving the possibility of storing many commands in the *queue* region has proven very useful to users who want to replay long sequences of commands, for example when changing only slightly the statement of a theorem to prove.

When the user requests commands to be undone, the system first removes the commands in the *queue* region, if any. If the last requested command is in the *buffer* region, the undo command is kept in memory on the user-interface side until the proof engine returns an answer. Depending on whether this answer is a result or an error, the undo command is dropped or sent to the proof system. If the last requested command is in the *final stack* region, the undo command is sent directly to the proof engine.

Moving a command from one region to the other is not performed by actually moving the command, but by moving the region. Commands simply change aspect depending on the region they belong to.

The *final stack* region has a read-only property. Commands in this region can be copied to construct new commands, but they cannot be modified. This behavior prevents the user from corrupting the session script. Thanks to this property and the careful design of the script management algorithm, the user interface ensures that the recorded script can be used to replay the current working session.

5.2. ABSTRACT VIEW OF THE PROOF SYSTEM

As far as scripting is concerned, a proof system has an initial state, it can receive commands, emit answers, and change state. There are two kinds of answers: acknowledgments and errors. To formally describe the problem of script management, we have the set *state* of all possible states of the proof system, with a distinguished element *initial_state*. The proof system receives commands taken from a set *command* that we describe later and it returns answers taken from a set $answer = \{ack, error\}$. The proof system is described by its initial state and the transition function *compute*:

$$compute : state \times command \rightarrow answer \times state$$

To safely remove commands that cause errors from scripts, we need to be sure that these commands do not affect the state of the proof system, hence the following postulate:

P_1(errors don't affect states) :

$$\forall s_1, s_2 \in state \quad \forall c \in command \quad compute(s_1, c) = (error, s_2) \Rightarrow s_1 = s_2$$

Although the constraints described so far seem fairly simple, a lot of proof systems do not satisfy this constraint. The first rule to be violated is the constraint that the proof system emit an acknowledgment for every command: most systems simply return the prompt after successfully executing some commands. The second rule to be violated is the rule that error causing commands do not affect the state.

An *undo* command is often found in proof systems to allow users to discard the effect of the last executed commands. In practice, proof engines often have two kinds of commands: the ones that can be undone, and the others. From a theoretical point of view, this may be considered a flaw of the proof engine. For pragmatic reasons, however, it appears that most systems present this kind of feature and so it must be accommodated. Abstracting away from the formal language accepted by the proof engine, we consider that there are only three constructs: *do*, *define*, and *undo*, where commands that can be undone are constructed using the function *do* and commands that cannot be undone are constructed using a function *define*. Still, we need to express that two *do* or *define* commands are different. We express this by stating that *do* and *define* commands contain some data taken from a set *command_contents* that we leave undefined. Hence, *do* and *define* are injective functions from *command_contents* to *command*, such that no command constructed with *do* is equal to a command constructed with *define*.

There are two possible semantics for the *undo* command. The first possibility is that this command enables the user to discard the last command, if this command can be undone. The second possibility is that this command enables the user to discard the last command that can be undone, even if it is not the last one. The systems Coq and Isabelle implement the second possibility. *Do* commands and *define* commands do not affect the same parts of the state and it is quite easy to undo a *do* command without interfering with the *define* commands. This second solution is easily formalized with the following two postulates:

$$
\begin{array}{l}
P_2(\textit{do then undo}\ \text{does nothing}):\\
\quad \forall s_1, s_2, s_3 \in \textit{state} \quad \forall t \in \textit{command\_contents}\\
\quad \textit{compute}(s_1, \textit{do}(t)) = (\textit{ack}, s_2) \wedge \textit{compute}(s_2, \textit{undo}) = (\textit{ack}, s_3)\\
\quad \Rightarrow s_1 = s_3\\
P_3(\textit{define}\ \text{and}\ \textit{undo}\ \text{permute}):\\
\quad \forall s_1, s_2, s_3 \in \textit{state} \quad \forall t \in \textit{command\_contents}\\
\quad \textit{compute}(s_1, \textit{define}(t)) = (\textit{ack}, s_2) \wedge \textit{compute}(s_2, \textit{undo}) = (\textit{ack}, s_3)\\
\quad \Rightarrow \exists s_4 \in \textit{state}\\
\qquad \textit{compute}(s_1, \textit{undo}) = (\textit{ack}, s_4) \wedge \textit{compute}(s_4, \textit{define}(t)) = (\textit{ack}, s_3)
\end{array}
$$

Note that these postulates only describe the interaction of successful commands. Error causing commands do not need to be undone, because they did not do anything in the first place. Also, these postulates do not specify that the *undo* command succeeds. First, there may not be any command to undo, second some systems put a limit on the number of undos that can be performed.

An *abort* command can also be found in a few proof systems. The interactive proof of a theorem usually begins with some kind of start command, where the user gives the statement to prove, then there are a few commands to perform the proof (usually undoable commands, since the search for the proof may require several trials), and it finishes with some saving command, which defines the statement as a new theorem. In the middle of such a proof attempt, the abort command enables the user to discard completely this proof attempt. This is useful when one discovers that the statement cannot be proved. There may be several variants, depending on whether it is possible to start a new proof while one is already under way or whether *define* commands are affected by the *abort* command. In the following, we describe the case where it is not possible to start new proofs and *define* commands happening in the interval between the start of the proof and the *abort* command are discarded[†].

To formally describe *abort*, we need to consider three other kinds of commands: *starting* commands, *aborting* commands, and *saving* commands. Starting commands are constructed with a function

$$start : command\_contents \rightarrow command.$$

The *abort* command is represented by the object $abort \in command$. The behavior of the abort command when it succeeds is described by the following postulates:

$$\begin{array}{l} P_4(\textit{start then abort} \text{ does nothing}) : \\ \quad \forall s_1, s_2, s_3 \in state. \forall t \in command\_contents \\ \quad compute(s_1, start(t)) = (ack, s_2) \wedge compute(s_2, abort) = (ack, s_3) \\ \quad \Rightarrow s_1 = s_3 \end{array}$$

$$\begin{array}{l} P_5(\textit{do then abort} \text{ cancels the do}) : \\ \quad \forall s_1, s_2, s_3 \in state. \forall t \in command\_contents \\ \quad compute(s_1, do(t)) = (ack, s_2) \wedge compute(s_2, abort) = (ack, s_3) \\ \quad \Rightarrow compute(s_1, abort) = (ack, s_3) \\ P_6(\textit{define then abort} \text{ cancels the define}) : \\ \quad \forall s_1, s_2, s_3 \in state. \forall t \in command\_contents \\ \quad compute(s_1, define(t)) = (ack, s_2) \wedge compute(s_2, abort) = (ack, s_3) \\ \quad \Rightarrow compute(s_1, abort) = (ack, s_3) \end{array}$$

For simplicity, we will consider saving commands to be mere *define* commands. This does not clash with property P_6 when no *abort* command can succeed directly after a saving command.

For our formal treatment, we need a clear description of what it means to execute a list of commands. We define the function $compute\_list : state \times command\ list \rightarrow state$ using the following equations:

$$\begin{array}{l} \forall s, s' \in state \quad \forall c \in command \quad \forall tl \in command\ list \quad \forall a \in answer \\ \qquad compute(s, c) = (a, s') \Rightarrow compute\_list(s, [c.tl]) = compute\_list(s', tl) \\ \forall s \in state \quad compute\_list(s, []) = s \end{array}$$

5.3. A state machine for script management

For on-line script management, one wants to maintain the script of processed commands and keep this script clean with respect to undo and abort while working with the proof system. At any time, the user interface process should be able to produce the clean script that makes it possible to reach the current state of the proof system starting from its initial state.

To implement such a functionality, we propose a state machine that receives inputs

[†] This is the behavior implemented in Coq V5.8

coming from users or from the proof system one by one and updates its internal state accordingly. To verify formally the correctness of this state machine, one needs to compare three lists of commands: the commands that were issued by the user, the commands that were actually sent to the proof assistant, and the commands that are stored in the clean script of the session. We suppose that these commands can be extracted from the state by a function *clean_script*. The various criteria for correction are as follows:

1. if the state machine has received as many answers from the proof system as it has sent commands and if the state machine has sent the list of commands l, then the state m of the machine should have the property:
$$compute\_list(initial\_state, clean\_script(m)) = compute\_list(initial\_state, l).$$
2. (*erroneous criterion*) if the state machine has received as many answers from the proof system as it has sent commands and if the state machine has received the list of commands l from the user, then the same equality as above should hold.
3. the list of answers from the proof system to the list $clean\_script(m)$ should contain only acknowledgments.
4. if l is a list of proper commands that can be sent to the proof system without raising an error, if the user requests this list of commands to the state machine, then after the state machine receives as many answers from the proof system as there are commands in l one should have:
$$clean\_script(m) = l.$$
5. if c is any command, m is a possible state of the state machine, and m' is the state of the machine after starting from m, receiving the request c from the user, and receiving the corresponding answer from the proof system, one should have:
$$compute\_list(initial\_state, clean\_script(m')) =$$
$$compute\_list(initial\_state, clean\_script(m) + [c]).$$

These criteria deserve a few comments. We will actually provide a state machine that does not respect the second criterion, for two reasons. The first reason is that the state machine may have received more commands from the user than have actually been processed by the proof system. The second reason is that the following scenario may occur:

a. the user sends several commands, assuming that they will execute without error,
b. the proof system answers that the first command provokes an error.

In this case, there are two choices: either the state machine discards the first command and tries repeatedly with the following commands, maybe raising as many errors as there are commands, or the state machine simply decides to discard all the commands. The second solution seems more user-friendly, but it does not respect the second criterion. In the following we describe a state machine that respects all the other criteria.

The state of this machine consists of the following fields:

- a final stack, intended to receive the processed commands,
- a buffer, intended to receive the command that is currently being processed, and
- a queue, intended to receive the commands that wait for the proof system to process them.

When the user requests a regular command, the state machine simply sends this command to the proof system, storing the command being processed in the buffer. If more commands arrive before the proof system completes the current one, they are stored in the queue. Commands move from the buffer to the final stack only when they have been duly acknowledged by the proof system. The behavior is more complicated when the proof system answers with an error message, or when the user requests undo and abort commands. Intuitively, *undo* and *abort* commands first check whether the undone or aborted commands are still in the queue. The function *clean_script* is defined as follows:

$$clean\_script(stack, buffer, queue) = reverse(stack)$$

Formally, the state machine can be described by the function

$$record(stack, buffer, queue, input) = (stack', buffer', queue'), optional\_command$$

that takes as arguments the various fields of the state given above and an input from the user, or from the proof system, and returns a new state and the command that will be sent to the proof system. The following equalities describe how inputs from the user are treated.

$$\begin{array}{rcl}
record(s, [], [], c) & = & (s, [c], []), c \\
record(s, [c], q, c') & = & (s, [c], q + [c']), none \\
 & & \text{provided } c' \text{ is a proper command} \\
record(s, [c], q + [c'], undo) & = & (s, [c], q), none \\
 & & \text{provided } c' \text{ is a proper command} \\
record(s, [c], q, undo) & = & (s, [c], q + [undo]), none \\
 & & \text{provided } q \text{ does not contain proper commands} \\
record(s, [c], q + [c'], abort) & = & record(s, [c], q, abort) \\
 & & \text{provided } c' \text{ is a } do\text{, a } define \text{ or an } undo \\
record(s, [c], q + [start(t)], abort) & = & (s, [c], q), none \\
record(s, [c], q, abort) & = & (s, [c], q + [abort]) \\
 & & \text{provided } q \text{ contains only } abort \text{ commands}
\end{array}$$

The first equality describes the behavior of the state machine if the proof system is idle when the user inputs a request. All the other equalities describe the cases when the proof system is already busy. The second equality indicates that proper commands are simply recorded in the queue. The third and fourth equalities indicate the behavior of *undo* requests with respect to the queue of pending commands. In particular, *undo* requests make it possible to remove any command from the queue, not just *do* commands. This choice seems more user-friendly than the choice of allowing only to remove *do* commands, in accordance with the proof system's behavior. The next three commands indicate the behavior of *abort* commands. Note that *abort* commands clear the queue up to the last *start* command.

The queue always has the following shape:

$$queue = queue_a + queue_u + queue_r$$

where $queue_a$ contains only *abort* commands, $queue_u$ contains only *undo* commands, $queue_r$ contains only regular commands, and + represents normal sequence concatenation. For this reason, the queue could also be implemented with a triple $(queue_r, n_a, n_u)$ where $queue_r$ only contains regular commands, n_a is the number of pending *abort* commands, and n_u is the number of pending *undo* commands.

The reaction of the state machine to answers from the proof system uses an auxiliary function *next_step* that computes the next command sent to the proof system, if any:

$$\begin{array}{rcl} \mathit{next\_step}(s, c.q) & = & (s, [c], q), c \\ \mathit{next\_step}(s, []) & = & (s, [], []), \mathit{none} \end{array}$$

The rest of the *record* function is described in the following equalities. At this stage, the *buffer* field of the state should not be empty.

$$\begin{array}{rcl} \mathit{record}(s, [c], q, \mathit{ack}) & = & \mathit{next\_step}(c.s, q) \\ & & \text{provided } c \text{ is a regular command} \\ \mathit{record}(c.s, [\mathit{abort}], q, \mathit{ack}) & = & \mathit{record}(s, [\mathit{abort}], q, \mathit{ack}) \\ & & \text{provided } c \text{ is a } \mathit{do} \text{ or a } \mathit{define} \\ \mathit{record}(\mathit{start}(t).s, [\mathit{abort}], q, \mathit{ack}) & = & \mathit{next\_step}(s, q) \\ \mathit{record}(s, [\mathit{undo}], q, \mathit{ack}) & = & \mathit{next\_step}(s', q) \\ \multicolumn{3}{c}{\text{where } s' \text{ equals } s \text{ with the last } \mathit{do} \text{ command removed}} \\ \mathit{record}(s, [c], q, \mathit{error}) & = & (s, [], []), \mathit{none} \end{array}$$

Note that when the proof system returns an error, all the commands that have been queued by the user are discarded. This conservative solution corresponds to the assumption that the user did not expect an error to occur when he started queuing several commands.

This state machine alters in a complex fashion the interaction between the user and the proof system. Depending on the current state of the machine, actions from the user may have an immediate consequence on the proof system or only a delayed one. To achieve reasonable user-friendliness, it is very important to keep the user informed of the current state of the machine. A second aspect needs careful attention: upon receiving an error, the state machine will discard all the commands that have been queued by the user, who may have spent a large amount of time editing these commands. It is very important that these commands will not be completely lost, so that the user can re-use them easily.

We solved these problems by implementing the *stack*, *buffer*, and *queue* fields of the state as moving regions in an editing window, where each region can be distinguished using color. Adding a command to a region is seldom done by actually moving the command from one place to another in the window, but most often by simply extending the region to that expression. An exception occurs when a command is added to the *stack* region: to preserve the continuity and the order of this region, the command may have to be copied at the end of the stack. Requesting a command to be sent to the prover is simply done by pointing at that command and clicking on the "do it" button. Depending on the current state of the machine, the command simply takes the color of the *buffer* or *queue* region. Commands that are undone simply return to the normal color. It is then easy to select these commands again and to click on the "do it" button to reuse these commands.

When a large number of commands are replayed in this manner, it is reasonable to avoid redisplaying the intermediate states during the replay. The user interface can switch off the redisplaying of the *state* area while the replay is taking place and turn it back on when the answer to the last command arrives. To distinguish between "replay" mode and normal mode, it suffices to check whether the *queue* field is empty. The case where one of the commands provokes an error deserves special care: in this case, the queue empties brutally and it is necessary to re-synchronize the content of the *state* area with the proof system. The implementation of a *replay* mode has significantly improved the proof environment.

The editing window only contains proper commands, *undo* and *abort* commands can be requested by simply clicking on appropriate buttons. In this case, it is preferable to use the implementation of the queue as a triple $(queue_r, n_a, n_u)$ that we described earlier (n_a and n_u are the numbers of pending *abort* and *undo* commands).

6. Textual Explanation of Proofs

Proof scripts are very concise and useful objects to represent proofs. They are usually kept for later replay of proofs. Also they can often be reused for proving different theorems. This is the motivation for the special care given to script management tools. Often scripts are also considered as the *only* evidence that proofs have been carried out. Still, they are clearly *insufficient* to be used as a means of communicating proofs. The first problem is that a large familiarity with the proving system is required to understand these scripts, the name of commands are system dependent. To cope with this aspect some systems, like PVS (Shankar et al. (1993)), decorate the proof script with sentence patterns associated to each proof procedure and include the intermediary subgoals between each proof step. But a second problem is that scripts (like programs) often hide some calculations that the reader has to rediscover to capture the intellectual content of the proof. A direct decoration of the commands will not solve this problem.

We believe that it is an important task of the interface at the end of the proof to produce a document that represents the proof and that can be understood by someone who is not familiar with the prover. For these reasons, we feel that proof scripts are inadequate and we advocate the use of *proof objects*, i.e., the objects that contain the *whole* sequence of proved facts to get to the theorem. Each element of the sequence represents an elementary step of reasoning. Proof objects are thus more readily understood than scripts but they are also larger and more detailed.

Textual explanation is an attempt to make proof objects visible in the interface in an intelligible form. After a number of unsuccessful experiments with graphical representations, we are convinced that the best method is to build translators from proof objects to pseudo natural language. In the following, we first motivate our choice of the natural language to represent proof objects and then we sketch how the translation is performed in order to produce concise and pertinent output.

6.1. Proof objects and their presentation

Different methods can be used in building proofs: resolution and tableau methods are popular in the automatic theorem proving community, while natural deduction is favored for interactive proof assistants. Natural deduction, proposed by Gentzen (Szabo (1969)) and further elaborated by Prawitz (1965), is *natural* because it formalizes the reasoning used in ordinary mathematical text. This is also why it is popular in interactive theorem provers and an obvious candidate for experiments in producing text from formal proofs.

Natural Deduction Trees

The original format proposed by Gentzen for natural deduction proof is a tree format. For each connective, two sets of rules are provided. The first gives *introduction rules* and describes how formulae containing the connective may be formed. The second set gives *elimination rules* and describes how formulae governed by the connective can be used. As an example, there is one introduction rule and two elimination rules for conjunction:

$$\wedge\ intro : \frac{A \quad B}{A \wedge B} \qquad\qquad \wedge\ elim : \frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

A *deduction* in this system is a composition of instances of rules forming a tree-like structure. An example of such a deduction is the following tree:

$$\wedge intro: \dfrac{\wedge elim: \dfrac{A \wedge B}{B} \qquad \wedge elim: \dfrac{A \wedge B}{A}}{B \wedge A}$$

An additional mechanism in this formalism is the *discharge* operation which handles hypothetical reasoning. This mechanism can be explained with the introduction rule of implication. The rule is usually displayed as

$$\begin{array}{c} [A] \\ \vdots \\ \supset intro : \dfrac{B}{A \supset B} \end{array}$$

where the brackets around A means that A can be used freely without justification (as a leaf) in the deduction leading to B. We then say that the hypothesis A is discharged by the rule. A *proof* is a deduction in which all leaves have been discharged. To link precisely an assumption with the rule that discharges it, each assumption is numbered. This number is then mentioned again as a superscript whenever the assumption is used and the rule that discharges an assumption is annotated as well. Applying this convention to the previous rules gives the following

$$\begin{array}{c} [A^i] \\ \vdots \\ \supset intro : \dfrac{B}{A \supset B}\ [i] \end{array}$$

The simple proof of *præclarum* that follows illustrates the advantages and disadvantages of such a format.

$$\supset intro: \dfrac{\supset intro: \dfrac{\wedge intro: \dfrac{\supset elim: \dfrac{\wedge elim: \dfrac{x \wedge y^2}{x} \qquad \wedge elim: \dfrac{(x \supset z) \wedge (y \supset t)^1}{x \supset z}}{z} \qquad \supset elim: \dfrac{\wedge elim: \dfrac{x \wedge y^2}{y} \qquad \wedge elim: \dfrac{(x \supset z) \wedge (y \supset t)^1}{y \supset t}}{t}}{z \wedge t}}{x \wedge y \supset z \wedge t}\ [2]}{(x \supset z) \wedge (y \supset t) \supset x \wedge y \supset z \wedge t}\ [1]$$

Using two dimensions to represent the structure of the proof makes operations such as finding the scope of an assumption or finding the premises of a rule easier. The systems EUODHILOS (Sawamura et al. (1992)) and Jape (Bornat & Sufrin (1994)) use this presentation. The drawback is that formulae share the horizontal dimension with the proof structure. Consequently, proof trees tend to grow much more in width than in height. This phenomenon is even amplified by the fact that the leaves, being assumptions, are large formulae. Automatic ellision to hide parts of the proofs could be a solution but it is hard to find a satisfactory strategy. Even for small proofs, we have found the natural deduction proof-tree layout impractical.

Natural Deduction in Linear Format

As an alternative, several authors have proposed linear presentations of natural deduction proofs: Fitch (1952), Kalish et al. (1980). This style is used in theorem provers such as Mural (Jones et al. (1991)) and TPS (Andrews et al. (1992)). The basic idea is to represent a proof as a vertical sequence of lines, where each line is either an axiom, an hypothesis, or the consequence of previous lines. To give an idea of such format, here is the same proof of *præclarum* in the format described in Fitch (1952):

1	$\quad\quad (x \supset z) \wedge (y \supset t)$	*hyp*
2	$\quad\quad\quad x \wedge y$	*hyp*
3	$\quad\quad\quad x$	2, $\wedge$ *elim*
4	$\quad\quad\quad x \supset z$	1, $\wedge$ *elim*
5	$\quad\quad\quad z$	3 and 4, $\supset$ *elim*
6	$\quad\quad\quad y$	2, $\wedge$ *elim*
7	$\quad\quad\quad y \supset t$	1, $\wedge$ *elim*
8	$\quad\quad\quad t$	6 and 7, $\supset$ *elim*
9	$\quad\quad\quad z \wedge t$	8 and 5, $\wedge$ *intro*
10	$\quad\quad x \wedge y \supset z \wedge t$	2-9, $\supset$ *intro*
11	$\quad (x \supset z) \wedge (y \supset t) \supset x \wedge y \supset z \wedge t$	1-10, $\supset$ *intro*

Each line contains a number, a formula, and the justification of the formula. For example, at line 5, "z" is the consequence of applying $\supset$ *elim* with lines 3 and 4. A special symbol *hyp* is used when introducing an hypothesis. Additionally, a vertical bar is used to denote a *subordinate proof*. Finally discharging an assumption is represented by a justification taking a subordinate proof whose first line is an *hyp*. Lines 10 and 11 are examples of such a phenomenon. Note that assumptions, once allocated a line number, need not be repeated.

The linear style obviously gives a more vertical presentation than the tree style. But because of constant references to earlier lines, such proofs are reminiscent of assembly language programs, and reading them is just as tedious.

6.2. Transducing proof objects

Different experiments have been carried out to produce explanations out of proofs: Felty (1988), Huang (1994), and Edgar & Pelletier (1993). Our approach is somewhat more pragmatic. We only want to produce pseudo English, using indentation to outline the structure of the proof. For example, the result we expect for præclarum is the following:

Theorem: *præclarum*
Statement
$\forall x, y, z, t \colon Prop.\, (x \supset z) \wedge (y \supset t) \supset x \wedge y \supset z \wedge t$
Proof
Let x, y, z, t be propositions
Assume we know $x \supset z$ (i) and $y \supset t$ (j)
Assume we know x (k) and y (l)
- Using i with k we deduce z
- Using j with l we deduce t
Altogether we have $z \wedge t$

The basic idea is simple: we associate a textual equivalent to each rule of the logic. For example, using the relation $\triangleright$, we define below the layout associated to the $\wedge$*introduction*:

$$\frac{P \qquad Q}{A \wedge B} \quad \triangleright \quad \begin{array}{l} \text{- } P \\ \text{- } Q \\ \text{Altogether we have } A \wedge B \end{array}$$

where P and Q represent respectively the proof of A and the proof of B.

Then the technique must be slightly refined to obtain more concise and pertinent explanations. So far we have implemented three different refinements:

1. melting together repeated constructs
2. accessing theorems and definitions
3. altering the direction of the discourse

Repeated constructs

An example where improvement can be easily obtained is when the proof contains consecutive applications of the same rule. It is often the case that a shorter presentation can be found. The rule for $\wedge$*introduction* is an example of such a rule. Trying to prove $A \wedge B \wedge C$ means proving $A \wedge B$ and C (or A and $B \wedge C$). The textual explanation would be of the form:

$$\begin{array}{l} \quad\quad \vdots \\ \quad \text{- } A \\ \quad\quad \vdots \\ \quad \text{- } B \\ \text{- Altogether we have } A \wedge B \\ \quad \vdots \\ \text{- } C \\ \text{Altogether we have } A \wedge B \wedge C \end{array}$$

Melting together the sequences of applications reduces the length of the text and more importantly avoids a drifting to the right:

$\vdots$
- A
$\vdots$
- B
$\vdots$
- C

Altogether we have $A \wedge B \wedge C$

Theorems and Definitions

Applications of theorems and definitions are very frequent in proof objects and most of the time do not deserve any explanation. Any application of a theorem should hide the specialization and the deduction that are present in the proof object. For example given the commutativity of disjunction:

$$\vee commutative : \forall A, B.\, A \vee B \supset B \vee A$$

A naive presentation of an application of this theorem would look like the following:

$\vdots$
- $x \vee y$
- Specializing $\vee commutative$, we get $x \vee y \supset y \vee x$

We deduce $y \vee x$

the improved version gives a more concise explanation:

$\vdots$
$x \vee y$
By $\vee commutative$, we get $y \vee x$

Rupture in the discourse

In Edgar & Pelletier (1993), the authors present two ways of chaining the explanation. *Forward chaining* goes from the facts that are known to what is to be proved. *Backward chaining* goes in the opposite direction. We adopt the forward style, except for some intermediate results which are announced before being proved, so as not to lose the reader. Such a case occurs for induction principles. If we take the usual induction on natural numbers:

$$Induction\ on\ Nat : \forall P.\, (P\, 0) \wedge (\forall n.\, (P\, n) \supset (P\, n+1)) \supset \forall n.(P\, n)$$

The explanation of a proof by induction in forward chaining style would be:

$\vdots$
- $Q\,0$

$\vdots$
- $\forall n.(Q\,n) \supset (Q\,n+1)$

Using *Induction on Nat*, we get $\forall n.\,(Q\,n)$

Announcing first the induction is important because induction drives the reasoning and the different induction cases that may be cumbersome:

To prove $\forall n.\,(Q\,n)$ using *Induction on Nat*, we have two cases
Case$_1$: we will prove $Q\,0$
$\vdots$
Case$_2$: Assume $(Q\,n)$, we will prove $(Q\,n+1)$
$\vdots$

A similar situation occurs at the top level where one prefers to first announce the theorem and then give the proof as in the example of *præclarum*.

6.3. Implementation

The implementation of textual explanation is split in two. Inside the prover, transformations are applied to the raw proof object to extract relevant information and produce a refined proof object. In the interface, a `PPML` specification gives the layout and the phrases associated to the different constructs. Note that the phrases can thus be easily changed by the user. A precise definition of textual explanation in the Coq environment can be found in Coscoy et al. (1995).

7. Conclusion

In sections 2 and 3, we have presented methods that make it possible with little effort to plug a graphical user interface onto interactive proof systems. In sections 4, 5, and 6, we have described features that can be adapted to a wide variety of interactive proof systems in this framework.

The most characteristic aspect of this work is the pervasive influence of interactive manipulation of structures. The advanced capabilities of proof-by-pointing and textual explanation of proofs rely, obviously, on the structure of logical formulae. Our conclusion is that the structure of formulae must appear in the user interface and that it is pertinent to use many tools around this notion of structure, like template directed editing, tree navigation, and elision mechanisms. The importance given to structural manipulation need not prevent users from communicating in a textual mode, and we have shown that structure and textual editing could collaborate efficiently.

Another important concept is the architecture used to assemble a proof system and a user interface. By relying on processes that communicate through a simple ascii protocol, we have made it possible to separate logical and meta-mathematical concerns from user interface concerns in the design of the complete system. This separation has two immediate consequences: first the data structures used for communicating with the user do not need to be isomorphic to the data structures used internally in the proof engine. The internal structure can be made more abstract and more suited for automatic operations while the external data structure can adapt more faithfully to usual mathematical practice. Second, this working methodology has lead to remarkable opportunities for modularity and software reuse. Indeed, we have experimented with many proof systems and the same basic tools have also been used for user interfaces adapted to computer algebra systems (Kajler (1992)).

The use of a graphical user interface makes it possible to envision new means of communication that would not be feasible with a traditional simple interface based on character streams. Proof-by-pointing is a characteristic example: the basic notion in this functionality is the notion of position. The user can communicate with the proof system by simply referring to an expression at a specific position in a context that may be a very large formula. Rather than the expression value, it is its *position* in the larger formula that is used for command generation. Expressing this position would be very cumbersome if the user had to encode it in some textual form, while the use of a pointing device makes it a trivial matter. More generally, the graphical representation will make it possible to use the symbolic system's output as material for generating the next input command and to provide direct manipulation on a representation of the system's state.

There is room for future work in all the domains described in this paper. For proof-by-pointing, we have described how the algorithm could be extended to new logical combinators by having the user specify new rules. We believe these rules should themselves be inferred directly from the new combinators' definition. For script management, it seems necessary to go beyond the linear structure of scripts. There are dependencies between commands that are not adequately represented by the sequential aspect: it should be possible to discard an old command and only those commands that depend on this old command, without imposing the burden of replaying everything on the user. This issue might be solved by computing a closer connection between the command script and the proof object constructed in the proof system, since the proof object naturally contains the dependencies between various steps in the proof. Finally, textual explanation of proofs can progress in two ways: first it should be possible to see the text of a proof not only after the proof is terminated, but also as it is constructed during the proof, with holes corresponding to the remaining subgoals to prove. Second, a connection between the proof object and the command script should also be exploited to produce more concise text: sometimes the method used in a proof is more important than the actual inferences. This method is better described by the commands than by the proof object.

Acknowledgments

This work is supported in part by the Esprit Basic Research Action "Types for Proofs and Programs". We want to thank the developers of proof systems that have collaborated in our experiments and gave encouraging feedback. Also we are grateful to the developers of the Centaur system for their generous support.

References

Andrews, P., Issar, S., Newmith, D., & Pfenning, F. (1992). The tps theorem proving system. *Journal of Symbolic Logic*, 57:353–354.

Bertot, Y., Kahn, G., & Théry, L. (1994). Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160.

Bornat, R. & Sufrin, B. (1994). Jape: a literal, lightweight, interactive proof assistant. Technical Report 641, Queen Mary and Westfield College, University of London.

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., & Pascual, V. (1988). Centaur: the system. *Software Engineering Notes*, 13(5). Third Symposium on Software Development Environments, (Also appears as INRIA Report no. 777).

Caferra, R. & Herment, M. (1995). A generic graphic framework for combining inference tools and editing proofs and formulae. *Journal of Symbolic Computation*, 19:217–243.

Cameron, D. & Rosenblatt, B. (1991). *Learning GNU Emacs*. O'Reilly & Associates, Inc.

Clément, D. (1990). A distributed architecture for programming environments. *Software Engineering Notes*, 15(5). Fourth Symposium on Software Development Environments.

Clément, D., Montagnac, F., & Prunet, V. (1991). Integrated software components: a paradigm for control integration. In *European Symposium on Software Development Environments and CASE Technology*, volume 509 of *Lecture Notes in Computer Science*.

Constable, R., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harber, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., & Smith, S. F. (1986). *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall.

Coscoy, Y., Kahn, G., & Théry, L. (1995). Extracting text from proofs. In *Typed Lambda Calculus and its Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 109–123. Springer-Verlag.

Dowek, G., Felty, A., Herbelin, H., Huet, G., Murthy, C., Parent, C., Paulin-Mohring, C., & Werner, B. (1993). *The Coq Proof Assistant User's Guide*. INRIA. Version 5.8.

Déry, A.-M. & Rideau, L. (1994). Distributed programming environments: an example of message protocol. Rapport Technique 165, INRIA.

Edgar, A. & Pelletier, F. (1993). Natural language explanation of natural deduction proofs. In *First Conference of the Pacific Association for Computational Linguistics*.

Felty, A. (1988). Proof explanation and revision. Technical Report MS-CIS-88-17, University of Pennsylvania.

Felty, A. (1989). *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania.

Fitch, F. B. (1952). *Symbolic Logic: an introduction*. Ronald Press Company.

Gordon, M. J. C. & Melham, T. F. (1993). *Introduction to HOL : a theorem proving environment for higher-order logic*. Cambridge University Press.

Gray, S., Kajler, N., & Wang, P. (1994). MP: A protocol for efficient exchange of mathematical expressions. In *International Symposium on Symbolic and Algebraic Computation (ISSAC'94)*. ACM.

Griffin, T. (1988). *Notational definition and top-down refinement for interactive proof development systems*. PhD thesis, Cornell University.

Huang, X. (1994). Reconstructing proofs at the assertion level. In *Proceedings of CADE-12*, volume 814 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Jacobs, I., Montagnac, F., Bertot, J., Clément, D., & Prunet, V. (1993). The Sophtalk reference manual. Rapport Technique 150, INRIA.

Jones, C., Jones, K., Lindsay, P., & Moore, R. (1991). *MURAL: A Formal Development Support System*. Springer-Verlag.

Kajler, N. (1992). CAS/PI: A portable and extensible interface for computer algebra system. In *International Symposium on Symbolic and Algebraic Computation (ISSAC'92)*, pages 376–386. ACM.

Kalish, D., Montague, R., & Mar, G. (1980). *Logic : techniques of formal reasoning*. Harcourt Brace and Company.

Leeman, G. B. (1985). A formal approach to undo operations in programming languages. *ACM Transactions on Programming Languages and Systems*, 8(1):50–87.

Luo, Z. & Pollack, R. (1992). LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh.

Magnusson, L. & Nordström, B. (1994). The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes Computer Science*, pages 213–237. Springer-Verlag.

Paulson, L. C. & Nipkow, T. (1994). *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag.

Prawitz, D. (1965). *Natural Deduction, a proof-theoretical study*. Almqvist & Wiksell.

Reps, T. & Teitelbaum, T. (1988). *The Synthesizer Generator: a system for constructing language based editors*. Springer-Verlag. (third edition).

Ritchie, B. (1988). *The design and implementation of an interactive proof editor*. PhD thesis, University of Edinburgh.

Sawamura, H., Minami, T., & Ohashi, K. (1992). Proof methods based on sheet of thought. Research Report IIAS-RR-92-6E, Fujitsu Laboratories Ltd, Shizuoka.

Shankar, N., Owre, S., & Rushby, J. M. (1993). A tutorial on specification and verification using PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA. (Beta Release).

Syme, D. (1995). 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.

Szabo, M. E. (1969). *The Collected papers of Gerhard Gentzen.* North-Holland.

Théry, L., Bertot, Y., & Kahn, G. (1992). Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5):120–129. 5th Symposium on Software Development Environments.

Thimbleby, H. (1990). *User Interface Design.* Frontier Series. ACM Press.

Vitter, J. S. (1984). Us&r: A new framework for redoing. *IEEE Software*, 1(4):39–52.