# A TACTIC-DRIVEN SYSTEM FOR BUILDING PROOFS

*L. Hascoët*
*INRIA Sophia-Antipolis*
November 15, 1987

**Abstract**

We present a system to ease the process of building proof trees, which may be driven either manually by the user or automatically by user-defined tactics. Applications include: testing various strategies of resolution, partial evaluation of programs written with inference rules, building a theorem prover that automates structural induction strategies. We describe how to use the system for these applications. We examine the technical implementation considerations, as well as provide a brief user's manual.

## 1. INTRODUCTION

### 1.1. Aim of This Work

This paper is intended to fulfill many goals at one time. First it presents a new tool to build proofs, by building proof trees from inference rules and tactics. We give some background as to why we want such a tool. As well, we give a detailed description of implementation problems that we encountered, while programming this system using TYPOL [6][7], Prolog [1], and Lisp [3]. Second, since we would greatly appreciate people using our system, we include also a short user's manual, where we describe:

*i)* how to drive manually the process of building a proof tree, and also

*ii)* how to write tactics that control the proof tree construction.

We finish by presenting some ideas for applications of our system. Some of these are only proposals, while others have been implemented in the form of tactics in the system, and have given interesting results. This system is not fixed yet, and we hope that we will be able to improve it substantially. This paper is just here to set the mark, from which we should go on; it summarizes the results of work that started at the end of '86.

## 1.2. The Origins of our Work and the CENTAUR Background

This work is part of the development of the CENTAUR system [15], which is meant to become a comfortable, efficient and generic programming environment. One of the main tools in CENTAUR is a language using inference rules, called TYPOL [7]. TYPOL is used to define all kinds of static and dynamic operations on programs, for instance type-checking or interpreting. The inference rules written in TYPOL basically define a way of traversing an abstract syntax tree while computing information at the same time. In some sense, TYPOL can be loosely related to attribute grammars. We call this way of specifying semantics "natural semantics" [4].

A TYPOL program is made of a collection of TYPOL rules. Typical TYPOL rules are like the following, that define the (executable) meaning of the "while" construct in a Pascal-like language:

$$\frac{\sigma \vdash \textsc{exp} : \text{“}true\text{”} \qquad \sigma \vdash \textsc{stm} : \sigma' \qquad \sigma' \vdash \textbf{while}\ \textsc{exp}\ \textbf{do}\ \textsc{stm}\ \textbf{end} : \sigma''}{\sigma \vdash \textbf{while}\ \textsc{exp}\ \textbf{do}\ \textsc{stm}\ \textbf{end} : \sigma''} \tag{1}$$

$$\frac{\sigma \vdash \textsc{exp} : \text{“}false\text{”}}{\sigma \vdash \textbf{while}\ \textsc{exp}\ \textbf{do}\ \textsc{stm}\ \textbf{end} : \sigma} \tag{2}$$

Usually, TYPOL programs are first compiled in a straightforward manner into C-Prolog [1], or MU-Prolog [17], then the Prolog program is executed. This approach has revealed two main problems.

- First, the semantics are not the same in TYPOL and in Prolog. While the former should be nondeterministic in theory, the latter is in practice bound to the depth-first strategy of Prolog. This is why we are now switching to MU-Prolog. While MU-Prolog is more flexible, some problems still remain.
- Second, when one wants to debug a TYPOL program, one typically has to go into Prolog trace mode, which is hard to read, when it exists. Also, one may want to show the meaning and the execution of TYPOL rules from the TYPOL level, *i.e.* without compiling. Since we need a way of interpreting TYPOL programs instead of compiling them, we decided to build a TYPOL interpreter.

A possible method to solve these problems is to represent the successive inference steps by a growing proof tree. This proof tree would start by the question (predicate or goal) asked to TYPOL. This question could be for instance "what is the dynamic semantics of this program". A typical step in this growing process is: select any unsolved goal in the proof tree; select a TYPOL rule which solves this goal; apply the rule to the goal. This results in a new proof tree, which we can now grow even further. Now, if we fix the order in which goals and rules are selected, using the classical depth-first strategy, we obtain exactly a TYPOL interpreter or debugger. On the other hand, if we take any other strategy to select the goals and rules, we obtain a way of generating other proof trees. For instance, we can simulate true nondeterminism, or a breadth-first strategy.

When trying to make a convenient system, we encountered several practical problems : we want the user to select goals and rules easily, with a mouse for instance. We also want the system to print the results in a readable format. To solve these, we used the full power of the CENTAUR system, that provides tools for manipulating windows, with menus and mouse.

## 1.3. The Need for Tactics

We also discovered on the way more complex and interesting applications of a proof tree builder. We felt for example that it would be interesting to automatize in some way structural induction, while building real proofs written with TYPOL rules. One example of this will be given later in this paper. Another example can be found in J. Despeyroux paper [5], which presents a very nice proof of equivalence between two TYPOL programs. However, this proof is tedious to check, because it implies building a huge amount of complex proof trees, built from TYPOL rules. This paper showed also a clear need for automatizing structural induction.

Another application comes from how to perform partial evaluation of a TYPOL question. Here some "input" values are not known at partial evaluation time. We describe this example in the last section of this paper.

For these applications, we need tactics to drive the building of the proof tree, instead of letting the user select and direct each step. In another domain, this is what happened in the LCF project [11]. Large parts of these problems can be helped automatically by a tactic. It is a natural trend in computer science to leave boring tasks to the computer. Thus, there should be two ways of driving the construction of a proof tree, by hand, and by tactics. Such tactics, as we shall see, will be written as fairly simple inference rules.

The main tactics we have already defined are the depth-first strategy, and a partial evaluation method. We hope to complete a structural induction strategy in the near future.

### 1.4. Plan of This Paper

Here we outline the remainder of this paper. Section 2 presents a global description of the system. To compare with other proof editors, you should need to read this section only. There is a quick description of the interface we provide between the system and the user. We describe the operations on proof trees that we consider atomic and the control the user has while calling these atomic operations.

Section 3 then gives precise information on how the system is implemented. It is aimed at people who want to know all the tricks and hacks, to compare to theirs. This description ranges from the clean algorithms used effectively to build proof trees, to the techniques used to force garbage collection in Prolog. We also present time optimization techniques, written in Lisp, to remember previous states of the display. Another interesting point is the use of Prolog backtracking to implement "undo" control in the building of the proof tree.

For actual users of the system, section 4 provides a brief user's manual. This explains how to call it from under CENTAUR and how to use the various menus. We also give indications on how to write your own tactics and use them.

The last section is devoted to some examples of use of the system, mostly with tactics. We give ideas of problems that may be partly solved by a tactic-driven construction of a proof tree. Some of these tactics have not yet been written, although we believe they present no new problems. For those tactics that are already implemented and tested, we give a description of the results obtained.

## 2. OUTLINE OF THE SYSTEM

The proof tree editor is completely built on top of the CENTAUR programming environment. CENTAUR is composed of two main parts. First a LISP data base, containing for instance the edited programs, and which allows manipulations of the abstract syntax tree of these programs. The Lisp engine also provides all the interfaces, with "windows" and a "mouse". Second, there is a Prolog engine, which allows to execute the TYPOL specifications. For instance, TYPOL executable specifications of type-checkers are in fact executed in Prolog. Within CENTAUR, we consider all documents, programs, etc, as syntax trees. These syntax trees are displayed in some window, using pretty-printing routines. Conversely, when one wants to "move the cursor" to some place in a document, one moves in fact a position marker in the abstract syntax tree. This can be done using a mouse or using commands like "up", "right", or "down".

The system needs, to start with, a set of axioms and inference rules from which it will build the proofs. These axioms are the rules from a given TYPOL program, which is read using CENTAUR's input primitives. Then two windows are opened, in which the proof tree is going to be built.

One window, which we call the "proof" window, will show the partial proof tree throughout the constructing process. The other window, which we call the "rules" window, contains the list of applicable TYPOL rules. To begin, the user has to provide the starting point of the proof tree, for instance the predicate he wants proved, or a partial proof tree.

## 2.1. **The Basic Operations**

We consider that the process of building a proof tree can be split into many calls to some basic operations. We present here these operations, along with other control commands that proved to be useful.

### 2.1.1. *Selection of the next goal to expand*

When we have a partial proof tree, *i.e.* a proof tree in which some goals are only hypothesis, and need further work to be proved, it is the natural first step to select from these goals the one to work on first. This is a typical operation which may be performed by the user, or by a tactic. For instance, the classical depth-first strategy always chooses the first unsolved goal met when traversing the proof tree in preorder. In our system, the user selects whatever unsolved goal he wants. To do this the user uses the mouse to select a goal of the proof tree in the proof window. If the user doesn't click exactly on an unsolved goal, the system automatically recovers, by finding the "nearest" unsolved goal.

### 2.1.2. *Goal expansion*

After selecting a predicate the user wants to expand it. For that, the system gives the user the list of all TYPOL rules that can apply to this goal. From a Prolog viewpoint, this means searching for all the rules whose conclusion (head) matches the given goal or question. This operation is of course always performed by the system itself. It is a repetitive operation, that is programmed once in TYPOL. When the user clicks on the "Expand" button of the menu, the system searches in the starting TYPOL program, gathering all applicable rules in one TYPOL "set". This set appears in the "rules" window.

### 2.1.3. *Selection of the rule to apply*

Now that the user is given all the applicable rules, he has to choose among them. This is analogous to selecting the goal to solve, with the difference that the user is then required to click on a TYPOL rule in the "rules" window. The classical Prolog strategy, for instance, always takes the first applicable rule. Then, in case of backtracking, it selects the next one. When no rule remains, the call is said to "fail", and backtracking goes up to previous step. Like the previous selection of a goal, this operation may be left to the user or performed by the tactic. The same recovery procedures are used to correct the selected position when it is not exactly on a TYPOL rule.

### 2.1.4. *Application of a rule to a goal*

After the user selected the rule to apply to the previously selected goal, the system can do the necessary work. This means computing a new proof tree, obtained from the previous one in the following way: unify the selected goal with the bottom predicate of the selected rule; propagate the resulting substitutions inside the proof tree and the selected rule; replace in the instantiated proof tree the goal by the instantiated rule. In this process, the system also has to be careful about renaming variables to avoid $\alpha$-conversion problems. In TYPOL, the same identifier in two different rules doesn't denote the same variable but different, unrelated variables. Now, the goals which are on top of the rule become new unsolved goals. The above four operations clearly form a loop, which can be repeatedly performed.

### 2.1.5. *Executing a goal*

Some predicates in TYPOL programs are predefined. This means they can only be solved by a call to some other language, in which the predicate is defined. Here, this other language is Prolog.

Thus we have to solve a translated version of the predefined predicate. This arises for instance in the case of arithmetic or input-output predicates.

The execution operation applies to such unsolved predefined goals. In this case the goal is first translated to its Prolog version. Then Prolog is called to solve it. If it fails, then the "execution " fails also. If it succeeds, yielding an instantiated version of the question, then this answer is translated back to a TYPOL predicate. Thus we build a temporary TYPOL axiom which is the answer to the particular TYPOL goal. This axiom is then applied to give the resulting proof tree. In this case, no new unsolved goal is generated, since the applied rule is in fact an axiom.

#### 2.1.6. *Simplifying the proof tree*

One useful operation, which doesn't contribute directly to the construction of the proof tree is simplifying the proof tree. When a proof tree becomes too complex, or too hard to read and understand, the system provides a way of shrinking it, by removing intermediate steps (intermediate goals). By just keeping the bottom predicate and the list of all unsolved goals on top of the proof tree, the system yields a new TYPOL rule. That is in fact the theorem that has been proved so far. This is a nice shortcut for displaying interesting results.

#### 2.1.7. *Modifying the proof tree*

Sometimes the user modifies the displayed proof or the displayed set of rules in the windows. Thus we must have a way to inform the system to update its internal representation for this proof tree or these rules. This is useful for example when restoring a partial proof tree from a previous session.

### 2.2. **Calling and Combining the Basic Operations**

A proof tree editor can be considered as a system in which the user is allowed to call all the basic operations repeatedly, without knowing what happens inside each of these operations. Our system uses the facilities of CENTAUR to give the user an interactive way of driving the proof. This requires programming not only in high-level TYPOL, but also in lower-level Lisp, in which all the user interfaces are written. Details will be found in next section.

As we explained for each atomic command, the user may perform all the selection operations by positioning the mouse on some region of the proof or rules. All the other operations are activated by selecting an item in a menu. All results are displayed in CENTAUR windows. Whenever the user is required to drive the next step of the proof, everything he needs is displayed in these windows. The fact that the user has to use menus, combined with the numerous error recoveries and error messages included in the system, leads to a fairly safe proof editor. Sometimes, the user wants to have a better control on the proof tree. The following routines are intended to give this control

#### 2.2.1. *Undoing operations*

Undoing previous operations is obviously an indispensable command. For example, the user may find that he has not chosen a good rule, or perhaps he wants to simulate backtracking. In that case, he must be allowed to go back to any previous state, and then call another operation to go on building the proof tree in another direction.

For this purpose, the menus always provide an "undo" option. When the user selects this option, he goes back to the most recent point where he was given a choice (either to select a goal or rule, or to call an atomic operation). This means that everything after this point is forgotten, and the two windows display exactly what was displayed at that time. The user may then start forward again, by selecting and calling operations, or continue undoing steps. Each undo brings him back to an older decision point. When the user, calling repeatedly undo, comes back to the beginning of the proof, any extra undo is forbidden and results in an error message.

2.2.2. *Opening and closing levels*

Operations are provided to suspend a proof, start to develop another proof, and at the end of this proof return to the previous one. This may be useful when a lemma is needed. This is useful for tactics, but the user may call these routines himself. This is done through the notion of levels. Initially, the system opens a level, which is numbered 0. Closing this level results in stopping the program. At any point, the user can open a level above the current level. In this new level, he may build the proof tree he wants. No "undo" is allowed beyond the beginning of this level. When he is satisfied with the result, the user may close the level, and then come back to the previous one. From then on, the terminated above level is considered as a black box. It is possible to undo it, and this results in undoing all the operations as a single one.

2.2.3. *Garbage collection*

The underlying Prolog engine uses a lot of memory space to remember all the backtracking points. It is possible to throw away all this information in order to save space. Thus the system provides a command called "checkpoint". When this command is called, all backtracking information between the current state and the opening of the current level is removed, saving a huge amount of Prolog global stack. Of course, it is then impossible to undo the operations between the current point and the current level opening, since the current state now becomes the *first* state of the current level.

## 2.3. **Tactic Facilities**

Tactics are defined as a set of TYPOL rules. The call to a basic operation is written in TYPOL in the following way:

$$\sigma_1 \overset{\text{command}}{\vdash} \text{CMD} : \sigma_2$$

where $\sigma_1$ represents the state of the proof before the command, and $\sigma_2$ the state after the command. This is useful for writing tactics, because these calls can be placed inside the TYPOL rules that define the tactic. These tactic rules call basic routines to decide the goals and rules to select, basic commands like "expand" or "apply", and of course other tactics. It is natural to define tactics with TYPOL rules, as it gives us all the power of backtracking. When a tactic fails, we can choose another tactic, for instance.

There is a drawback in writing tactics in TYPOL. Namely TYPOL rules are considered in a clean, nondeterministic way. Our tactics in TYPOL are procedural and are, in fact, Prolog programs paraphrased in TYPOL. So we want to insist that tactics in TYPOL are a convenient misuse of inference rules. The same problem arises with rewriting rules, which are clean as long as they are written as equations. When equal signs are replaced by arrows, then the rewriting is more procedural.

Tactics written in TYPOL look rather natural, and it is pleasant not to use too many languages at a time. Another good point is that it is very easy to combine tactics and user's decisions. In fact everything is regarded as a tactic, even the fact of asking the user for a decision. This means there exists a special tactic, called "user", which asks the user whenever a decision is to be taken. The definition of this tactic is the single following rule:

$$\frac{\text{display}{<}\sigma_1 \rightarrow \sigma_2{>} \qquad \text{menu}{<}\sigma_2 \rightarrow \text{CMD},\sigma_3{>} \qquad \sigma_3 \overset{\text{command}}{\vdash} \text{CMD} : \sigma_4 \qquad \text{tactic}{<}\text{TAC}{>} \qquad \sigma_4 \vdash \text{TAC}}{\sigma_1 \vdash \text{“user”}}$$

which means that, to apply the "user" tactic, the system first needs to display the current state, then to ask the user to select things and then to choose what basic operation to do (The user may answer “undo”). Then it performs the command asked by the user. The “tactic” predicate calls for the name of the current tactic, and finally the system calls this current tactic, which in most cases will be “user” again. Notice that, since the $\sigma$ environments contain some information about the state of the display, the “display” predicate itself changes this environment. This happens as well with the “menu” predicate.

Since everything is written in the form of a tactic, it is very easy to mix automatic tactics and user-driven proof construction.

*i)* When in “user” mode, one may call a tactic to solve a particular goal, *e.g.* the depth-first tactic. When that tactic terminates, the control goes back to the user, and the tactic is “user” again.

*ii)* If one wants to define a tactic which, at some time, has to ask the user to make a crucial decision, then the tactic calls the “user” tactic for a while. Once the user has finished, control switches back to the tactic.

It is recommended that tactics should use from time to time garbage collection, and also should open and close levels to clearly separate important steps of reasoning. Also, to speed up tactics, then do not display their intermediate steps, unless the definition of the tactic explicitely says to do so. This is done with a predicate called “glimpse”. Thus, a tactic-driven decision is faster than a user-driven one, because no time is spent in displaying the state of the proof for the user, as well as in understanding the command given by the user. Examples of tactics are given in the last two sections.

## 3. IMPLEMENTATION

Here we describe those aspects of the implementation that we find most important, original or interesting. The underlying methods are informally presented, and are sometimes illustrated by parts of the system. To improve readability, we only show modified portions of the system, in which remain only the parts relevant to the described technique.

Some of these techniques are clean Prolog algorithms, while others could be called “dirty tricks”. A few of the techniques are written in Lisp, in which the notion of “cleanliness” is less defined. We finish this section by discussing speed issues of the system.

### 3.1. Typol and Prolog Techniques

As we mentionned above, we use many CENTAUR facilities, for defining menus and manipulating the windows.

Also, when suspending a TYPOL execution to give control to the user, we use a feature of the Le_Lisp system that allows us to open a Lisp session on top of another one. Centaur automatically gives the control back to the proof building process when the user is through.

#### 3.1.1. *Internal representation of proof trees*

The goals we have:
We want to use Prolog unification to unify the selected TYPOL goal and the bottom predicate of the selected TYPOL rule. We also want a way of displaying this internal representation on the screen, such that the displayed object is a normal TYPOL object. Conversely, we want to be able to translate TYPOL objects to their internal representation.

The representation we chose:
Our internal representation of a TYPOL syntax tree has two parts. The first part is a copy of the abstract syntax tree, where all subtrees representing TYPOL variables are replaced by PROLOG variables. The second part is the mapping from PROLOG variables to the TYPOL variable that

they replaced. The fact of replacing TYPOL variables by Prolog ones can be considered as a partial compilation, which doesn't change the rest of the structure of the TYPOL program.

The way this works:
We first provide the translator from TYPOL terms to their internal representation. This process builds and uses the mapping from Prolog variables to TYPOL ones, to ensure that the same TYPOL variable is "compiled" to the same Prolog variable. Now unification between two TYPOL terms is trivial, because it is Prolog unification. We also give the way to reconstruct the TYPOL form, which is easy using the mapping part of the internal representation. We just have to unify every Prolog variable with its TYPOL version. This ensures that we do not lose the actual TYPOL variable names, that are often meaningful to the user.

Now we can describe how the "apply" operation works. What we are given is a proof tree, with its internal representation, and the unsolved goal that was selected before. This goal is located by its "access path" in the proof tree. What we call "access path" [13], is the list of moves that lead from the top of the tree to the designated goal. We are also given the rule to apply at this place, in its TYPOL form. Here are the operations to perform:

*i)* Generate the internal representation of the TYPOL rule, including the corresponding list of variables associations.

*ii)* Traverse the proof tree, guided by the access path, to find again the chosen goal. This goal, since it is part of the proof tree, is already in internal form. Its association table for variables is the same as the proof's.

*iii)* Call Prolog unification between this goal and the bottom part (conclusion) of the rule. Normally unification should succeed, since the rule was picked from a set of applicable rules.

*iv)* Replace in the unified proof tree the chosen goal by the unified chosen rule. We get a new internal proof tree. However, we must take care of name conflicts between TYPOL variables.

*v)* To do this, combine the two associations tables, which contain the TYPOL variables. If the same TYPOL name denotes different Prolog variables in the resulting proof tree, we generate a new TYPOL identifier.

*vi)* Thus with a new proof tree and associations table, we return.

### 3.1.2. *Saving global stack space*

Often a Prolog predicate is called, which does a lot of work, but whose results (unifications) are not used. For example, a display predicate first builds the external representation of the proof, then sends it to a window, and does even more operations to highlight some given subpart of the proof tree. When Prolog solves this predicate, it keeps all the history of this solving in what is called the "global stack". We know however this is useless, since there is no backtracking point inside. Experience shows it is not enough to write a cut. Instead, to prevent this waste of space, we do the following: whenever the definition of a predicate is of the form:

**predicate(INPUT1,...,INPUTn) :- "Heavy stuff".**

Then replace it by the equivalent definition:

**predicate(INPUT1,...,INPUTn) :- "Heavy stuff", fail.**

**predicate(INPUT1,...,INPUTn).**

The failing removes all the history from the global stack. This is a typical method, well known to prolog programmers (*e.g.* see [9]).

### 3.1.3. *The MENU predicate*

The MENU predicate must ask the user for a command, and return this command to the system. Another requirement is that it should be a repetition point in case of backtracking. For example, if a user calls a command, and then failure and backtracking occur, then the user should be given the chance to call another command. Last requirement is that this predicate must deal with the “undo” command.

The way we execute an undo command is simple. We just make the current evaluation fail, so that a Prolog backtrack begins. That is exactly what we need, since we want to be set back to a previous state. We also must determine where this backtracking stops. We said it should be stopped at the previous choice point. Most often, this means the previous menu. Here follows the sketched implementation of the MENU predicate:

```
menu(CMD) :-
      openlisp,
      is_command(“undo”),!,
      fail.

menu(CMD) :-
      is_command(CMD).

menu(CMD) :-
      menu(CMD).
```

We use to write this also in TYPOL, although the “cut” predicate is not legal in TYPOL, in the following way:

$$\frac{\text{openlisp} \qquad \text{is\_command}<\text{“undo”}> \qquad ! \qquad \text{fail}}{\text{menu}<\textsc{cmd}>}$$

$$\frac{\text{is\_command}<\textsc{cmd}>}{\text{menu}<\textsc{cmd}>}$$

$$\frac{\text{menu}<\textsc{cmd}>}{\text{menu}<\textsc{cmd}>}$$

The “openlisp” predicate opens a Le_Lisp level where the user has the control. When the user calls a command, the predicate “is_command” is created, and the control is returned to TYPOL, because the “openlisp” predicate succeeds.

The first rule ensures that an undo command causes a Prolog failure This is done with the cut-fail combination. For all other cases, the second rule reads the command and gives it as a result. Finally, the third rule sets a choice point. Whenever the program backtracks to this point, this rule calls a new instance of the menu predicate, and we are back to the first rule.

### 3.1.4. *Trace of the proof behavior*

When constructing a proof tree, many messages, errors or comments, appear in a special window. We added two refinements to this message mechanism. First, we add the notion of backwards information, *i.e.* messages to be displayed only in case of backtracking. This is convenient to indicate that some procedure or command has failed. The definition is easy:

```
back_info(IMPORTANCE,TEXT).

back_info(IMPORTANCE,TEXT) :-
          info(IMPORTANCE,TEXT),
          fail.
```

The second refinement is about the “IMPORTANCE” variable in the above program. This is a simple method, by which we give every message a severity level. The user specifies the limit level for the messages that appear. This is easy to define in Prolog.

### 3.1.5. *Levels and garbage collection*

We now present our ideas about what is opening and closing a proof level. Here a level is a loop in which one repeatedly calls operations and tactics. This loop is opened by calling a predicate, "go_work", which eventually calls a tactic itself. The "go_work" predicate takes care of forbidding "undo" operations beyond the opening of the level. The normal way of closing a level is exiting this "go_work" predicate. After exiting, a fail is done to recover room in the global stack. This induced backtrack is stopped immediately, and the proof goes on.

Complications occur when dealing with garbage collection. When the user calls for garbage collection, or closes the current level, the system must forget all choice points since the opening of the current level. This is still done with a "fail" predicate just after the predicate that opens the level, "go_work".

Thus, if the user wants to exit the current level, Prolog exits the "go_work" predicate (after performing a cut to remove choice points). It then calls a fail to free memory space, and the level is exited.

If the user desires to garbage collect while inside a level, then the system essentially places a marker (is_loop_on) and closes the level thus freeing space. Then the level is reopened according to the marker. The necessary Prolog rules for handling levels are as follows:

```
main_loop :-
        erase_loop_on,
        go_work,
        fail.

main_loop :-
        is_loop_on, !,
        main_loop.

main_loop.

go_work :-
        last_state(S), repeat, back_info(0,"No more undoing"),
        tactic(T), typol_tactic(S,T), !.
```

### 3.1.6. *A time optimization for "expand"*

For each expand command, we are always going to search the same set of TYPOL rules. To avoid reading this TYPOL file every time, we read it only once, then store it into the Prolog memory. Reading the TYPOL rules from the Prolog memory is fast and worth the cost in space.

### 3.1.7. *Problems of reverse compilation*

When executing a goal in Prolog, we said that we translated the TYPOL goal into Prolog, then the resulting instantiated Prolog goal back into TYPOL. However this translation is not a one-to-one application, and sometimes a given Prolog term may be the compiled version of many TYPOL terms. Therefore it is clear that examples can always be found where the reverse translation finds a possible TYPOL origin term, and this term is not the one the user hoped for.

## 3.2. **Lisp Techniques**

### 3.2.1. *Remembering a state of the display*

In Centaur windows, there is no cursor like in a text editor. There is instead a notion of current subtree, displayed for instance in bold face, or in reverse video. Thus if one wants to remember the state of the display, one also has to remember the selected subtree. We do this again with the notion of “access paths” (For more details see [13]).

### 3.2.2. *Remembering all the previous states of the display*

We would like to remember all the previous states of the display so that in backtracking we can easily update the display to a previous state. Since Prolog considers the display as a side effect, we build display routines that undo their effects in case of backtracking. Of course we then need to maintain a stack of all previously displayed states. We provide Lisp functions to push and pop display states from this stack. These functions are called from the TYPOL level, when some important display operation occurs, like the printing of a new state, the start of a backtracking, or the display of a previous state before giving the control to the user. Let’s give an example. Here is a drawing representing a possible sequence of commands:

Now, on the same skeleton, here are the display operations performed at the same time:

This stack of displays must be manipulated to maintain consistency with other mechanisms such as levels and checkpointing. This is done as folllows

- When the user opens a level a mark is pushed into the stack. This indicates that the previous state does not belong to the same level and thus cannot be popped.
- When the user closes a level, since all the operations done inside this level are forgotten, everything is popped from the stack upto and including the mark.
- When a “checkpoint” (garbage collection) is done all the removed choice points correspond to displays. These displays are also removed from the stack. But the mark remains on the stack.
- When the user calls the “simplify” command the previous state, *before* the modification, must be stored. However this previous state is no longer on the screen. Thus every display is “photographed” before the user touches it. If the contains of the window are modified, then the picture is pushed on top of the stack.

## 3.3. Efficiency

At first sight, our system may seem a little slow. The timings given are measured on an SM 90, with a 68010 chip. These times would be about four times faster on a Sun 3/75.

Note that the times depend on whether or not the resulting state is displayed. Inside tactics, for example, states are usually not displayed. The time spent inside a display only depends on the CENTAUR system and not on our program. Usually, the display of a twenty-line proof tree takes about five seconds. This time includes the conversion from the TYPOL term to its Lisp representation. The following times do not include display time. These are real times, since Cpu measures are not very reliable in C-Prolog.

| **Operation** | **Approximate real time** |
|---|---|
| **"Apply"** | **2 s.** |
| **"Expand"** | **Dependant on the size of TYPOL rule set:**<br>**40 rules: 5 s.**<br>**12 rules: 2 s.** |
| **"Undo"** | **< 1 s.** |

Note that disregarding display time we feel that little can be gained on the Prolog side. Expanding could be speeded up with a more suitable way of storing the TYPOL rules .

Instead of storing the entire proof tree we could store the "names" of the rules applied and the order in which they were applied (This idea was suggested by J. and Th. Despeyroux). The unsolved goals on top of the proof tree should be stored explicitely. It is clear that memory space is spared. Conversely, there is an extra cost in time: when displaying the proof tree, names have to be replaced by the actual rules, and $\alpha$-conversions should be performed for identical identifiers in different rules.

## 4. A SHORT USER'S MANUAL

### 4.1. Getting Started

We assume some familiarity with the CENTAUR system, especially with its mouse and menus facilities, the syntax directed manipulations of programs, and the TYPOL sub-system. We suppose we have called the CENTAUR system and edited some TYPOL file in a window. Now, we want to do some proofs about, or at least using, this TYPOL program.

The first thing to do is to load our proof system, which consists in a Le_Lisp file and a Prolog file. This is done by typing the Lisp command:

```
(loadfile "proof.ll" t)
```

Then, the system is loaded, and a new item appears in the menu associated with the TYPOL program. This item is called "prove". Selecting this item starts the proof, by opening the first level, numbered 0. The Lisp prompt sign is modified to "*n*>", where *n* is the current level number. The current position in the TYPOL file is used to find a goal to prove. If something is found there then it is the initial value of the proof tree. Otherwise this initial value is void, and we get the message:

**Nothing to prove yet**

Since the system starts with the "user" tactic, it must display its starting state. The user must give the positions (with the mouse) of the two windows "proof" and "rules". The current window is "proof". It is a TYPOL window because proofs are written with the TYPOL syntax. Two new submenus appearon the pop-up menu. These are called "proof" and "tactics". Inside these submenus, we can find all the basic commands that were presented in the second section. The window which is not current, here "rules", is displayed with a grey shade. This means it is disabled, and we should not touch its contents or call a menu in it.

Usually, we shall have to define by hand the starting state of the proof tree. Suppose this proof is in a file. Then we do the following:

*i)* Load this file into the "proof" window, using the "read" option in the menu.

*ii)* It is VERY important to verify that the name of the proof window is still "proof". If not, reset it to "proof", using the "rename" option of the menu.

*iii)* Inform the system that we have changed the current proof using the "modify" option of the "proof" submenu.

## 4.2. Hand Proof Editing

We are now going to list all the available operations. For each operation, we indicate the name of the corresponding button to click on, followed by the submenu that contains it. Then we indicate in which window(s) this operation is available. This means that the operation is only available when that window is the current one (non-greyed). For each operation, we also indicate whether it is considered as a basic operation, a control primitive, or a tactic. We follow with the message that the user gets after clicking the button, and the message when this operation is undone. (Say the user clicked on "Undo" just after).

$\boxed{\textbf{Expand}}$ from $\boxed{\textbf{Proof}}$ submenu of proof window. basic operation

$\longrightarrow$**Looking for rules for expanding**
$\longleftarrow$**Undoing an expand**

Once selected, the system takes control. It searches the current proof for an unsolved goal. The search starts from the currently designated part of the proof tree (*i.e.* the highlighted subtree). If no goal is found, then the operation fails with the message:

**No goal around**

and the user returns to menu mode. When a goal is found, then the system searches the reference TYPOL program for all applicable rules. It builds a (possibly empty) set with these rules. The final step is the display of this set into the "rules" window. The rules window becomes the current one, while the proof window is shaded. The control returns to the user.

$\boxed{\textbf{Apply}}$ from $\boxed{\textbf{Proof}}$ submenu of rules window. basic operation

$\longrightarrow$**Applying a rule**
$\longleftarrow$**Undoing an apply**

The system takes control and looks for a rule in the "rules" window. The set of rules is searched starting from the currently highlighted point. If no rule can be found, then we get the message:

**No rule around**

and the user returns to menu mode. Otherwise the rule is applied to the previously selected goal in the proof tree. If this cannot be done, say because of a false hand-manipulation in the set of rules, then the message:

**This rule does not apply to the current goal**

is printed, and the user is back to the previous menu. Otherwise the new proof tree is displayed in the "proof" window, which becomes the current window. The highlighted subtree is on the part of the proof tree which is the copy of the applied rule. The control returns to the user.

$\boxed{\textbf{Execute}}$ from $\boxed{\textbf{Proof}}$ submenu of proof window. basic operation

$\longrightarrow$**Solving directly in PROLOG**
$\longleftarrow$**Undoing an execution**

The system looks for an unsolved goal, as above. The search starts from the currently highlighted subtree, then goes into the right brothers, etc. The error message is the same as above when no goal is found. The goal is solved directly by calling Prolog. This implies its definition must be present at this moment in the Prolog memory. If the Prolog call fails then the message:

**PROLOG solving fails**

is printed, and the user returns to menu mode. Otherwise a new proof tree is found that is displayed in the "proof" window. The recently solved goal is the new highlighted subtree. The control returns to the user.

$\boxed{\textbf{Instantiate}}$ from $\boxed{\textbf{Proof}}$ submenu of proof window. basic operation

$\longrightarrow$**Instantiating a variable**
$\longleftarrow$**Undoing an instantiation**

*This operation is not yet implemented.* The system looks for a TYPOL variable in the proof tree. The search starts from the highlighted subtree. If no variable is found then we get the following error message:

**No variable to instantiate**

and the user returns to menu mode. If a variable is found then the user is asked to write a TYPOL term to instantiate it. If this term is legal to TYPOL, then all occurrences of the variable in the proof are replaced. The new proof tree is displayed in the "proof" window. The highlighted subtree is as before. The control returns to the user.

$\boxed{\textbf{Simplify}}$ from $\boxed{\textbf{Proof}}$ submenu of proof window. basic operation

$\longrightarrow$**Simplifying the proof tree**
$\longleftarrow$**Undoing a simplification**

The system produces the TYPOL rule consisting of the bottom predicate of the proof, with the list of all unsolved goals in the proof as numerator. The new highlighted subtree corresponds to the goal which was "closest" to the proof tree's highlighted subtree. The new proof tree is displayed in the "proof" window, which remains the current window. The control returns to the user.

$\boxed{\textbf{Modify}}$ from $\boxed{\textbf{Proof}}$ submenu of proof (*resp.* rules) window. basic operation

$\longrightarrow$**Taking in account the modifications to the proof tree (*resp.* rules)**
$\longleftarrow$**Undoing a modification**

The system reads from the current window the new value of the displayed object (proof tree or set of rules) and makes it the new current value. An undo command would return the user to the state *before* the modification, *i.e.* after the end of the previous command. The control returns to the user

$\boxed{\textbf{Undo}}$ from $\boxed{\textbf{Proof}}$ submenu of proof (*resp.* rules) window. control primitive

$\longrightarrow$*No message*
$\longleftarrow$*No message*

This command returns the user to the previous time when he was offered some choice (or when the tactic made a choice). It displays this state exactly, with the same current window and the same highlighted subtree. The control then returns to the user (or the tactic). Another undo returns the user to an even earlier state, one state further back. That is, an "Undo" does not undo an "Undo".

It is not possible to undo when there is no previous command, or when this previous command is an "open level" command. An "Undo" doesn't enter inside a completed (closed) level. All the operations are undone with one "Undo" command and the user sees the message:

**Undoing a complex sequence of actions**

$\boxed{\textbf{Open level}}$ from $\boxed{\textbf{Tactics}}$ submenu of proof window. control tactic

$\longrightarrow$**Opening a new level**
$\longleftarrow$*No message*

A new level is opened, and nothing changes in the two displayed windows, except that the rules window contents are erased. This is only to signify that the current state is now the starting state (of the current level). Thus it is useless to know the way it was obtained. The Le_Lisp prompt is changed. For instance, if it was "3>" then it becomes "4>" indicating that we are on the $4^{\text{th}}$ level. The level numbered 0 is the starting level.

It is forbidden to undo an "Open level" command. One must close the level first, then undo the level as a block.

$\boxed{\textbf{Close level}}$ from $\boxed{\textbf{Tactics}}$ submenu of proof window. control tactic

$\longrightarrow$**Closing the current level**
$\longleftarrow$**Undoing some complex sequence of actions**

The current level is closed, and the system goes back to the previous level. However the results of the level are not lost. Nothing is changed in the displayed windows, except that the rules window is erased, becuse the history of the displayed proof inside the closed level is forgotten now. When a level is closed, all the actions performed in it are regarded as a single, atomic, complex action. The results of this action, the new state, is still the current state.

The Le_Lisp prompt is set back to its previous value, before the level was opened. When the current level is the starting level (prompt is 0>), closing the level results in stopping the proof system itself, with the message:

**Stopping proof**

If a level is undone, *i.e.* when the user clicks on "undo", or the system backtracks, then all the operations are undone. The user returns to the state where he clicked on the "Open level" button.

$\boxed{\textbf{Checkpoint}}$ from $\boxed{\textbf{Tactics}}$ submenu of proof window. control tactic

**⟶Crunching all from the previous checkpoint**
**⟵No undoing inside this checkpoint level**

When this command is selected, nothing is changed in the display except that the "rules" window is erased. The system then garbage collects all the choice points between the current state and the previous opening level, or the previous call to the "checkpoint" command. The "rules" window is erased for the same reason as in the "Open level" case. The history leading to the current state becomes irrelevant, since it is forbidden to backtrack upon it.In other words, the current state becomes the new starting state of the current level.

$\boxed{\textbf{Helped user}}$ from $\boxed{\textbf{Tactics}}$ submenu of proof window. example tactic

**⟶New tactic: Helped user**
**⟵Previous tactic again**

This is only a short and simple tactic that we introduce as an example. Like all tactics, it can be called only when the "proof" window is active (non-greyed). When the user clicks on it, nothing at all happens to the display. The current tactic is now "Helped user" and not "User". The difference is that, after an expand, if only one rule is available, then it is applied directly. However, if then the user calls "undo", then the one-rule set is displayed. The user can modify the set to add new rules before calling "apply".

$\boxed{\textbf{User}}$ from $\boxed{\textbf{Tactics}}$ submenu of proof window. basic tactic

**⟶New tactic: User**
**⟵Previous tactic again**

This is the basic tactic, the one which is called at the very beginning. When the user clicks on this button, the tactic "User" is again the current tactic, whatever the previous tactic was.

## 4.3. A Small Example

Let's suppose we are editing a TYPOL program which defines operations on lists. The rules appear in the "rules" window during the proof. When we click on "prove", say no goal to prove is found. So we load a file containing:

$$(a) \qquad \overset{\text{length}}{\vdash} \ \text{LIST} : \text{N}$$

Next we rename the "proof" window, click on "Modify". We can start now the actual proof building. We start by opening a level, indicating that we are starting a proof, and this proof may be considered as a whole. Thus we click on the "Open level" button. Say we want to know about the length of every list matching list[A, B|L]. This term denotes a list starting with two elements A and B, and continuing with any list L. Let's expand our current proof. The system shows us two applicable rules:

$$(\text{L}_1) \qquad \overset{\text{length}}{\vdash} \ \text{list}[\,] : 0$$

$$(\text{L}_2) \qquad \frac{\overset{\text{length}}{\vdash} \ \text{L} : \text{N} \qquad \text{plus}<\text{N}, 1, \text{N}_1>}{\overset{\text{length}}{\vdash} \ \text{list}[\text{A}|\text{L}] : \text{N}_1}$$

We select the second rule ($L_2$) by clicking somewhere in it, then click on the "Apply" button. Then the system prints the new proof tree:

$$(b) \qquad \frac{\overset{\text{length}}{\vdash}\ \textsc{l}:\textsc{n} \qquad \text{plus}<\textsc{n},1,\textsc{n}_1>}{\overset{\text{length}}{\vdash}\ \text{list}[\textsc{a}|\textsc{l}]:\textsc{n}_1}$$

We select the first unsolved goal and ask for "expand". The rules ($L_1$) and ($L_2$) are displayed again. Suppose we choose the first one this time. The new proof tree is:

$$(c) \qquad \frac{\dfrac{}{\overset{\text{length}}{\vdash}\ \text{list}[\,]:0} \qquad \text{plus}<0,1,\textsc{n}_1\ >}{\overset{\text{length}}{\vdash}\ \text{list}[\textsc{a}]:\textsc{n}_1}$$

We realize this is not what we are interested in. Thus we click on "undo", and we are back to the previous choice of what rule to apply. Selecting the second one this time and then clicking on "apply" we get the following proof tree:

$$(d) \qquad \frac{\dfrac{\overset{\text{length}}{\vdash}\ \textsc{l}:\textsc{n} \qquad \text{plus}<\textsc{n},1,\textsc{n}_2>}{\overset{\text{length}}{\vdash}\ \text{list}[\textsc{a}_1|\textsc{l}]:\textsc{n}_2} \qquad \text{plus}<\textsc{n}_2,1,\textsc{n}_1\ >}{\overset{\text{length}}{\vdash}\ \text{list}[\textsc{a},\textsc{a}_1|\textsc{l}]:\textsc{n}_1}$$

If we simplify we get:

$$(e) \qquad \frac{\overset{\text{length}}{\vdash}\ \textsc{l}:\textsc{n} \qquad \text{plus}<\textsc{n},1,\textsc{n}_2> \qquad \text{plus}<\textsc{n}_2,1,\textsc{n}_1>}{\overset{\text{length}}{\vdash}\ \text{list}[\textsc{a},\textsc{a}_1|\textsc{l}]:\textsc{n}_1}$$

This states that if the length of $\textsc{l}$ is $\textsc{n}$ then the length of $\text{list}[\textsc{a},\textsc{a}_1|\textsc{l}]$ is obtained by incrementing $\textsc{n}$ twice. Since we consider our proof finished, we click on "Close level" and perhaps "checkpoint" (to recover memory space). The system is ready to perform other operations. To stop, we just click one more time on "Close level".

### 4.4. How to Write Tactics

Tactics are written with TYPOL rules themselves. As seen in section 2.3 with the "User" tactic, these rules are to be read in a procedural way. They indicate the sequence of actions to be performed. From this point of view, the choice of TYPOL is disputable. However having one language is an advantage for the user. While the user can define others, here are the basic predicates predefined to build tactics:

| | |
|---|---|
| $\sigma_1 \overset{\text{command}}{\vdash} \textsc{cmd}:\sigma_2$ | To call the basic operation CMD. Since a command modifies the state, $\sigma_1$ and $\sigma_2$ denote the input and output "state" of the proof. |
| $\sigma_1 \overset{\text{next\_goal}}{\vdash} \sigma_2$ | The state $\sigma_2$ is the result of looking into $\sigma_1$ for the next goal to solve. The search, as usual, starts from the current highlighted subtree in the proof part of $\sigma_1$. When backtracking occurs, this predicate gives the next goal found, and so on. It fails when no goal remains. |
| $\sigma_1 \overset{\text{find\_goal}}{\vdash} \sigma_2$ | The state $\sigma_2$ is the result of looking into $\sigma_1$ for a goal to solve. The search starts from the current highlighted subtree in $\sigma_1$. When backtracking occurs, this predicate just fails. This predicate can be defined in Prolog by: $\mathbf{next\_goal(S1, S2)},!$. |

| | |
|---|---|
| $\sigma_1 \overset{\text{next\_rule}}{\vdash} \sigma_2$ | The state $\sigma_2$ is the result of looking into $\sigma_1$ for the next rule to apply. The search, as usual, starts from the current highlighted subtree in the rules part of $\sigma_1$. When backtracking occurs, this predicate gives the next rule found, and so on. It fails when no rule remains. |
| $\sigma_1 \overset{\text{is\_state}}{\vdash}$ STATE | This unifies the variable STATE with "proof" or "rules", according to the current state of the proof tree construction. |
| $\sigma_1 \overset{\text{number\_rules}}{\vdash}$ N | This unifies N with the number of rules in the "rules" part of $\sigma_1$. |
| menu<$\sigma_1 \rightarrow$ CMD, $\sigma_2$> | The "menu" predicate gives back the command CMD that the user selected, along with a new state $\sigma_2$. This state includes the user manipulations on $\sigma_1$. |
| glimpse<$\sigma_1 \rightarrow \sigma_2$> | This displays the current state of the proof. Since $\sigma_1$ contains information about what is displayed, it is modified to $\sigma_2$. |
| display<$\sigma_1 \rightarrow \sigma_2$> | This is a special case of "glimpse". One must call "display" instead of "glimpse" when one intends to call "menu" after. This is needed by the manipulations in the stack of previously displayed states. |
| info<N, T><br>back_info<N, T> | To print message T on the screen, with importance N. "back_info" prints on backtrack. Definitions are in section 3.1.4. |
| text_level<N> | This defines the messages that are to be printed or not. Every message with "IMPORTANCE" level above a fixed limit are not displayed. This predicate sets the value of this limit to N. The starting value of N is 100. |
| $\sigma \overset{\text{next\_tactic}}{\vdash}$ TAC | This is the way a tactic may call another tactic TAC. It updates the global name of the current tactic to TAC. It can be defined by the following sequence of predicates (*cf* section 2.3):<br>$\sigma \overset{\text{command}}{\vdash}$ TAC : $\sigma_2$ tactic<$\tau$> $\sigma_2 \overset{\text{tactic}}{\vdash} \tau$ |
| $\sigma \overset{\text{next\_tactic}}{\vdash}$"Open level"(TAC) | This is a special case of the previous case. This calls for a new level, in which the starting tactic will be TAC. If one just calls "Open level", then the starting tactic in the level is the current one. |
| $\sigma \overset{\text{next\_tactic}}{\vdash}$"Close level"(TAC) | Same as "Open level". However, for implementations reasons, one must always make sure that this predicate is called at the end of the rule. Moreover, when this predicate is solved, the next predicate to solve in the standard Prolog order must be the "!" at the end of GO_WORK. (*cf* section 3.1.5) |
| $\sigma \overset{\text{next\_tactic}}{\vdash}$"Checkpoint"(TAC) | Same remarks as for "Close level". This predicate garbage collects everything since the opening of the current level. |

It is recomended that tactics should call tactics always at the end of the rules that define them. However, for the applications we have already written, we felt no need for putting a tactic call in the middle of a tactic definition. Such calls are always put as the last predicate to solve. We finish this section with an small example tactic.

## 4.5. **A Small Tactic**

Here we define the ordinary depth-first solving of a predicate. Our tactic opens a level first, then stops on the first solution found depth first, and displays it. If the user then chooses "Undo", this tactic gives the next solution in depth-first order and so on. We define this tactic as follows:

$$\frac{\overset{\text{is\_state}}{\sigma \vdash \text{“proof”}} \qquad \overset{\text{next\_tactic}}{\sigma \vdash \text{“Open level”(“df0”)}}}{\overset{\text{tactic}}{\sigma \vdash \text{“Depth first”}}}$$

$$\frac{\overset{\text{next\_tactic}}{\sigma \vdash \text{“df1”}}}{\overset{\text{tactic}}{\sigma \vdash \text{“df0”}}}$$

$$\frac{\text{info}{<}3, \text{“Depth first solving fails”}{>} \qquad \overset{\text{next\_tactic}}{\sigma \vdash \text{“Close level”(“User”)}}}{\overset{\text{tactic}}{\sigma \vdash \text{“df0”}}}$$

$$\frac{\overset{\text{find\_goal}}{\sigma_1 \vdash \sigma_2} \qquad \overset{\text{command}}{\sigma_2 \vdash \text{“Expand”} : \sigma_3} \qquad \overset{\text{next\_tactic}}{\sigma_3 \vdash \text{“df2”}}}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“df1”}}}$$

$$\frac{\text{info}{<}3, \text{“All goals are solved”}{>} \qquad \overset{\text{next\_tactic}}{\sigma \vdash \text{“User”}}}{\overset{\text{tactic}}{\sigma \vdash \text{“df1”}}}$$

$$\frac{\overset{\text{next\_rule}}{\sigma_1 \vdash \sigma_2} \qquad \overset{\text{command}}{\sigma_2 \vdash \text{“Apply”} : \sigma_3} \qquad \overset{\text{next\_tactic}}{\sigma_3 \vdash \text{“df1”}}}{\overset{\text{tactic}}{\sigma_1 \vdash \text{“df2”}}}$$

This tactic has been tested. When no other solution exists, a message is printed. The system then closes the current level. The effect is that an extra “Undo” undoes all the depth first solving. If the user wants at some stage to forget all the remaining solutions, he may click either on “Checkpoint” or “Close level”. Thus, this tactic exactly simulates the behavior of a simple Prolog.

## 5. APPLICATIONS

### 5.1. Typol Evaluation

Evaluation of TYPOL predicates is one of the goals from which this work originated. In fact, the main part of its implementation lies in the tactic for depth-first solving (as seen in section 4.5). The only additional rules needed are for solving TYPOL goals which execute directly in Prolog.

We find the behavior of “undo” interesting. Thus one can repeatedly see the different solutions until none exists. We like to be able to decide between:

*i)* Getting the next solution, or

*ii)* Stopping on one solution, without ever wanting to see the others.

This is similar to C-Prolog’s “;” facility, which gives all successive solutions, until “return” is typed.

### 5.2. Others Tactics of Evaluation-Resolution

It is interesting to try other tactics of evaluation (or resolution, depending whether one thinks about TYPOL or Prolog). For example, one tactic could implement a breadth-first algorithm. Other interesting tactics could be found by trying to introduce more nondeterminism to TYPOL, say for interpreting parallel languages.

We may want to freeze goals until some of their arguments are completely instantiated (no free variables) as in Prolog-II [2]. Another criterion for freezing goals could be if one of its arguments gets instantiated more precisely because of the solving. This is what happens in MU-Prolog for instance. All these tactics seem quick to implement in our system. (Since the resulting interpreter is slow, this should be viewed as a test of the algorithm, a prototype, or a demonstration tool.) Another tactic could aim at helping somebody who wants to execute a TYPOL goal or prove a TYPOL theorem (rule). The idea of this tactic is to refuse all commitments, as long as other things can be done. It can easily be related to the two previous evaluation strategies. This can be done as follows:

*i)* Select a goal in the current proof tree. Say we use the ordinary preorder search, although another tactic yields yet another interpreter.

*ii)* Expand this goal. If there is more than one applicable rule then do not choose between these rules, simply go back to step *(i)*

*iii)* If there is only one rule then apply it. Start again with the resulting proof tree.

The underlying idea is that some goal, which can be expanded in two ways at some time, may be instantiated through the expansion of another goal. This instantiation may decrease the number of applicable rules to it. If there remains only one rule then it becomes safe to expand the corresponding goal. One can consider this tactic safe and can use it as much as possible. It could be combined with other tactics that choose between rules, in order to continue with the proof. The call to these less "clean" tactics can be delayed to the very last moment.

## 5.3. Theorem Proving

Since we are not specialists of the underlying theories of theorem proving, we shall explain the use of our system for theorem proving in more common language.

Let's show a trivial example of theorem proving. Suppose that we have TYPOL axioms about proving predicates in an environment listing all known true facts. What we want to prove is for any environment $\rho$, for all $\mathrm{A}$ and $\mathrm{B}$, the predicate:

$$\mathrm{A} \Longrightarrow ((\mathrm{A} \Longrightarrow \mathrm{B}) \Longrightarrow \mathrm{B})$$

This corresponds to the following TYPOL goal to prove:

$$\rho \vdash \mathrm{A} \Longrightarrow ((\mathrm{A} \Longrightarrow \mathrm{B}) \Longrightarrow \mathrm{B})$$

Say we have, among others, the following TYPOL rules:

(*lookup*)
$$\frac{\mathrm{P} \in \rho}{\rho \vdash \mathrm{P}}$$

(*introduction*)
$$\frac{\rho_2 = \rho_1 \cup \mathrm{X} \qquad \rho_2 \vdash \mathrm{Y}}{\rho_1 \vdash \mathrm{X} \Longrightarrow \mathrm{Y}}$$

(*modus ponens*)
$$\frac{\rho_1 \vdash \mathrm{X} \qquad \rho_2 \vdash \mathrm{X} \Longrightarrow \mathrm{Y} \qquad \rho_1 \cup \rho_2 \subset \rho}{\rho \vdash \mathrm{Y}}$$

To prove the implication we have the choice between a rule which just searches for the entire term in $\rho$, (*lookup*), and the (*introduction*) rule. Obviously, we don't want the (*lookup*) rule, because it is environment specific. Conversely (*introduction*) suits us very well. We apply the introduction rule again, and we have the following proof tree (state of the growing proof):

$$\cfrac{\cfrac{(\mathrm{A} \Longrightarrow \mathrm{B}) \bullet \mathrm{A} \bullet \rho \vdash \mathrm{B}}{\mathrm{A} \bullet \rho \vdash (\mathrm{A} \Longrightarrow \mathrm{B}) \Longrightarrow \mathrm{B}}}{\rho \vdash \mathrm{A} \Longrightarrow ((\mathrm{A} \Longrightarrow \mathrm{B}) \Longrightarrow \mathrm{B})}$$

Next, we choose the (*modus ponens*) rule:

$$\frac{\rho_1 \vdash \textsc{x} \qquad \rho_2 \vdash \textsc{x} \Longrightarrow \textsc{y} \qquad \rho_1 \cup \rho_2 \subset \rho}{\rho \vdash \textsc{y}}$$

which means that if $\textsc{x}$ holds in $\rho_1$, and if $\textsc{x} \Longrightarrow \textsc{y}$ holds in $\rho_2$, then $\textsc{y}$ holds in every environment $\rho$ which contains both $\rho_1$ and $\rho_2$.

The next goal that we choose to expand is the third one, about the inclusion between environments. This is because it is the only goal in which some arguments are sufficiently instantiated (known). This could be done by a tactic simulating a MU-Prolog like strategy. The rule about environments that we choose to apply here is trivial:

$$\rho \cup \rho \subset \rho$$

The resulting proof tree is now:

$$\cfrac{\cfrac{\cfrac{(\textsc{a} \Longrightarrow \textsc{b}) \bullet \textsc{a} \bullet \rho \vdash \textsc{x} \quad (\textsc{a} \Longrightarrow \textsc{b}) \bullet \textsc{a} \bullet \rho \vdash \textsc{x} \Longrightarrow \textsc{b} \quad \overline{\vdots}}{(\textsc{a} \Longrightarrow \textsc{b}) \bullet \textsc{a} \bullet \rho \vdash \textsc{b}}}{\textsc{a} \bullet \rho \vdash (\textsc{a} \Longrightarrow \textsc{b}) \Longrightarrow \textsc{b}}}{\rho \vdash \textsc{a} \Longrightarrow ((\textsc{a} \Longrightarrow \textsc{b}) \Longrightarrow \textsc{b}}$$

The next goal we solve is the second one, again because we don't know enough about the instantiation of the first goal. This of course unifies $\textsc{x}$ with $\textsc{a}$. Now, the first goal is successfully solved, and the theorem is proved.

## 5.4. Structural Induction

Although it is not possible to fully automatize the process of building proofs of theorems, We would like to automatize as much as possible. An outstanding example is structural induction. We plan to write a tactic which helps the user in finding proofs by structural induction. The method we use is called induction on the length of the proof. (This is also used in [5]). The tactic would work in two phases. Let's illustrate this on an example. Say we want to prove the following theorem, in a context defining the usual operations on lists and natural numbers:

$$\frac{\text{length}{<}\textsc{a}, \textsc{la}{>} \qquad \text{length}{<}\textsc{b}, \textsc{lb}{>} \qquad \text{length}{<}\textsc{c}, \textsc{lc}{>} \qquad \text{concat}{<}\textsc{a}, \textsc{b}, \textsc{c}{>}}{\text{plus}{<}\textsc{la}, \textsc{lb}, \textsc{lc}{>}}$$

The first part of the tactic is to expands the hypothesis predicates in all possible ways. Of course, there are infinitely many ways, so what we do is to expand this list of hypotheses to a given depth in all possible ways. The obtained expansions should represent all the possible complete expansions. The user helps determine the depth of the expansion. The corresponding unifications are propagated to the conclusion predicate.

In our example, it is sufficient to expand the first, third and fourth predicates in the following two manners:

$$\frac{\overline{\text{length}{<}[\,], 0{>}} \qquad \text{length}{<}\textsc{b}, \textsc{lb}{>} \qquad \text{length}{<}\textsc{b}, \textsc{lc}{>} \qquad \overline{\text{concat}{<}[\,], \textsc{b}, \textsc{b}{>}}}{\text{plus}{<}0, \textsc{lb}, \textsc{lc}{>}}$$

$$\frac{\dfrac{\text{length}{<}\textsc{a}, \textsc{la}{>}}{\text{length}{<}[\textsc{x}|\textsc{a}], s(\textsc{la}){>}} \qquad \text{length}{<}\textsc{b}, \textsc{lb}{>} \qquad \dfrac{\text{length}{<}\textsc{c}, \textsc{lc}{>}}{\text{length}{<}[\textsc{x}|\textsc{c}], s(\textsc{lc}){>}} \qquad \dfrac{\text{concat}{<}\textsc{a},\textsc{b},\textsc{c}{>}}{\text{concat}{<}[\textsc{x}|\textsc{a}], \textsc{b}, [\textsc{x}|\textsc{c}]{>}}}{\text{plus}{<}s(\textsc{la}), \textsc{lb}, s(\textsc{lc}){>}}$$

For each expansion, the second phase is prove the conclusion using:

*i)* The known axioms, which are the given TYPOL rules defining “plus”, “concat”, “length”. These axioms must be used at least once because one should not use the induction hypothesis at once.

*ii)* At some time, if necessary, the theorem we want to prove. This is the induction hypothesis.

*iii)* Then the unsolved predicates from the top of the expanded proof tree (if there are any), considered as axioms now. The unsolved predicates are the ones with no horizontal rule above them.

In our example, for the first case (expansion), it is clear that the conclusion will be proved using only step *(i)*. In the second case, we start expanding by the only applicable rule for “plus”, namely:

$$\frac{\text{plus}{<}\text{X}, \text{Y}, \text{Z}{>}}{\text{plus}{<}s(\text{X}), \text{Y}, s(\text{Z}){>}}$$

Now we apply the theorem we are trying to prove. This is the inductive step. It is clear that the hypotheses of the inductive step are the four unsolved predicates from the expanded proof tree. Thus the theorem is proved.

## 5.5. **Partial Evaluation**

Partial evaluation is another application for our system. This deals with evaluating (executing) a program when some of inputs are not known. Say we separate the input values in two parts $P_1$ and $P_2$. The result of partial evaluation of the program when only $P_1$ is known is a new program whose input is only $P_2$. We present this fully in our paper [14]. Here we sketch the ideas behind this application.

This is our most complex implemented tactic thus far. It is a dozen rules long, but uses most of the power of the system. The tactic defines what should be a partial evaluation of a TYPOL question. This is only *our* vision of how a partial evaluation could be performed. By using well-known methods of partial evaluation and mixed computation, we “compile” ASPLE (small sub-Pascal) files without writing a compiler.

The basic idea, originating from Ershov [8], and Futamura [10], is to partially evaluate the (ASPLE) interpreter when only one of the interpreter’s inputs is known, that being the program to interpret. The input values to the program itself are not known. What we obtain in turn is a TYPOL program: that is an ASPLE interpreter, specialized to the given ASPLE program. This resulting TYPOL program takes as input the input to the program and gives the results we would obtain by interpreting this program. This means that this TYPOL program we obtain is the *compiled* version of the original ASPLE program.

For example consider factorial function written in ASPLE:

```
begin
   int x, y, z;
   input x int;
   y:=1;
   z:=1;
   if (deref x <> 0)
     then while (deref z <> deref x) do
                  z:=deref z +1;
                  y:=deref y × deref z;
              end;
   fi;
   output deref y int;
end
```

The result of partial evaluation is the following short set of TYPOL rules:

$$(F_1)\qquad \frac{\textbf{read}{<}a, \mathrm{int}{>} \qquad a \vdash \mathrm{t}_2 : b, c \qquad \textbf{write}{<}c{>}}{\vdash \mathrm{t}_1 : \mathrm{in}[a], \mathrm{out}[c]}$$

$$(F_2)\qquad \frac{\textbf{different}{<}a, 0, \text{“}\mathit{true}\text{”}{>} \qquad a, 1, 1 \vdash \mathrm{t}_3 : c}{a \vdash \mathrm{t}_2 : a, c}$$

$$(F_3)\qquad 0 \vdash \mathrm{t}_2 : 1, 1$$

$$(F_4)\qquad \frac{\textbf{different}{<}c, a, \text{“}\mathit{true}\text{”}{>} \qquad \textbf{add}{<}c, 1, c_1{>} \qquad \textbf{times}{<}b, c_1, b_1{>} \qquad a, b_1, c_1 \vdash \mathrm{t}_3 : d}{a, b, c \vdash \mathrm{t}_3 : d}$$

$$(F_5)\qquad a, b, a \vdash \mathrm{t}_3 : b$$

where rules ($F_2$) and ($F_3$) deal with the **if** subtree, which is named $t_2$, and ($F_4$) and ($F_5$) deal with the **while** subtree ($t_3$). This TYPOL program is in fact a TYPOL implementation of the factorial function. It has the same behavior as ASPLE factorial, but nothing in it depends on ASPLE any longer. Another experiment using ML instead of ASPLE, gave another TYPOL program, very similar to the one above.

## 6. CONCLUSION

This paper presents version 1 of our system. We think it is a useful tool for people building proofs of theorems or writing tactics for evaluation. Thus we explain in detail the basic commands, sketch a user's manual, and present implementation techniques. We briefly present some applications and we hope other applications will follow.

This system is still evolving. Minor changes could be different ways of displaying a proof tree, or of storing it internally. In the next versions, we hope for improvements in speed and better ways of writing tactics. We think it is interesting to study the relationship between this system and other proof editors, theorem provers, systems to drive the execution of inference rules, *e.g.* the Edinburgh LF [12], or the ECRINS project [16].

**ACKNOWLEDGEMENTS**

I would like to thank all the members of the Centaur team, who supported me, gave me many fundamental ideas and urged me to write something about this system. More pragmatic thanks to Janet Incerpi and Gilles Kahn, who read this to improve both my style *and* my ideas.

## REFERENCES

[1] **Bowen, D., Byrd, L., Pereira, L., Warren, D.**, *“C-Prolog user's manual, version 1.5”*, Edited by F.Pereira, SRI International, Menlo Park, California, (February 1984).

[2] **Caneghem, M. Van,** *“Prolog II, Manuel d'utilisation”*, Technical report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, Marseille, (1982).

[3] **Chailloux, J., et al.**, *“Le_Lisp de l'INRIA, Version 15.2, Manuel de référence”*, Reference manual, INRIA, (May 1986).

[4] **Clément, D., Despeyroux, J. and Th., Hascoët, L., Kahn, G.**, *“Natural semantics on the computer”*, Rapport de recherche INRIA, N$^o$ 416, (June 1985).

[5] **Despeyroux, J.**, *"Proof of translation in natural semantics"*, Proceedings of the 1$^{st}$ Symposium on Logic in Computer Science, IEEE, Cambridge, Mass. (June 1986).

[6] **Despeyroux, Th.**, *"TYPOL, a formalism to implement natural semantics"*, Rapport technique INRIA, (1986).

[7] **Despeyroux, Th.,** *"Executable specification of static semantics"*, Semantics of data types, Lecture Notes in Computer Science, Vol. 173, (June 1984).

[8] **Ershov, A.P.**, *"On the partial computation principle"*, Information Processing Letters, 6(2):pp 38–41, (April 1977).

[9] **Fuller, D.A., Abramsky, S.**, *"Mixed computation of Prolog programs"*, DOC 12/87, Dept. of Computing, Imperial College, London, U.K. (June 1987).

[10] **Futamura, Y.**, *"Partial evaluation of computation process. An approach to a compiler-compiler"*, Systems, Computers, Controls, 2(5):pp 45–50, (1971).

[11] **Gordon, M., Milner, R., Wadsworth, C.**, *"Edinburgh LCF: A Mechanized Logic of Computation"*, Lecture Notes in Computer Science, Volume 78, (1979).

[12] **Harper, R., Honsell, F., Plotkin, G.**, *"A Framework for Defining Logics"*, Proceedings of the 2$^{\mathbf{nd}}$ ACM-IEEE Symposium on Logic in Computer Science, Cornell University, (1987).

[13] **Clément, D., Hascoët, L.**, *"How to designate subtrees in CENTAUR"*, ESPRIT 3$^{rd}$ Annual Report, (January 1988).

[14] **Hascoët, L.**, *"Partial Evaluation with Inference Rules"*, Proc. workshop on Partial Evaluation and Mixed Computation, Denmark, Oct 1987. Also New Generation Computing Journal, Vol. 6, N$^{o}s$ 2 & 3, (1988).

[15] **Kahn, G. et al.,** *"CENTAUR. The system"*, INRIA Report N$^{o}$ 777 (December 1987).

[16] **Madelaine, E., de Simone, R.**, *"ECRINS: Un laboratoire de preuve pour les calculs de processus"*, INRIA Research Report n$^{o}$ 672, (1986).

[17] **Naish, L.,** *"MU-Prolog 3.2 reference manual"*, Technical report 85-11, University of Melbourne, (1985).