THEORY MEETS EFFICIENCY: A NEW IMPLEMENTATION FOR PROOF TREES

UNE IMPLEMENTATION DES ARBRES DE PREUVE EFFICACE ET FONDEE THEORIQUEMENT

Laurent Hascoët
INRIA Sophia-Antipolis
September 27, 1989

Abstract

The implementation of a system for manipulating proof trees shows that the time spent on basic manipulations of proof trees and inference rules is critical, as well as the memory space used. This paper describes a new data structure, D-trees, to represent proof trees and inference rules. This structure is designed to optimize the time spent to find inference rules applicable to a subgoal of a proof, as well as applying such an inference rule to the proof tree. It is also designed to consume very small memory space. This implementation is very closely related to *formulas as types* theories, so that it becomes clearer to understand and easier to check.

Keywords: Inference rules, Proof trees, Formulas as Types, Difference lists

Résumé

Dans un système de construction d'arbres de preuve à partir de règles d'inférence, le temps consacré aux opérations fondamentales telles que trouver une règle applicable et l'appliquer, est critique, ainsi que la taille mémoire utilisée. Nous décrivons ici une structure de données pour la représentation des arbres de preuve et des règles d'inférence. Cette représentation a été créee spécialement pour optimiser les opérations de base, ainsi que la place mémoire. Un aspect intéresant est que la nouvelle structure correspond étroitement à la vision des preuves dans la théorie "formules = types". Ainsi, la nouvelle implémentation a deux lectures, l'une opérationnelle et l'autre plus théorique.

Mots Clés: Règles d'inférence, Arbres de preuve, Formules=Types, D-listes

THEORY MEETS EFFICIENCY: A NEW IMPLEMENTATION FOR PROOF TREES

Laurent Hascoët
INRIA Sophia-Antipolis
September 27, 1989

Abstract

The implementation of a system for manipulating proof trees shows that the time spent on basic manipulations of proof trees and inference rules is critical, as well as the memory space used. This paper describes a new data structure, D-trees, to represent proof trees and inference rules. This structure is designed to optimize the time spent to find inference rules applicable to a subgoal of a proof, as well as applying such an inference rule to the proof tree. It is also designed to consume very small memory space. This implementation is very closely related to *formulas as types* theories, so that it becomes clearer to understand and easier to check.

1. INTRODUCTION

This paper describes a new data structure to implement proof trees built with inference rules. The inference rules we use are written in the natural semantics style [1]. This implementation of proof trees is based on a new data structure, designed to speed up the growing and manipulating of proof trees. The coding was actually done in Prolog, but we believe it may be transposed with the same advantages to imperative or functional languages. The implementation gains conciseness and readability thanks to a strong relation with theory.

The context of this work is the CENTAUR system [6], which is a syntax and semantic directed editor that we develop at INRIA in Sophia-Antipolis. Our group, and this CENTAUR system, are involved in an ESPRIT project, called GIPE. We are focusing here on the semantic part of CENTAUR. Formalisms are provided for the user to specify semantic operations on programs, such as type-checking, interpretation or translation. The choice is to define these operations on abstract syntax trees that represent the object programs (after the parsing step). These operations are defined by means of inference rules, in a language called TYPOL [3]. For a very rough understanding, TYPOL rules look like PROLOG clauses, and recursively define traversals of abstract syntax trees,

during which semantics actions are performed. This is called natural semantics. Actually TYPOL rules are compiled into Mu-Prolog clauses for direct execution.

Our work starts by the design of a tactic-driven system to combine TYPOL inference rules into TYPOL proof trees [4]. The system is written in TYPOL, Prolog, and Lisp. Its goals are clear enough:

- To give a semantics of TYPOL rules not relying only on its compiler.
- To simulate TYPOL actual execution, i.e. to build a TYPOL interpreter, for debugging or demonstration.
- To simulate by tactics other execution strategies, such as breadth first search of the resolution tree, delay mechanisms, or partial evaluation [5].
- To build proof trees in the context of some logic defined with inference rules, or to prove theorems in these logics. A structural induction strategy is in project.

Apart from tactics or other higher-level features, this system relies heavily on a few operations, such as selecting an unsolved subgoal of a proof tree, finding all TYPOL rules applicable to it, apply such a rule to grow the proof tree. Display operations and finding the inference rule derived from a proof tree are important operations too.

While speed and memory space concerns are not very important while using the system by hand, they become essential when using tactics. For instance, let us count for one step the interactive commands to look for TYPOL applicable rules and to apply a rule to grow the proof tree further. The **partial evaluation** strategy on an average TYPOL set of rules and a TYPOL initial goal may take up to 800 steps and more. With respect to memory space, the biggest proof tree built by depth first tactic so far was made of about 30 inference rules, and the size of this proof tree becomes critical when it comes to storing it into PROLOG memory. Lastly, such a system easily becomes heavy and unreadable if some thought is not given to some theoretical guideline. This is why we designed a way to represent and manipulate proof trees sparing time and memory space. This paper describes a solution, and shows how a strong link with formulas as types theory makes things clearer and easy to check.

This paper is organized as follows:

- In chapter 2, we specify the problem by showing why a straightforward coding is slow and heavy. We then state the constraints a good implementation should satisfy.
- In chapter 3, we describe the ideas behind the new implementation. In particular, we describe the main new data structure, which we call **D-trees**.
- In chapter 4, we show how simple (hopefully) and fast it becomes to build proof trees inside the new implementation.
- In chapter 5, the relation with the **proofs as types** theory is emphasized, and we give a new reading for the implementation of basic actions on proof trees.
- In chapter 6, we describe new manipulations on proof trees that were suggested by this implementation. We also show how these could be implemented.

2. SPECIFICATION OF THE PROBLEM

We shall describe here the goals that should be matched by a good implementation of proof trees. First we shall give a feeling for what we want by pointing out the imperfections of a straightforward solution. We want to emphasize that although some imperfections are due to the fact that the implementation language is PROLOG, many problems would arise also in functional or Pascal-like languages. Therefore the data structure that we are looking for should be efficient not

only for PROLOG but also for other implementation languages. In the following, we shall keep PROLOG as the guideline example.

A straightforward implementation of inference rules and proof trees is the natural abstract syntax of TYPOL rules. The formulas inside the rules are called *sequents*. Roughly, an inference rule is a data structure with one field containing a list of hypothese sequents, and one field containing the conclusion sequent. Each sequent is itself a data structure, defined by TYPOL abstract syntax. The natural extension to allow proof trees is that the elements of the first field of a rule may now be sequents *or again* rules.

Our experience is that this data structure is very inefficient in time and memory space, for many reasons:

- Most manipulations involve a *current place* in the structure, and it is expensive to traverse the whole structure from the root to this current place. This problem is independent of the PROLOG implementation language.
- It is often the case that a field of a structure should be given a new value. In PROLOG, this is simply impossible, and the whole data structure needs to be rebuilt. In functional languages, this is dirty but possible, with the famous **rplac** function. In Pascal, the problem is that the data structure is more complex when the old and new values may have different types.
- When building a proof tree, the data structure may become very big, and the memory space needed to store a proof is critical. This problem is also independent of the implementation language.
- All these manipulations of traversing and copying structures are source to many bugs. Even proofreading is difficult, because there is no guideline to improve understanding of the code. Writing the code should be helped by a theoretic background.

Therefore the problem is to define a data structure for proof trees and inference rules such that:

1) The operations of building and manipulating proof trees should be efficient, by avoiding as much structure traversals and structure copies as possible.
2) The memory space for a proof tree should not grow too fast with respect to the number of inference rules involved in it.
3) There should be a theoretical interpretation for the data structure, so that implementation of the actions may be checked more easily. We shall see that this can be seen (surprisingly) as a by-product of the other constraints.
   To a lesser extent, the following are desirable features:
4) The solution proposed should be efficient for any implementation language. However it is clear that we shall take profit of PROLOG for some operations, such as unification.
5) The data structure should be analogous (or equal) for inference rules and for proof trees, this because they are very similar objects in the associated theories.
6) Implementation of basic action should be simple to write and to check, because of the guideline of a theory of proof trees. The code should also be compact enough.

We shall now describe the efficient data structure we designed.

3. REPRESENTING PROOF TREES BY DIFFERENCE TREES

3.1. An analogy with Difference Lists

We saw that the main reasons for speed inefficiency are:

1a) The time spent for traversing a proof tree structure, using some kind of addressing mechanism, to reach a desired place. For instance, one may want to reach some unsolved sequent, or *goal* deep inside the proof tree, in order to either read it or replace it by an inference rule.

1b) The time and space for building the new proof tree after the substitution of an inference rule for a goal. This is because PROLOG doesn't allow to replace subterms of a given term.

If we were programming in Pascal, the best way to solve that is to use pointers. If we know the place(s) of interest inside our structure, we may keep along a pointer towards each place. Then the traversal of the structure is spared. Problem 1b) does not show up in Pascal. *Difference lists* , or *D-lists* are a well known Prolog technique which solves the two above problems 1a) and 1b) with a simulation of pointers. Suppose we want to concatenate two lists. First there is no pointer to the end of the first list, so it must be traversed to reach the end. Second, it is not possible to replace the end, which is the empty list [ ], by another term (which would be the second list). A D-list is an alternative data structure for lists, where the tail of a list is not [ ], but a free variable. To concatenate something at the end of the list is done by simple unification of the last free variable. Then to avoid the traversal of a D-list to reach this last free variable, the free variable is repeated "on the side", just like a pointer was kept along in the Pascal solution. For instance, the D-list for **[a, b, c]** is **[a, b, c | X] :: X** , where "**::**" is a binary infix operator whose first son is the list, and second is the repeated tail variable. The definition to concatenate two D-lists is very simple:

d-concat(L1::T, T::T2, L1::T2).

For proof trees, we must be even more careful, because we want to substitute an inference rule for a subgoal, and not only for a free variable **X** like in D-lists. To do that, we must not commit ourselves by having this subgoal inside the proof tree, but instead we must keep a free variable in its place. It will therefore be possible to unify this variable with the new rule. But since we must keep this subgoal in the meantime, we keep it on the side, along with the repetition of the corresponding free variable. Since this looks very much like D-lists, we call this structure *difference trees* or *D-trees*; it has the same advantages as D-lists have. When **applying** a D-tree on top of another D-tree, it is no longer necessary to traverse the first tree to get the variables, since this variable is kept on the side. The actual application (or plugging) needs no re-building of the first tree, only a free variable must be instantiated.

3.2. Lazy application of rules

Now we want to address requirement number 2), about memory space of a proof tree. We shall rely on the fact that each TYPOL inference rule is given a distinctive name. If any two rules may be distinguished with their names, then it is cheaper to build the proof trees with names of rules rather than with the complete rules.

One extreme way would be to remember only the names of the rules. This would be extremely cheap in memory space. However there is a drawback when building the actual proof tree (for display for example). The passage from the names of the rules to the proof tree must be a translation, that builds the proof tree from scratch, and constructs a large structure. This is expensive in time and space. On the other hand, if we keep the same structure as the final proof tree, leaving some holes (free variables) from time to time, it is much cheaper to fill in the holes when necessary, because one must not build a new term, but just instantiate the old one.

Therefore, when a rule applied has a distinctive name, we take profit of this in the following way: What is plugged inside the proof tree is not the complete inference rule, but the inference rule with a free variable in place of its conclusion. There is already a free variable in place of every premise, because of the D-tree structure, so the actual size added to the proof tree is very small.

3.3. Selection of the next subgoal

In D-trees, the actual unsolved subgoals are not kept in the proof tree (there are free variables

instead). Instead, they are kept in a association table, mapping each free variable to the associated subgoal.

We take profit of this table to ease the two frequent operations: **Select next subgoal** and **Select previous subgoal**. We use two lists. Each element of these lists is a mapping from a free variable to a TYPOL subgoal. The first list contains all subgoals *before* the current one. The second list contains first the current subgoal, then all the following ones. The first list is stored in the reverse order. The constructor used between these two lists is denoted "&&". We call the result a **double stack**. Shifting inside this list of hypotheses is implemented by popping an element from one stack and pushing it onto the other.

As stated by constraint 5), we want a proof tree and an inference rule to have the same structure, for instance because we may apply directly a proof tree to solve a subgoal of another proof tree. When applying a proof tree or an inference rule, one must replace the subgoal in the double stack by a list of subgoals coming from another double stack. This indicates we must use D-lists. Actually only the second stack needs to be a D-list.

For example, starting from:

BEFORE && [CURRENT | AFTER]::TAIL

to go to next subgoal is done by building the new double stack

[CURRENT | BEFORE] && AFTER::TAIL

and to go to previous subgoal is just apply the reverse manipulation.

3.4. Unification and propagation of unifications

When applying an inference rule (or a proof tree) to solve a subgoal, there is a unification between two sequents. The proof tree and the inference rule get modified by the propagation of the unification. We take profit of the fact that we are in Prolog by pushing the TYPOL unification to Prolog. If we were in Pascal, we should have to program our own unification algorithm of course.

What we do here is replacing all TYPOL variables by Prolog variables. This way we may just call Prolog unification between two TYPOL sequents. The propagation of unifications comes automatically. Now, for pretty-printing reasons, it is good that we should keep the TYPOL variable names, so that the user keeps seeing the names he chose himself. So TYPOL names are replaced by Prolog variables, and there is an association list, called the name list, kept along. This is again very similar to D-trees. When an instantiation takes place, it is possible to replace a TYPOL variable by a TYPOL term, since there was actually only a free Prolog variable there. Of course the name list need not be arranged as a double stack. On the other hand, some manipulations are required because of α-conversion problems. Some duplicate names must be renamed, some variables are removed from the list because they got instantiated, the variables that have no TYPOL name are given a generated new name.

So what we get can be viewed as a two-level D-tree. These two levels reflect the two ways of growing a proof tree. First way is to apply a rule on an unsolved goal. This amounts here to instantiating a free variable in the proof tree structure and updating the double stack. Second way is to specialize the proof by precising the value of any free variable. This is done here by instantiating a free variable in a TYPOL subgoal inside the double stack, and updating the name list.

3.5. The complete representation

A proof tree (or an inference rule) is represented by a 4-tuple:

1) First son is the **Structure**, or the **History** of the proof tree. It represents only the name of the rules applied, and their order. It may be transformed into the complete TYPOL proof tree simply by instantiating its free variables (or **holes**). Some holes represent intermediate subgoals (see **lazy application of rules** section 3.2). Some holes represent premises (unsolved subgoals). One hole represents the final conclusion. Conversely, from a TYPOL proof tree one may derive a **Structure** by replacing the sequents by free variables, and keep only the names of the rules.

2) Second son is a pair mapping the conclusion hole to its actual value, i.e. the final goal that is being proved.

3) Third son is the double stack of all the unsolved subgoals. The state of the double stack indicates which subgoal is considered current: the current subgoal is the head of the second stack.

4) Fourth son is the name list. It associates free variable from the sequents in sons 2) and 3) to their actual name, i.e. the TYPOL identifier. Some free variables may have no name. In this case, some new identifier must be generated before pretty-printing time.

Consider for example the following proof tree. It represents some possible stage of the dynamic semantics (interpretation) of an imperative Pascal-like language. Predicates in it mean for instance that, with some given environment (value list), the evaluation of an expression gives some value, or that the execution of an instruction gives some new environment.

$$\dfrac{\textit{plus:}\dfrac{[\mathbf{x}\triangleright 8;\mathbf{y}\triangleright 2]\vdash \mathbf{3.x}\Rightarrow \textit{V1}\quad \textit{int:}\dfrac{}{[\mathbf{x}\triangleright 8;\mathbf{y}\triangleright 2]\vdash \mathbf{5}\Rightarrow 5}\quad \textit{V2}=\textit{V1}+5}{\textit{assign:}\dfrac{[\mathbf{x}\triangleright 8;\mathbf{y}\triangleright 2]\vdash \mathbf{3.x}+\mathbf{5}\Rightarrow \textit{V2}}{[\mathbf{x}\triangleright 8;\mathbf{y}\triangleright 2]\vdash \mathbf{z}:=\mathbf{3.x}+\mathbf{5}:[\mathbf{x}\triangleright 8;\mathbf{y}\triangleright 2;\mathbf{z}\triangleright \textit{V2}]}}\qquad \textit{putenv:}\dfrac{\textit{newvar:}\dfrac{}{[\,],\mathbf{z},\ \textit{V2}\vdash[\mathbf{z}\triangleright \textit{V2}]}}{[\mathbf{y}\triangleright 2],\mathbf{z},\ \textit{V2}\vdash[\mathbf{y}\triangleright 2;\mathbf{z}\triangleright \textit{V2}]}$$

Suppose the current unsolved subgoal is the first one $[\mathbf{x}\triangleright 8;\mathbf{y}\triangleright 2]\vdash \mathbf{3.x}\Rightarrow \textit{V1}$. This choice is arbitrary. The internal representation is shown below, slightly arranged for readability. Big upper case identifiers **C**, **I1**, **I2**, **I3**, **I4**, **I5**, **G1**, **G2**, **X1**, **X2**, **T** represent PROLOG variables.

1) **Structure:**

$$\dfrac{\textit{plus:}\dfrac{\mathbf{G1}\quad \textit{int:}\dfrac{}{\mathbf{I1}}\quad \mathbf{G2}}{\mathbf{I2}}\qquad \textit{putenv:}\dfrac{\textit{newvar:}\dfrac{}{\mathbf{I3}}}{\mathbf{I4}}}{\textit{assign}:\dfrac{}{\mathbf{C}}\qquad \textit{putenv:}\dfrac{}{\mathbf{I5}}}$$

2) **Conclusion:**

$$\mathbf{C}\ \longrightarrow\ [\mathbf{x}\triangleright 8;\mathbf{y}\triangleright 2]\vdash \mathbf{z}:=\mathbf{3.x}+\mathbf{5}:[\mathbf{x}\triangleright 8;\mathbf{y}\triangleright 2;\mathbf{z}\triangleright \mathbf{X2}]$$

3) **Hypotheses:**

$$[\ ]\ \&\&\ [\mathbf{G1}\ \longrightarrow\ [\mathbf{x}\triangleright 8;\mathbf{y}\triangleright 2]\vdash \mathbf{3.x}\Rightarrow \mathbf{X1}\ ,\ \ \mathbf{G2}\ \longrightarrow\ \mathbf{X2}=\mathbf{X1}+5\ \mid\ \mathbf{T}]::\mathbf{T}$$

4) **Name List:**

$$[\mathbf{X1}\ \longrightarrow\ \textit{V1}\ ,\ \ \mathbf{X2}\ \longrightarrow\ \textit{V2}\,]$$

Obviously a rule should be represented in the same way (any single inference rule *is* a proof tree!). Yet there is a slight difference in the **Hypotheses** part. When applying an inference rule

R to a subgoal of a proof tree, there is no notion of a current subgoal in R. Thus the first stack of the double stack of R must always be empty.

Isolated unsolved goals and axioms are trivially translated in the same way, since they are degenerate cases of proof trees and inference rules.

4. IMPLEMENTING BASIC ACTIONS ON PROOF TREES

Now that we have this new representation for proof trees, it is clear that memory space is spared. Let us see what happens to execution times of basic actions on proof trees. We shall focus here on three manipulations: checking applicable rules, applying a rule and simplifying a proof tree.

4.1. Checking an applicable rule

All inference rules known to the system are first translated into internal form one by one, and then each 4-tuple is stored into PROLOG memory as a fact of the following shape:

rule(d_tree(STRUCTURE, CONCLUSION, [ ]&&PREMISES::TAIL, NAME_LIST))

where the four subterms follow the shape defined in section 3.5.

We suppose given a goal, to be checked against the rules. Usually, this goal is the current unsolved goal, and will be found as the first element of the second list of the double stack. The following clause finds an applicable rule to a given goal **G**. It finds all solutions by backtracking. It is obviously fast:

```
applicable(G, d_tree(S, C, P, N)) :-
        rule( d_tree(S, C, P, N)),
        not( not( C = _ → G )).
```

4.2. Applying a rule

Now we suppose given two D-trees. One represents the proof tree in its current state. This proof tree has a current unsolved subgoal. The second D-tree represents the inference rule to apply on the current unsolved subgoal. More generaly, this second D-tree may also be any proof tree. We suppose that this inference rule has been selected among the applicable rules (*cf* previous paragraph), or at least has been checked applicable. There is no need for a current subgoal in the D-tree representing the inference rule, therefore we may suppose that the first list of its double stack is empty. We want to build the proof tree grown by applying the inference rule.

The following clause does that. It takes the initial proof tree as first argument, the applicable rule as second argument, and the resulting proof tree as third argument. It takes care of giving the resulting proof tree in the form of a D-tree itself. Also the new name-list is computed.

```
apply( d_tree(PROOF, C1, B1 && [VAR1 → G1 | A1] :: T1, N1),
       d_tree(RULE, VAR2 → C2, [ ] && A2 :: T2, N2),
       d_tree(PROOF, C1, B1 && A3 :: T3, N3)) :-
    G1 = C2,
    VAR1 = RULE,
    d_concat(A2 :: T2, A1 :: T1, A3 :: T3),
    combine_name_lists(N1, N2, N3).
```

Let us explain this clause. First we must check that the rule is applicable, and this time we must perform the unification. This is done by **G1 = C2**. Note that all the unifications are propagated automatically in the proof tree and in the rule. Then we must plug the rule structure **RULE** on the

proof structure, by instantiating the correct PROLOG variable, which is **VAR1**. Note that neither the text of the conclusion nor the text of the hypotheses are in **RULE**, so that **PROOF** remains a lightweight structure, and the application of the rule is lazy in the sense of section 3.2. Then the new double stack is obtained by concatenation of the new hypotheses between the "BEFORE" and "AFTER" part of the hypotheses list. That is done using the concatenation of two D-lists, **d-concat** as defined in section 3.1. The new current subgoal is the first subgoal of the rule, which seems natural enough. Last thing is to merge the two name lists.

Now this clause may be more compact, because the = predicate can be expanded in place, as well as **d_concat2**. The clause becomes thus:

```
apply( d_tree(PROOF, C1, B1 && [RULE → G | A1] :: T1, N1),
       d_tree(RULE, VAR2 → G,[] && A2 :: A1, N2),
       d_tree(PROOF, C1, B1 && A2 :: T1, N3)) :-
   combine_name_lists(N1, N2, N3).
```

This is much more efficient than the straightforward method. because both the traversal of the initial proof tree and the construction of the resulting proof are spared, and replaced by a single unification. The only thing that could not be spared is the merging of the two name lists.

4.3. Simplifying a proof tree

Given a proof tree, one often wants to see in a simpler way what this proof tree is all about. Seeing all the inferences is sometimes confusing when one wants to know what the proof tree really means. What we provide is a tool to "shrink" the proof tree, forgetting its history and keeping only the hypotheses and conclusion. Thus the simplified proof tree is an inference rule.

In this implementation, simplification is done simply by modifying only the **structure** part of the D-tree. We have to build the structure of a single inference rule, whose conclusion is replaced by a variable, and where the list of hypotheses is replaced by a list of free variables. These variables are to come from the **conclusion** and **hypotheses** parts of the D-tree. The TYPOL abstract syntax operator for an inference rule is named **inf_rule**. The clause to do this simplification is:

```
simplify(d_tree(PROOF, VAR → C, H, N),
         d_tree(inf_rule(LVAR, VAR), VAR → C, H, N)) :-
   collect_vars(H, LVARS).
```

Here again, this implementation is fast, because the only thing built is the **structure** part, and the term built is short. Moreover the current unsolved goal is kept the same, since the **hypotheses** part is not changed. The predicate **collect_vars** traverses the double stack to build in the right order a TYPOL list of sequents, in which there is a variable in place of each sequent, this variable unified with the corresponding one in the double stack.

One may decide that this new rule must be added to the others in the PROLOG memory. The only thing to do then is to modify the double stack part so that all hypotheses go to the second stack and the first stack is empty. Then do an **assert** to add it to the previously known rules. We call this adding a **lemma** to our environment. An important thing the name of the new proof tree. We must give it a name so that if the added lemma is used, we may go on with lazy application.

4.4. Pretty-printing a proof tree

The first thing to do to pretty-print a D-tree is to transform it back to the corresponding TYPOL abstract syntax tree. Everything was prepared to ease this operation. All the free variables that were carefully kept free can now be given their value by a simple unification.

On the other hand, the D-tree structure must remain unchanged outside of the pretty-printing routine. The way to do that is the classical call to the **not not** combination.

What must be done is:

- Unify the conclusion free variable with the conclusion text (sequent).
- Unify the tail of the second list of the double stack with [].
- Unify the hypotheses variables with their respective text (sequent).
- Fill in the intermediate sequents holes by unifying each named inference rule (with holes) from the proof with the actual text of the named rule. This step is a kind of last checking that the rules that were applied were *really* applicable, and it is very similar to type-checking. In section 5, the similarity will be given a theoretic background.
- Replace the free variables representing TYPOL variables by the actual TYPOL variables, coming from the **name list**.
- Lastly, when some free variables remain (because α-conversion problems prevented them from having a name in the name list), just generate new TYPOL identifiers to name them.

One may note that this is a lot of work for a simple pretty-printing. However, this is the price to pay to keep an efficient internal form for proof. The situation is somehow analogous to an implementation of rational numbers using two integers (numerator and denominator). This is a fast implementation for multiplying or dividing two rationals, but it takes more time to print the result in decimal form, more useful to the user. Here is the program to do this pretty-print of proofs:

```
pretty_print(D_TREE) :-
        not not pp(D_TREE).

pp(d_tree(PROOF, VAR → CONCL, BEFORE && AFTER :: [], NAME_LIST)) :-
        VAR = CONCL,
        l_unify(BEFORE),
        l_unify(AFTER),
        rules_unify(PROOF),
        l_unify(NAME_LIST),
        gensym_freevars(PROOF),
        typol_display(PROOF).

l_unify([]).

l_unify( [TEXT → TEXT | ASSOC_LIST]) :-
        l_unify(ASSOC_LIST).
```

The code for the **rules_unify** predicate is slightly longer, but gives no further problem.

5. THE THEORETICAL INTERPRETATION

In this chapter, we are going to show how close this implementation is from the theories which assimilate formulas and types. In these theories, proof-checking is the same as type-checking. Actually, If we look for a straightforward implementation of these theories, it is very likely that we should end up with something similar to our fast implementation of chapters 3 and 4.

5.1. "Formulas as types" theories:

Formulas as types theories first consider that for proving proposition P, one should start by asking the question $a \in P$. This question is given many meanings at the same time:

i) a is an element in the set P.

ii) a is a proof of proposition P.

iii) a is an object of type P.

A more precise presentation can be found in many papers, for instance in [7].

Suppose one starts from a proposition P and then builds a proof tree T such that no more unsolved goal remains on top of T. Naturally enough, according to the *ii)* interpretation of formulas, T is a proof of proposition P. But also, according to *iii)*, T is an object of type P. This latter interpretation leads to consider inference rules as functions which, to objects (proof trees) of type P_1, P_2, ..., (the hypotheses sequents), associate an object (a bigger proof tree) of type C (the conclusion). Therefore the inference rule:

$$and\_rule : \frac{\rho \vdash \mathrm{A} : true \qquad \rho \vdash \mathrm{B} : true}{\rho \vdash \mathrm{A \ and \ B} : true}$$

is represented as the following function in typed lambda notation (Types are between curly braces for readability):

$$and\_rule = \lambda \mathrm{P}_1{:}_{\{\rho \vdash \mathrm{A} : true\}} . (\lambda \mathrm{P}_2{:}_{\{\rho \vdash \mathrm{B} : true\}} . and\_rule(\mathrm{P}_1, \mathrm{P}_2){:}_{\{\rho \vdash \mathrm{A \ and \ B} : true\}})$$

So our initial proof tree T of formula P, is also an object of type P, since it is a function with no argument (the proof tree has no unsolved subgoal) that gives a result of type P. It is thus equivalent to a constant, or an object, of type P.

In this interpretation, what is the application of an inference rule on top of a proof tree? We are given a partial proof tree, *i.e.* a proof tree with some goals unsolved yet. This proof tree can be viewed as the skeleton of an eventual complete proof tree. The interpretation *ii)* says that, from a skeleton of proof of P, application of a rule gives another skeleton of proof of P. Every instantiation of the second skeleton will be an instantiation of the first skeleton. It is more interesting to apply interpretation *iii)*. Application of a rule to a proof tree becomes the same as the composition of functions. For instance applying

$$rule: \frac{\mathrm{A}_1 \quad \mathrm{A}_2}{\mathrm{B}_2}$$

on the proof tree

$$\frac{\mathrm{B}_1 \quad \mathrm{B}_2 \quad \mathrm{B}_3}{\mathrm{C}}$$

is the same as composing the functions

$$rule = \lambda y_1{:}_{\{\mathrm{A}_1\}} . \lambda y_2{:}_{\{\mathrm{A}_2\}} . rule(y_1, y_2){:}_{\{\mathrm{B}_2\}}$$

and

$$proof = \lambda x_1{:}_{\{\mathrm{B}_1\}} . \lambda x_2{:}_{\{\mathrm{B}_2\}} . \lambda x_3{:}_{\{\mathrm{B}_3\}} . \mathrm{PROOF}{:}_{\{\mathrm{C}\}}$$

Where PROOF is a formula containing free occurrences of x_1, x_2 and x_3. The composition is done on the argument number 2 of the second function. We are going to denote this by some combinator $\circ_2$. Applying the rule *rule* on the proof tree *proof* is just building the composition

$$new\_proof = \circ_2 \ rule \ proof$$

$$= \lambda x_1{:}_{\{\mathrm{B}_1\}} . \lambda y_1{:}_{\{\mathrm{A}_1\}} . \lambda y_2{:}_{\{\mathrm{A}_2\}} . \lambda x_3{:}_{\{\mathrm{B}_3\}} . PROOF[rule(y_1, y_2) \setminus x_2]{:}_{\{\mathrm{C}\}}$$

Please note that this $\circ$ notation is handy for simple compositions like the previous one, but is very clumsy when compositions are more complex. In the general case, we must define a combinator.

If we take the notations of [2], the general combinator $\mathbf{A}_p^n$ to apply an inference rule *rule* with n premises to the subgoal of rank p inside proof tree *proof* gives

$$\mathbf{A}_p^n \; proof \; rule$$

such that

$$\mathbf{A}_p^n \; proof \; rule \; x_1 \; x_2 \; \ldots \; x_{p-1} \; x_p \; \ldots \; x_{p+n-1} \;\; = \;\; proof \; x_1 \; x_2 \; \ldots \; x_{p-1} \; (rule \; x_p \; \ldots \; x_{p+n-1})$$

The definition of the combinator in terms of simpler ones will be of little importance here, since we have chosen instead to define it by its operation on $p + n - 1$ arguments. Nevertheless, to be complete, let us just give that definition:

$$\mathbf{A}_p^n = \mathbf{C}_{[p-1]}^{-1} \bullet \mathrm{B}_{(p-1)}^n$$

where $\bullet$ is the composition, the superscripts are powers with respect to composition, $\mathbf{C}_{[p-1]}$ does a circular permutation from its second argument to the one of rank $p+1$, and $\mathrm{B}_{(p-1)}$ is the combinator B deferred by $p-1$ steps.

Let us see now where unification appears. When composing a rule and a proof, some type-checking has to be done. The type of the result of the rule must correspond to the type of the selected argument of the proof. For the consistency of the resulting proof, these two types must have (at least) a common element. This is best said using the *i)* interpretation of $a \in \mathrm{P}$. The rule and the proof can be composed if the object set of the rule has a non-empty intersection with the set of all the values of the selected argument in the proof. So we have two formulas (or two sets) P_1 and P_2, and we must check that they intersect. In our context, where the types are formulas, the intersection of two types is the same as unification. Two types intersect if they are unifiable. When they intersect, their intersection is the result of the unification.

Therefore, when applying to an unsolved goal P_p of a proof, a rule whose conclusion is the sequent P_r, both the rule and the proof will be restricted. If P' is the unification of P_p and P_r, then the rule will be restricted by instantiating it so that P_r becomes equal to P', and the proof also will be restricted by instantiating it so that P_p is equal to P'. Only after this restriction may the combination be performed.

We now have described what the manipulation of proofs should be in a **formulas as types** theory. We are going to show that a straightforward implementation of it in TYPOL is equivalent to the implementation described in the previous chapter.

5.2. Equivalence with the D-tree structure:

An inference rule is a function. Its representation must contain only the following points:

a) Its list of arguments.

b) For each argument, its type.

c) The formula denoting the result, in which the arguments may occur free. For a rule, this formula is trivial and redundant: *name*(arg_1, arg_2, ...). Its only purpose is to denote the *name* of the function.

d) The type of the result.

A proof tree is also a function. Its representation is the same, except for the *c)* part, which is more elaborate because it keeps track of the successive applications of rules. For example, when a proof has formula PROOF, and that the second argument x_2 is expanded by the rule

$$rule = \lambda y_1{:}_{\{\mathrm{T}_1\}} \, . \lambda y_2{:}_{\{\mathrm{T}_2\}} \, . name(y_1, \; y_2){:}_{\{\mathrm{C}\}}$$

then the resulting proof has formula

$$\text{PROOF}_2 = \text{PROOF}[name(y_1,\ y_2) \setminus x_2]$$

We can see now that this is equivalent to our representation in previous chapter.

- The **Structure** part is the *c)* part, and the substitution used to build the bigger proof formula PROOF_2 above is implemented by instantiation of a free variable. Actually, Prolog instantiation of a free variable is the same as substitution of a term for this variable.
- The **Conclusion** part is the *d)* part, the type of the result (or the conclusion).
- The **Hypotheses** part is the list of the arguments *a)*, together with their types *b)*.
- The **Name List** part has no counterpart here, since it is only useful for pretty-printing. Problems of α-conversion, which are treated in the **Name list**, have been implicitely neglected in this section.

5.3. New reading for the apply predicate:

The application of a rule to a proof is a function composition. The manipulations of the **double stack** are just the selection of the index p in the combinator $\mathbf{A}_p^n$. Function composition starts by restricting the two functions by unification of the types (formulas) P_p and P_r. Then the new list of arguments (hypotheses formulas) is built, and the new formula is obtained by the correct substitution in the old one.

The implementation of the apply operation in the previous chapter is the same. Restriction of proof and rule is done (when possible) by unification of the two sequents. After this unification, the proof and the rule are instantiated so that they may be combined. The new list of argument is built by concatenation of the list of arguments of the rule just after the && in the double stack. The last operation performed is unification of the free variable representing the goal to expand with the **Structure** of the rule. It is easy to see that it is exactly the same as substituting the rule formula for all occurrences of the chosen proof argument in the proof formula.

5.4. New reading for the simplify predicate:

In **formulas as types** theory, simplification does not seem very important. The only effect of it is to forget about the successive combinations of rules, to keep only the type information. In other words, to simplify a proof is done by replacing the formula in it by a much simpler one, which is the mere *list* of its arguments (the variables occurring free). So just replace the proof formula (with free occurrences of $\mathbf{x}_1$, $\mathbf{x}_2$, ...) by

$$some\_new\_name(\mathbf{x_1}, \mathbf{x_2}, ...)$$

It is clear that this is exactly the behavior of the **simplify** predicate.

6. NEW POSSIBILITIES

This different way of reasoning about proof trees suggests new interesting operations on TYPOL proofs. Many of these operations become easier to implement, and more efficient, with the D-tree representation.

- **More compact storage**
  The D-tree notation is more compact. Moreover, the information needed to expand it back to its pretty-printed form is not local to the proof development session. This means that one may store the proof tree in a file, then restore it in a new Centaur session, and no information is lost if the TYPOL files were kept unchanged. This comes from the fact that the only external

information in a D-tree is the text of the TYPOL rules, and this text is easily retrieved using the names of the rules.

- **Loading of a brand new proof**
  It is now possible to start from a brand new proof tree. The format to load this proof tree can look like the "**history**" part of a D-tree. Then an important operation must be done, which is the checking of this proof tree. From the theoretical standpoint this is nothing but the type-checking of the proof. This operation is already implemented in the new formalism. According to the "**history**" part, all applied rules are checked applicable. Each time a rule is checked applicable, the variables instantiations are propagated, and the next checkings of rules applications obviously take in account (are restricted by) these instantiations. It looks like what is done for pretty-printing of a D-tree. Actually pretty-print is just type-checking followed by display.

- **Proof tree pruning and grafting**
  Since the D-tree notation separates the rules application schema from the rest (in the **structure** part), it is possible to remove a branch from the proof tree. It is clear that the rest of the proof tree is still legal on its own (the rules applied are still applicable). The remaining proof tree can then be grown again in the usual way. It is also possible to keep the pruned branch of the proof tree, which is also a legal proof tree on its own. One could also think of the reverse operation: "grafting".

- **Proof tree generalization**
  A proof tree may be built from an original goal more or less precise. Some precision (instantiations) in this goal may turn out to be useless, because all the rules applied could prove a more general thing. Also, when a proof tree comes from pruning operations, it may keep unifications that are no longer relevant. Thus an interesting operation is to generalize a proof tree to its most general shape, deduced only from the inference rules applied. This is very easy to implement here. Just keep the **Structure** part of the proof (and also the **Name list**), throw away the **Conclusion** and **Hypotheses** parts, and then load this as a brand new proof. As we saw, this will generate the most general hypotheses and conclusion sequents possible. We call this a generalization of the proof tree.

7. CONCLUSION

We presented an improved implementation of a proof tree builder. The main point in this implementation is the introduction of a new data structure, **D-trees**, to represent proof trees.

Though this improvement was initially motivated only by efficiency concerns, it now looks like a good compromise between a theory-driven implementation and a more user-comfortable one. The current coding of the "**applicable**", "**apply**" and "**simplify**" predicates can be seen as an exact paraphrasing of the theory. The only concessions to comfort are the following:

i) The predicate called to merge the informations about the possible names of free variables, "**combine-name-lists**", has no counterpart in the theory. It just deals with α-conversion, which is a rather clumsy piece of prolog code, and invents new names when necessary.

ii) The "**simplify**" predicate, as we saw, is not very relevant from the theory point of view. It is just here for comfort reasons, to provide the user a more compact display of a proof.

iii) The pretty-printing part has also little relation with theory, since the basic representation for a proof tree is the function notation, and not the TYPOL proof tree shape.

We found it very interesting that the programmer's point of view happens to meet a clean and strong theory. Of course it looks funny, and somehow it points out that it was lazy to consider only speed concerns, while a cleaner theoretic approach would had led directly from theory to the

"good" implementation. Another positive consequence is also that the new implementation is easier to debug, mostly because of its conciseness and closeness with the theory.

Further research can be done to extend the notion of "applicable" rule. From the theory point of view, it amounts to modifying the type system, and when two types have an intersection. From the practical point of view, this amounts to introducing something more complex than unification. For instance one could think of guarded clauses, where the conditions in the guard can not be expressed only by unification. The definitions of "**applicable**" and "**apply**" should then be modified. We already have a running version of a proof builder with such guards, where the applicability of a rule is restricted by constraints on TYPOL terms, such as belonging or not belonging to a given set. This special case of restriction is very commonly needed in actual TYPOL programming.

More generaly, one may want to change the definition of unification. Examples are introducing $\beta\eta$-conversion, or allowing inside lists variables denoting sublists. The present implementation is convenient because unification at the object level is called only in localized places, and it is thus easier to plug in another unification algorithm.

REFERENCES

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

[2] **Curry, H., Feys, R.**, *Combinatory Logic*, volume 1, chapter 5, North-Holland, (1968).

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

[4] **Hascoët, L.**, *"A tactic-driven system for building proofs"*, Actes du 7^{eme} séminaire Programmation en Logique, Trégastel, France, May 25–27, (1988), also INRIA Research Report N^{o}770, (1987).

[5] **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).

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

[7] **Nordström, B.**, *"Martin-Löf's Type Theory as a Programming Logic"*, Report 27, Programming Methodology Group, University of Göteborg, (September 1986).