THEO: an interactive proof development system

Joelle Despeyroux

▶ To cite this version:

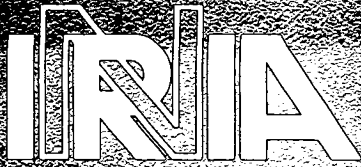
Joelle Despeyroux. THEO: an interactive proof development system. [Research Report] RT-0116, INRIA. 1990, pp.17. inria-00070050

HAL Id: inria-00070050
https://inria.hal.science/inria-00070050

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



UNITE DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports Techniques

N° 116

Programme 1
Programmation, Calcul Symbolique et Intelligence Artificielle

THEO : AN INTERACTIVE PROOF DEVELOPMENT SYSTEM

Joëlle DESPEYROUX

Février 1990

RT_0116

Theo: An interactive proof development system.

Theo: Un système de développement de preuves interactif.

Manuel utilisateur

User's manual

Joëlle Despeyroux

INRIA, Sophia-Antipolis, 2004 Route des Lucioles,
F-06565 Valbonne Cedex, France
e-mail: joelle.despeyroux@mirsa.inria.fr

Abstract

Theo is an interactive, tactic-driven, proof development system -a Theorem Prover. Both the meta and the object levels of our theorem prover are logics presented in Typol. Typol is the programming language that implements Natural Semantics -a semantics developed at Inria and pioneered by G. Plotkin under the name Structural Operational Semantics. So Theo is written in Typol and helps the user to build proofs in an object logic also written in Typol. Other important features of Theo are the form chosen for representing proofs, and the way proofs are performed. The internal form of the proofs is a very compact form, expressed with combinators, that can be related to the λ-calculus used in Automath and its descendants. Meanwhile, Theo performs proofs by a pure calculus on proofs, using a resolution rule. Proofs may be incomplete, and may contain logical variables. Theo is developed under the Centaur system, as well as Typol. This system provides a modern graphic man-machine interface that Theo uses, for the user's advantage. This is the reference and user's manual for Theo version 1.0 runing under Centaur version 1.0.

Résumé

Théo est un système de développement de preuves -ou prouveur de Théorèmes- interactif, dirigé par des tactiques. Les niveaux meta et objet de notre prouveur de théorèmes sont tous deux des logiques présentés en Typol. Typol est le langage de programmation implémentant la Sémantique Naturelle -une sémantique développée à l'INRIA à partir de la Sémantique Operationnelle Structurelle de Gordon Plotkin. Donc Théo est écrit en Typol et aide l'utilisateur à construire des preuves dans des logiques objets également écrites en Typol. Les autres principales caracteristiques de Theo sont la forme choisie pour représenter les preuves, et la manière dont les preuves sont construites. La forme interne des preuves en Théo est une forme très compacte. Elle utilise des combinateurs, et est très proche des lambda-calculs utilisés dans Automath et ses descendants. Théo, pourtant, construit les preuves en utilisant une règle de résolution. Les preuves peuvent etre incomplètes, et contenir des variables. Théo, ainsi que Typol, sont développés sous le méta-compilateur Centaur. Centaur fournit une agréable interface homme machine graphique, que Théo utilise, à l'avantage de l'utilisateur. Ceci est le manuel utilisateur ainsi que le manuel de reference de Théo version 1.0 implementé sous Centaur version 1.0.

$\nabla \vdash$

1. Introduction. Original motivations and Natural Semantics background

A Natural Semantics specification [6][16] is a formal system, that may be a logic or not, depending of the example. The idea of using a formal system to describe semantics of languages comes both from G. Plotkin's Operational Semantics [23]and unpublished notes of Robin Milner. A Natural Semantics specification is defined by means of axioms and inference rules, using Gentzen's sequents [11][3]. Models are not our concern. In Natural Semantics, all questions about program behavior or semantic properties of a language are understood as theorems or derived rules of inference to prove in the context of the 'logic' specifying the semantics. We first used Natural Semantics to describe static or dynamic semantics of programming languages, then more generally to describe any logic that can be expressed in it. Natural Semantics is implemented in a programming language called Typol [9]. A Typol program can be type-checked and compiled -presently into prolog or lisp. It is usely sufficient to compile Typol programs describing static or dynamic semantics to get either type-checkers or interpreters, compilers, debuggers, etc... This is no longer the case as soon as the logic describes a non-deterministic 'semantics', typically the dynamic semantics of a non-deterministic language, but also First Order Logic or the type-checker of the Calculus of Constructions of Thierry Coquand and Gerard Huet, for example. For these examples, we need an *interactive theorem prover.* In fact, the need of an interactive theorem prover goes even back to [6], where we prove the correctness of a translation, i.e. in our framework, a derived rule, by induction on the length of the proof. At that time we were already building Typol proof-tree, 'by hand', saying that this construction should be mechanized.

Theo was first presented in [7]and then in [8], where we discussed its choices at length, comparing it with other systems. Let us just list some of them here. The first systems being implemented were LCF [20]and Nuprl [5], which is still the most sophisticated system. Among the more recent systems, two systems use Prolog: Laurent Hascoët 's system [14]and λ-prolog [17][19][10], which may be -with Isabelle- the closest system to Theo. Two systems are not theorem provers but Mathematical and Logical Frameworks: the Calculus of Constructions [4][15]and the Edinburgh Logical Framework (ELF) [13][2]. T. Griffin's system is a theorem prover implemented under the Cornell's Synthetizer, initially designed to build proofs in ELF. Finally Isabelle [21][22]uses a resolution rule similar to our rule. It is worth noting that the use of a resolution rule enables one to avoid the validation function of LCF. Isabelle mainly differs from Theo in two points. The first one is the form of tactics, which, in Theo, are expressed in Typol, by means of inference rules. Then a proof, in Isabelle, is represented by a proposition of the meta logic, which is higher-order logic. This proof represents a derived rule of the object logic. So it is not really a -full- proof but more exactly an 'assertion of provability in the object logic'. This means that it seems a-priori not possible to do some calculus on a proof. This seems to us the only drawback of Isabelle.

This Manual is made of five parts. We first presents some features of our theorem prover 'Theo', particularly the form chosen to represent a proof. The second and third parts present a very simple example of an object logic for Theo, together with an example of a proof development. The resolution rules we use to develop a proof are given in the fourth part, while tactics are described in the last part.

2. The theorem prover

Theo is an interactive, tactic-driven system designed to help the user to build proofs by proving theorems or derived rules, in a backward way. As tactics are for us parts of the meta-logic, it seems natural to us to implement them by means of inference rules. So both the meta and the object levels of Theo are logics written in Typol, that evolves to fit the requirements of both Typol and Theo users. Theo is implemented under the Centaur system, which gives it two advantages. First the theorem prover manipulates any formula as a tree, second it naturally inherits the man-machine graphic interface of Centaur. First versions of Theo were presented in [7][8].

2.1. *Developing a proof under Theo. General organization of the system*

First of all, you must type-check and compile your object logic. As usual for such a function in Centaur, you have three ways to do that. In the interactive mode, you can either use buttons *type check* of the Typol menu and *compile logic* and *write code* of the Theo menu, or compile the Typol files involved in a 'Centaur -top', using the function 'theo-compile-files' in the same way you would use the function 'Typol-compile-files'. In the batch mode, you will use the Centaur command 'ctmake'. If you forget to do that, the first time where

Theo will need a rule of the object logic, he will stop, saying that he does not know it. You must type-check and compile your object logic each time you modify it. The compiled version of a rule is the same rule where every name of a set has been 'completed'. For example, a set S belonging to a set S1 of a program P must be named something like 'S-S1-P' in order to disambiguate it from an other set S in an other program P1 (S-P1). The operator names are also desambiguated, that is contain the name of the language involved. The compiled logic also contains some extra informations for Theo.

When you 'click' on the button *run* in the Theo menu, Theo first asks for a sequent to prove [1] , or for a partial proof to complete, in an object logic written in Typol (for example a dynamic semantics of a language, First Order Logic, a type inference system for the Calculus of Constructions, etc...). After some quick initialisations, the system enters in a while-loop, asking for a tactic to apply and some arguments, among which is the sequent to prove -the 'current goal'. Arguments are formulas, often sub-proof, that the user -thanks to the Centaur system- gives simply by designating them with the mouse. A tactic is chosed by selecting its name in the *Tactics' box*. After having given the arguments of a tactic, the user must click on the button *done* in this Tactics' box. Theo opens and uses its own windows, whose names are prefixed by **theo*. For example, the proof being built is displayed in a window named '*theo-proof*'. (The current goal has to be designated in it.) You are allowed to *stop* to built your proof at any time, eventually save your partial proof, and complete it later on. This feature is particularly useful when you want to build big proofs. In that case, you may stop the proof of your main theorem, prove a sub-theorem or a derived rule, add it in your object logic, and use it later on to proceed with the main proof.

2.2. *Representation of a Proof*

We defined a *proof*-or a proof term- with the aim to obtain the most compact form necessary to represent a proof, in a framework where proofs are developed backwards, using a resolution rule. The definition of a proof uses combinators, and defines a calculus that can be related to the λ-calculus used in Automath and its descendants [8]. A proof is recursively defined by:

- $R : T$, a sequent of the object logic. *axiom or theorem of name R of the object logic - atomic proved term*

- T, a sequent of the object logic. *term to prove*

- $(R\ P_1 \cdots P_n)$, where R is the name of an inference rule of the object logic and the list $P_1 \cdots P_n$ is a list of proofs. *application*

In general, a term as above contains some variables and denotes *an incomplete proof.* A term that contains no further 'term to prove' is a complete proof.

3. A simple example of an object logic

As a very simple example of an object logic for Theo, let us take the well known Propositional Logic. This example is sufficient to fix ideas. More interesting examples are presented in a forthcoming paper.

In the Centaur system, all formulae are manipulated as abstract syntax trees, parsed and pretty-printed in concrete form. So we have given to Centaur the concrete and abstract syntax of our propositional logic, togother with its pretty-printer. The inference rules we chosed are the propositional part of the intuitionist Gentzen's Sequents System for First Order Logic, called 'System LJ' [11], with only one structural rule instead of four. The rules given in Fig. 1 define the judgement $p \vdash a$ that may be read as 'p implies a'. The rules are to be read with the declaration of the judgement prop_s $\vdash$ prop. The inference rules of Fig.1 are the pretty-print version of the Typol inference rules.

---

[1] The theorem to prove must be written in Typol, using the abstract syntax of the object logic used. A good way to obtain this Typol term is to produce it by translating a term written in the object logic. See for example impl.cc and impl.ty in directory cc/cc/examples together with the translator cc_typol.ty in directory cc/cc/semantics.

$\Rightarrow_s$:	$\dfrac{a, p \vdash b}{p \vdash a \Rightarrow b}$	$\Rightarrow_a$:	$\dfrac{p \vdash a \quad b, p \vdash c}{a \Rightarrow b, p \vdash c}$		
and_s:	$\dfrac{p \vdash a \cdot p \vdash b}{p \vdash a \wedge b}$	and_{al}:	$\dfrac{a, p \vdash c}{a \wedge b, p \vdash c}$	and_{ar}:	$\dfrac{b, p \vdash c}{a \wedge b, p \vdash c}$
or_{sl}:	$\dfrac{p \vdash a}{p \vdash a \vee b}$	or_{sr}:	$\dfrac{p \vdash b}{p \vdash a \vee b}$	or_a:	$\dfrac{a, p \vdash c \quad b, p \vdash c}{a \vee b, p \vdash c}$
not_{sj}:	$\dfrac{a, p \vdash false}{p \vdash \neg a}$	not_{aj}:	$\dfrac{p \vdash a}{\neg a, p \vdash false}$		
axiom:	$a, p \vdash a$	*struct*:	$\dfrac{D, p, q \vdash a}{p, D, q \vdash a}$		

Figure 1. Inference Rules of FOL

Notations: Inferences rules are all 'introduction rules'. The names $\Rightarrow_a$, $\Rightarrow_s$, for example, are intended to mean an introduction rule in the antecedent (a) or succedent (s) of the implication ($\Rightarrow$). The name and_{al} (resp. and_{ar}) means introduction to the left (l) (resp. to the right (r)) in the antecedent of the conjunction.

From the point of view of programming languages, this example is interesting as a simple non-deterministi language. For example, the following structural rule:

$$struct : \frac{D, p, q \vdash a}{p, D, q \vdash a}$$

raises two problems, when used in a 'automatic' way. First the rule can be used at any time. Then, to apply it backwards, we would have to find, by pattern matching, an expression (D) in a list (p, D, q). As soon as an interactive theorem prover is available, an easy, and natural way, to solve the problem is to ask the user to designate with the mouse the expression D that he wants to consider in his proof. The resulting list D, p, q is build by the theorem prover, using a simple special purpose tactic.

4. An example of a proof development

Consider the Propositional Logic described in the previous section. Suppose we look for a proof of $\vdash x \vee (y \wedge z) \Rightarrow (x \vee y) \wedge (x \vee z)$. Theo first computes the corresponding incomplete proof, which is the same term. Then the construction of the proof begins, asking for a tactic to apply at each step. At the first step, the user may choose the following rule:

$$\Rightarrow_s : \frac{a \vdash b}{\vdash a \Rightarrow b}$$

Theo then computes and display the following incomplete proof term:

$$(\Rightarrow_s (x \vee (y \wedge z) \vdash (x \vee y) \wedge (x \vee z)))$$

The construction of the proof goes on, possibly producing the following partial proof:

$$(\Rightarrow_s (or_a \quad x \vdash [x \vee y] \wedge [x \vee z] \quad [y \wedge z] \vdash [x \vee y] \wedge [x \vee z]))$$

and finally the following complete proof:

$$(\Rightarrow_s (or_a \ (and_s \ (or_{sl} \ x \vdash x) \ (or_{sl} \ x \vdash x)) \ (and_s \ (or_{sr} \ (and_{al} \ y \vdash y)) \ (or_{sr} \ (and_{ar} \ z \vdash z))))))$$

At any time, the user can ask for the proof tree that corresponds to a proof, or a sub-proof. Theo computes the proof tree by 'pretty-printing' the proof, that is by applying each rule involved. The proof term is compact and clearly shows both the structure of the proof and the remaining goals, while the proof tree displays more information. For example, the proof tree corresponding to the proof $(or_{sr}\ (and_{al}\ y \vdash y))$ is the following:

$$\cfrac{\cfrac{y \vdash y}{y \wedge b \vdash y}}{y \wedge b \vdash a \vee y}$$

5. Two Resolution Rules

Proofs are developed using a Resolution Rule. It is worth noting that this allows us to avoid the validation function of LCF. The resolution rule has two forms, depending on the *mode* chosen. More precisely, there are two cases: either the user wants to find a *general proof,* either he is looking for a *particular proof.* For example, a proof in First Order Logic is a general proof, whereas a proof in the Calculus of Constructions (CC) is a typical particular proof. In the first case, the user wants to prove a sequent, for any value of the variables contained in it. In the case of CC, the user is looking for a term p that has type t, in an environment ρ. The user is building there a proof of $\rho \vdash p : t$, where p is a variable, which becomes more and more instanciated, as the proof proceeds. In other words, the user is looking for a particular proof of its initial goal.

5.1. *General proofs*

In this case the resolution rule consist in two Typol rules, which can be pretty-printed as follows (technical details are omitted):

$$\frac{P \overset{\text{app}}{\vdash} gen(R),\ c \rightarrow P'}{P \vdash R,\ c \rightarrow inst(P')} \qquad P \overset{\text{app}}{\vdash} n : \frac{p}{c},\ c \rightarrow P[(n\ p)/c]$$

The above rules describes the application of an inference rule R to a term c in a proof P. In the first rule, the rule R is 'generalized'. (This means that Typol identifiers are replaced by variables, which can later on become any term.) Then the new proof resulting of the application is 'instantiated' (only possibly new variables appearing in the premises of rule R need to be instanciated).

The second rule realizes the application of an inference rule $\frac{p}{\phi}$, of name n, to a term ψ in a proof P. First ϕ is match against ψ, resulting in c. Then the result is P in which the sequent c has been replaced by a term denoting the application of the rule n to a list of new terms to prove p_i, one for each premise of the rule. (This substitution is specified by a set of inference rules, not by β-reduction.) Notice that the application is mostly realized by unification.

5.2. *Particular proofs*

To find particular proofs, the resolution rule must be slightly modified:

$$\frac{gen(P) \overset{\text{app}}{\vdash} gen(R),\ gen(c) \rightarrow P'}{P \vdash R,\ c \rightarrow inst(P')} \qquad P \overset{\text{app}}{\vdash} n : \frac{p}{c},\ c \rightarrow P[(n\ p)/c]$$

In the first rule, in order to instanciate the current proof P, both the sequent c, the proof P and the rule R are 'generalized'. In the second rule, the unification matching ϕ against ψ affects the all proof P, thus possibly instanciating free variables, to find a 'particular proof' of the sequent c.

6. Tactics

Each tactic can be used in either of the two 'modes' (general or particular) that we described in the previous section. The mode can be changed dynamically, while building the proof. (Click in the *Tactics' mode box*). Each time the user asks to apply a tactic, Theo looks for the current mode, in order to use the corresponding form of the resolution rule. The basic implemented tactics are *user, breadth first* -a complete Prolog strategy, which is rarely implemented, because of its high costs-, and the direct call to *prolog* to prove a goal. When you have almost finished a proof, you may ask to *complete* it. The system will go all over the partial proof, applying the breadth first tactic to prove any term left to prove. You may also add a *derived rule*, obtained by compactifying a proof tree previously build. You may ask to apply a particular *instanciation* of an object inference rule. Finally, you may perform some 'editing manipulation': you may *cut* or *paste* a proof. Other tactics are special purpose tactics for the Calculus of Constructions. They are described in a forthcoming paper. Notice that all our tactics are *valid tactics.* Unvalid tactics just not seem to belong to our word (the 'Sequent's word').

6.1. *Control*

We have said that for us tactics are parts of the meta-logic, and are implemented by means of inference rules. So the problem arises to specify the *control* that can belong to a tactic when it cannot be written within standard logic. A typical example is the choice among several proofs found, for example, by the 'breadth first' strategy. In this case, we think that the user would like to choose by himself. This possibility is implemented in Theo by using the *debugging* facility of Typol, which enables the user to *control* the execution of its Typol program [9]. [2] In our case, the user may control the execution of the Typol program that describes the breadth first tactic, thus choosing among all the possible proofs that this tactic can build.

Another typical example of a 'control' is the usual if-then-else or repeat *tacticals.* We can specify this kind of tactic by two inference rules. The first rule says 'try tactic 1'. The second rule says 'try tactic 2 if tactic 1 fails'. Then there are two cases. Either we can specify the cases where the first tactic fails, either we cannot -or we could but this would cost too much. In the first case, there is no problem. In the second case, one possible solution is the following one. Theo is compiled into Prolog. Thus the strategy of evaluation of prolog gives some *control* to the tactics. Gilles Kahn compares this *execution* of a logic with the transformation of a set of equational rules into *oriented* rewriting rules. This feature must be used with much care, but may be useful. This is the solution adopted in the theorem prover written in λ-prolog [10].

6.2. *The user tactic*

This tactic consists in asking the user for an inference rule to apply and a place to apply it. It could be named: 'no tactic'. Notice, however, that this 'no-tactic' is the only tactic required by mathematicians for a great amount of their proofs. The user tactic is implemented by one inference rule, with a side condition that realizes the desired interface with the user:

$$\frac{P \overset{\text{apply}}{\vdash} R, x \rightarrow P'}{\text{`user'} \vdash P \rightarrow P'} \text{get}(x, R)$$

The predicate get consist in fact in several predicates. A first predicate asks the user to point out with the mouse on a sequent x, in window *theo-proof*, and gets it. Then a second predicate -set_of_rules- searchs in the object logic involved, the set of rules that are applicable to x. Set_of_rules is the only extra-logical components of Theo. (The search for *a set* is extra-logical.) The resulting rules are displayed in a window named *theo-rules*, where the user is asked to choose a rule. The set apply contains the resolution rules previously presented.

A particular case of the user tactic is the case when there is only one applicable rule. In this case, Theo applies it, without asking the user to choose it, as there is no choice. The tactic *user alone* is a particular

[2] The user can control the execution of its Typol program by 'stepping' in it. At any point he can choose to 'skip' a proof, to refuse a proof (by enforcing a 'fail') or to 'retry' a proof. He can also put 'break points', either in the data or in the 'semantics' programs, to run a proof until a particular point. At any point, the user can see any formula appearing either in the current data or in the current inference rule involved.

version of the user tactic, where the set of the applicable rules to x is not built. The user chooses a rule in its object logic, that Theo displays in a window named *theo-logic*. In this case Theo may fail to apply the rule.

6.3. *The prolog tactic*

In some cases where the 'depth first' strategy is convenient to solve a goal, we would like to solve it in an non-interactive mode, as quickly as possible. If we are not interested in the body of the proof, the best way is to call prolog to solve the goal. This only requires that the user has compiled (button 'Compile' in the Typol menu or function Typol-compile-files or theo-compile-files) into prolog the object rules involved. Theo translates the Typol goal into a prolog one, calls prolog on it, and translates the result back to Typol. This tactic can be used, either automatically or interactively, to prove such 'evidences' as succ, or add, or to perform β-reductions written in Typol for the type-checker of CC for example. In the next version of Theo, the user will be able to tell Theo to treat certains judgements of its object logic as evidences, may be by putting annotations on them. Most of the calls to prolog will then be automatic.

6.4. *Proof of a derived rule, adding a rule*

Theo enables you to proof a sequent. You can also prove a derived rule and add it in your object logic in order to use it later on. The proof of a derived rule is not done by induction, but just by 'compactifying' a partial proof. The tactic *add rule* asks for a partial proof. The system compactifies the proof into a rule that it adds in the object logic involved. (In the current version of Theo, the rule is 'asserted' in the Prolog data base. In the next version, the rule will be added in a list attached to the object logic.)

6.5. *Using an instanciation of a rule*

This tactic, named *inst rule*, enables the user to use an instanciation of an object inference rule, in order to guide a little bit more the construction of its proof. The tactic asks for three arguments: an identifier of an object inference rule, an expression that can be used to instanciate this identifier, and a sequent on which to apply the instanciated rule. A little technical difficulty here, for the user, is that the expression must use the syntax used in a proof, i.e. must be an expression 'seen by Typol'. For example it must be: 'exp_s[atom(id "id", string Logic)]' instead of the -much more readable- pretty-print version of this Typol term: 'Logic'. The easiest way to build this expression is to take it from a proof, using cut, paste and eventually edition of basic expressions. In this case, it is preferable to use the standard pretty-printer of Typol, instead of the 'mixed' pretty-printer, that Theo uses. Another good way to obtain an expression of a language L seen by Typol is to write a translator from the language L to Typol. This has been done for the language CC (see file cc_typol.ty in the directory cc/semantics). The same problem arises when writing a theorem to prove, in particular the initial goal, as we previously noticed when presenting the general organization of Theo.

6.6. *Editing manipulation: cut and paste*

The cut has to forms, called *cut* and *undo*. In both cases, the user designates the sub-proof to cut as usual, with the mouse. To 'cut' this sub-proof means building a new proof where the sub-proof cut has been replaced by a sequent whose values of variables are given by the rest of the proof. To 'undo' a sub-proof is slightly different: the resulting proof is the one the user would have obtained by constructing the same proof, except the cut part. To *paste* a -partial or complete- proof for a sequent to prove consists in proving that sequent by applying the pasted proof as a derived rule, except that the body of the pasted proof is kept in the resulting proof.

6.7. *Special tactics for CC*

Some special purpose tactics have been written for the Calculus of Constructions with local and global constants. They are described in a forthcoming paper.

6.8. *Proof trees*

As we have said in a previous section, the user can ask at any time, for the *proof tree* that corresponds to a proof, or a sub-proof. Theo computes the proof tree by 'pretty-printing' the proof, that is by applying each rule involved, and displays it in a special purpose window.

6.9. *Writing your own tactic*

The user can add his own tactic, writing it as an inference rule, or as a set of inference rules, in Typol. The main inference rule must have the following type:

$$\text{string} \vdash \text{PROOF_ENV} \rightarrow \text{PROOF_ENV};$$

where the string is the name of the tactic being defined. The parameters of the succedent of the sequent denote the proof before and after the application of the tactic. This main inference rule must be added in the file tactic.ty, that contains the definition of all the tactics. The name of the tactic has to be added in the file tactic.ll, where the tactic's box is defined. Annex A gives the abstract syntax of the proofs and of the environments of proofs. The currently implemented basic tools that can be used by a tactic are listed in annex B. We hope that the user of Theo will find the task to write a new tactic an easy task. We hope that Theo is an *open system*.

7. Conclusion

We presented here the first version of a Typol, tactic driven, theorem prover. An evident advantage of Theo is that it is build on top of Centaur, using the man-machine graphic interface of the system. Beside this, we think that Theo as two major advantages. The first one is the very compact form used to represent proofs. The second one is its design: a logic -presented as a Typol program- that manipulates formally an object logic -also given by a Typol program. Note that, on the contrary of most of the theorem provers available now, Theo is not a descendant of LCF, in that in Theo, tactics are not functions. In particular, we do not consider that 'the inverse of an object inference rule is a tactic', as many people do, even including people working in λ-prolog. In principle Typol should be able to define any logic expressed by a Gentzen-type sequent system, and evolves to that end. An important area of research concerns higher-order unification and quantifiers, as for the moment Typol only knows about first-order unification and universal quantifiers.

As far as examples of object logics are concerned, there seems to be two main cases that LF do not handle in an elegant way: the 'rules of proof' and 'Multiple consequence relation' described in [1]. We think that the 'rules of proof' should be naturally represented in sequent calculus. So an interesting example for Theo would be modal logic. 'Multiple consequence relation' seems another kind of system that we can handle well. This will just take advantage of the possibility of using different 'judgements' in a single Typol program (either in different sets of rules or not). Finally, an interesting tactic would be the proof of a derived rule, by induction on the length of the proof. This tactic would in particular realize the proof of the correctness of a translation, in the framework that we defined in [6].

8. Annex A: abstract syntax of a proof

Proofs and 'environments' of proofs are defined by the following abstract syntax, given in the Metal formalism.

```
chapter   PROOF
abstract   syntax
app                  →   RULE_NAME PROOF_S ;
proof_s              →   PROOF  *  ...     ;
x                    →   VALUE CONCLUSION ;
x_s                  →   VAR_TYPE  *  ...  ;
lambda               →   VAR_TYPE_S PROOF ;
PROOF                ::= x rule app lambda proof_s ;
PROOF_S              ::= proof_s     ;
VAR_TYPE             ::= x      ;
VAR_TYPE_S           ::= x_s      ;
PROOF_TREE           ::= x     rule     ;
end   chapter   ;

chapter PROOF_ENV
abstract   syntax
dic                  →   MAP   *   ...      ;
map                  →   ID     EXP     ;
DIC                  ::= dic      ;
MAP                  ::= map      ;
abstract   syntax
p                    →   PROOF    PENV   ;
proof_tree_env       →   PROOF_TREE PENV ;
p_s                  →   PROOF_ENV  * ...   ;
n                    →   DIC    NUMBER    ;
PROOF_ENV            ::= p     p_s     ;
PROOF_TREE_ENV       ::= proof_tree_env    ;
PROOF_ENV_S          ::= p_s      ;
PENV                 ::= n       ;
end   chapter   ;
```

Comments:

The name VAR_TYPE comes from the ELF world, and may be misleading. In a VAR_TYPE, the VALUE is the occurrence of the sequent CONCLUSION. This value is used by Theo to find the desired conclusion in a proof, in order to apply a resolution rule to it. This will be changed in a future version as Centaur now provides a general mecanism of 'access paths'.

The environment (PENV) of a proof has two components: a dictionary and a number. DIC is a dictionary, mapping variables of the proof to their greatest index. (0 means no index). The NUMBER is increased each time a rule is applied. This number may thus be used in a tactic as an unique number.

9. Annex B: Basic tools that can be used by a tactic

We list below the currently implemented basic tools that can be used by a tactic. For each relation, we give the judgement which is defined together with the file name in which it is defined, an example and the functionality realized. To understand this relations, the user needs the abstract syntax of a proof, given in annex A.

1. The two resolution rules

The following function is defined in file apply.ty.

▷ *APPLY*: string, PROOF_ENV ⊢ RULE, VAR_TYPE → PROOF_ENV, EXP, string

Example: $'general', p \vdash R, x \rightarrow p', ids, s$

The application of a rule R to a sequent x in a proof p, in mode 'general' or 'particular', gives a new proof p'. ids are the new occurrences produced.

2. Paste a proof

The following functions are defined in file aux_tac.ty.

▷ *S_PASTE* : PROOF_ENV ⊢ VAR_TYPE, PROOF_ENV → PROOF_ENV

$p(P, n) \vdash x, p \rightarrow p(P', n')$ 'paste' a proof p in x in a proof P, resulting in a new proof P'. The hypothesis is that the underscript of each variable in p is greater than the underscript of the same variable in P. If this cannot be guaranty, the user must use the general function paste below.

▷ *PASTE* : PROOF_ENV ⊢ VAR_TYPE, PROOF → PROOF_ENV

General case of the above function.

3. Proofs and proof trees

The following functions are defined in file theo2.ty.

▷ *TERM_TREE* : ⊢ PROOF_ENV → PROOF_ENV, PROOF_TREE_ENV

Transforms a proof term into its corresponding proof tree, eventually producing a new proof term, instanciation of the given one.

▷ *TREE_TERM* : ⊢ PROOF_TREE_ENV → PROOF_ENV

Transforms a proof tree into its corresponding proof term. This function calls a subpart of its inverse.

4. Rules of the object logic

The following functions are all defined in the prolog file theo_aux.pg, except for NEW_VARS, which is defined in aux_tac.ty;

▷ *add_rule* : (string, string, RULE, EXP, EXP)

(L, N, R, I_p, I_c) add a rule R of name of value N -that is 'string N' or 'Set.Set1...string N'- in the object logic L. The two last arguments are the list of new identifiers in the premises (I_p) and the list of new identifiers in the conclusion (I_c). By new identifier in the premises, for example, we mean an identifier that appears in the premises without appearing in the conclusion. These new identifiers can be obtained by the function NEW_VARS described later on. Add_rule is one of the two extra-logical basic functions of Theo. The new rule is asserted in the Prolog data base. This will probably be changed for an addition in a list. The new version will not cost too much in time as long as the user do not add too many rules in its logic.

▷ *get_rule* : (string, string, RULE)

(L, N, R) gets a rule whose name has for value the string N. The rule is search in the object logic L, either in its static definition or among the rules dynamically added in it, by the function 'add_rule'.

▷ *get_rule1* : (string, RULE_NAME, RULE)

Same function as before, except that it takes a rule_name instead of a string.

▷ $NEW_VARS : \vdash$ TYPOL $\rightarrow$ EXP, EXP

$\vdash R \rightarrow I_p, I_c$ Builds the list I_p of new identifiers in the premises and the list I_c of new identifiers in the conclusion of a given rule.

▷ *new_in_prem* : (string, string, EXP)

(L, N, I_p) gets the new variables of the premises of a rule whose -value of the- name is N, in the object logic L.

▷ *new_in_prem1* : (string, RULE_NAME, EXP)

Same function as before, except that it takes a rule_name instead of a string.

▷ *new_in_conc* : (string, string, EXP)

(L, N, I_c) gets the new variables of the conclusion of a rule whose -value of the- name is N, in the object logic L.

▷ *new_in_conc1* : (string, RULE_NAME, EXP)

Same function as before, except that it takes a rule_name instead of a string.

5. Generalization of a term

The following functions are all defined in file theol.ty. They describe the generalization of a term. Functions *GEN_FOO* -or *GEN_FOO0*- generalize a term of type foo, using a dictionary that maps variables -which are Typol identifiers- of the term to new variables. Except for GEN_RULE, functions having the suffix 0 build the dictionary they need, while functions without the suffix 0 just use the dictionary they have in parameter. By t has type foo, we mean that t is a term, or a subterm of a term of type foo. This will be general in the rest of this annex, each time we say informally that t has type foo, while it is declared of type TYPOL. Typol lacks of this particular notion of sub-type.

Warning: the user should generalize a proof with much care as the generated variables will then match any term. Theo only uses generalisation of a proof as a basic tool for, for example, applying a rule, evaluating a proof into a proof tree, or pasting a proof. In any case a term must be only temporarily generalized, to be instanciated as soon as possible, before the tactic ends. Otherwise the result will still be correct, but may be a little bit surprising.

▷ GEN_RULE : DIC $\vdash$ TYPOL $\rightarrow$ TYPOL, DIC

▷ GEN_PROOF0 : DIC $\vdash$ TYPOL $\rightarrow$ TYPOL, DIC

▷ GEN_PROOF : DIC $\vdash$ TYPOL $\rightarrow$ TYPOL

▷ $GEN_SEQUENT0$: DIC $\vdash$ TYPOL $\rightarrow$ TYPOL, DIC

▷ $GEN_SEQUENT$: DIC $\vdash$ TYPOL $\rightarrow$ TYPOL

6. Instanciation of a term

The following functions are defined in file theol.ty.

▷ $INST$: DIC, DIC, DIC $\vdash$ TYPOL $\rightarrow$ DIC

$d, d_1, pdic \vdash t \rightarrow pdic'$ instanciates a term t, of any type. It uses two dictionaries d and d_1, together with *p_dic* the dictionary attached to the current proof, and eventually updates it. This function calls the two following functions: $d \vdash t$ and $d_1, pdic \vdash t \rightarrow pdic'$.

▷ $INST$: DIC $\vdash$ TYPOL

Instanciates a term using a dictionary that maps particular identifiers of the term to variables.

▷ $INST$: DIC, DIC $\vdash$ TYPOL — DIC

Same function as before, except that the variable is instanciated in the identifier indexed by the successor of the maximal index used for this identifier in the proof. This maximum is the number which is mapped to that identifier in the dictionary attached to the proof. Each time an instanciation is performed, the dictionary of the proof is updated.

7. Dictionaries

The following functions enable to use and modify the dictionaries. They are defined in file dic.ty.

▷ *DIC* : DIC ⊢ ID ↦ TYPOL

Gives the value associated to an identifier in a dictionary. In the case where the identifier is not found, the result is this identifier.

▷ *DIC0* : DIC ⊢ ID ↦ TYPOL, DIC

Same function as before, except that in the case where the identifier is not found, the result is a variable, and the dictionary is updated with the new resulting mapping.

▷ *DIC1* : DIC ⊢ ID ↦ TYPOL

Same function as before, except that in the case where the identifier is not found, the result is string 'not_found'.

▷ *UPDATE_DIC* : DIC ⊢ ID, TYPOL → DIC

Updates a dictionary with a new mapping. If the identifier does not yet belong to the dictionary, the new mapping is just added to the dictionary.

▷ *UPDATE_DIC1* : EXP ⊢ PROOF_ENV → PROOF_ENV

Decreases the number associated to the identifier in the list -if any- by one. Removes the mapping involved if the number is 0.

▷ *S_APPEND_DIC* : ⊢ DIC, DIC = DIC

Append two dictionaries. Takes as hypothesis that the dictionaries do not overlap.

▷ *APPEND_DIC* : ⊢ DIC, DIC → DIC, DIC, DIC

General case of the above function. Append two dictionaries. In the case of overlaping, the number involved is increased to the sum of the two numbers. Gives the resulting dictionary together with a dictionary dic_gen and a dictionary dic_inst that can be used to 'shift' the identifiers of a proof.

▷ *SUB_DIC* : ⊢ DIC, EXP → DIC

Gives the sub-dictionary concerning the given list of identifiers.

8. Strings

The following functions are defined in the prolog file theo_aux.pg.

▷ *mk_id* : (string, integer, ID)

Takes a string s and an integer i as arguments, and builds the identifier id s.i whose value is the concatenation of s, '.' and i.

▷ *mk_id0* : (integer, integer, ID)

Takes two integers i and j has arguments and builds the identifier xi.j.

▷ *mk_string* : (string, integer, string)

Takes a string s and an integer i as arguments, and builds the string s.i.

9. Other basic relations

▷ *set_of_rules* : (string, string, CONCLUSION, RULE_S), defined in theo_aux.pg.

Gives the set of rules of a given object logic that are applicable to a given sequent, in a given mode. One of the two extra-logical basic tools of Theo. It calls the relation GET_RULE_MATCHING, which is written in pure logic.

▷ *GET_RULE_MATCHING* : string, string ⊢ CONCLUSION → RULE, defined in theol.ty.

$L, M \vdash S \rightarrow rule$ gets one applicable rule to the sequent S in the object logic L. M is the mode (general or particular) to be used.

▷ *NOT_IN_SEQ* : ID ⊢ TYPOL → EXP, defined in theo1.ty.

This function tells if an identifier is in a sequent or not. If the identifier does not appear in the sequent, it returns the list containing the identifier, otherwise it returns an empty list of identifiers.

▷ *VARS* : ⊢ TYPOL → VAR_TYPE_S, EXP, defined in theo2.ty.

Gives the list of the remaining terms to prove in a given proof, with their associated value.

▷ *SUBST* : ⊢ TYPOL, PROOF, PROOF → TYPOL, defined in theo2.ty.

$\vdash P, x, p \rightarrow P'$ performs the substitution of a proof p for a term to prove -or a proof- x in a proof P. This is a basic function, it does not do any verification. So the user may temporarily build incorrect proofs. Theo will stop as soon as it will try to evaluate the corresponding proof tree.

▷ *HYPS* : ⊢ TYPOL → PREMISE_S, defined in aux_tac.ty.

Gives the list of the remaining terms to prove in a given proof. This function is similar to the above function, but it builds a list of premises, each one being a term to prove, instead of a list of term to prove.

▷ *HYPS_T* : ⊢ TYPOL → PREMISE_S, defined in aux_tac.ty.

Same function as above, but takes a proof tree instead of a proof term.

▷ *SUBST_EXP* : ⊢ TYPOL, EXP, EXP → TYPOL, defined in aux_tac.ty.

$\vdash t, e, e_1 \rightarrow t'$ realizes the substitution of e by e_1 in a conclusion t, resulting in a new term t'.

10. Functions of the interface

The prolog functions that realyze the interface between Theo and the user are not listed here. They are defined in the file theo_int.pg in the directory theo/semantics. The file theo_int.pg will be a good help for the user who desires to add a function in the interface to ask for arguments for a new tactic. We list here the basic prolog functions that are used by Theo to realize the interface with the user. They are defined in the prolog file jd_resources.pg and use lisp functions defined in the file resources.ll, in the directory contrib/resources.

▷ *ask_name*: (string, integer)

ask_name(Name, Mess) send the message of number Mess -written in the file theo.mess- asking for a string Name.

The following functions are various kind of gettree and sendtree. In the case where they are relative to a typol-object, this typol-object must exist, and must contain at least a variable (see examples in the file 'theo.ll').

▷ *gettree*: (string, path, _)

gettree(To,_,T) gets the current tree T of the ctedit attached to the Typol object To.

▷ *rgettree*: (string, path, _)

rgettree(To,_,T) gets the root tree T of the ctedit attached to the Typol object To.

▷ *ogettree*: (string, path, _, string)

ogettree(To,_,T,Op) gets the tree T of type Op that is surrounding the current tree of the ctedit attached to the Typol object To. This function enables the user to click on any sub-expression of the term of a given type -for example a rule- he wants to get.

▷ *ogettree_proof*: (string, path, _)

ogettree_proof(To,_,T) is ogettree(To,_,T,"x" or "app")

▷ *ogettree_up*: (string, path, _, string)

Same function as ogettree, except that it goes up on the current tree before looking for the operator, thus always giving a strictly surrounding tree.

▷ *ogettree_s*: (string, path, _, string, integer)

ogettree_s(To,_, T, Op, n) is ogettree(To,_,T,Op) followed by (tree:down n) to get the nth son of the resulting tree.

To almost each one of the previous various kind of gettree functions correspond the same function having no typol object as argument. These various functions take as argument the ctedit memorized in a global lisp variable. The idea is that the user has previously affected this variable to a ctedit pointed to by the mouse. (See file resources.ll for the definition of the functions involved and file theo_int.pg for there use.) The various gettree functions are listed bellow:

▷ *mgettree*: (path, _), similar to gettree.

▷ *rmgettree*: (path, _)

▷ *omgettree*: (path, _, string)

▷ *omgettree_proof*: (path, _)

▷ *rmgettree_s*: (path, _, integer)

rmgettree_s(_, T, n) is rmgettree(_,T) followed by (tree:down n) to get the nth son of the resulting tree.

▷ *sendtree*: (string, _)

sendtree(To, T) sends the tree T into the ctedit attached to the Typol object To, and redisplays the window. If the ctedit does not exists yet, the function creates it. The user can initialize such arguments as the width and height of the ctedit, by using the 'put-property' Centaur function.

10. Annex C: How to install the system.

Theo is runing under Centaur, so you first have to install Centaur. If you have a Centaur system with the theorem prover, the directory 'contrib' contains the file 'Buildfile' together with the following directories: theo, resources, fonts and eventually cc. If that is not the case, just send a mail to the author of this manual, asking for a tape containing these directories.

Then add the following lines in your .centaur:

```
(setq dir0 DIR)
(# :centaur:user-language 'theo (catenate dir0 "/contrib/theo/"))
; Two lines for the Calculus of Constructions:
(# :centaur:user-language 'ccsem (catenate dir0 "/contrib/cc/cc/"))
(# :centaur:user-language 'cc (catenate dir0 "/contrib/cc/syntax/"))
; Then some resources:
(# :centaur:user-language 'resources (catenate dir0 "/contrib/resources/"))
(loadfile (catenate # :centaur:theo-directory "theo.ll") t)
(loadfile (catenate dir0 "/contrib/resources/resources.ll") t)
```

where the directory DIR is the directory containing the directory 'contrib'.

Finally, do 'ctmake' in the directory theo/semantics, and eventually in the directory cc/cc/semantics, if you are interested in this example.

References

[1] A. AVRON, "Simple Consequence Relations" Edinburgh Report ECS-LFCS-87-30, June 1987.

[2] A. AVRON, F. HONSELL, A. MASON, "Using typed λ-calculus to implement formal systems on a machine", Edinburgh Report ECS-LFCS-87-31, July 1987.

[3] J. BARWISE, "The Handbook of Mathematical Logic", North-Holland, Amsterdam, 1977, reprinted in 1983.

[4] TH. COQUAND, "An analysis of Girard's paradox" Proc. of the first ACM-IEEE Symp. on Logic In Computer Science, Cambridge, Ma, USA, June 1986.

[5] R. L. CONSTABLE and all, "Implementing Mathematics with the Nuprl Proof Development System", Prentice-Hall, 1986.

[6] J. DESPEYROUX, "Proof of translation in natural semantics", Inria Research Report 514, april 1986. also in the Proc. of the first ACM-IEEE Symp. on Logic In Computer Science, Cambridge, Ma, USA, June 1986.

[7] J. DESPEYROUX, "First experiments with theorem proving in Centaur: the Calculus of Constructions, the Edinburgh Logical Framework, and Theo", in "Esprit project 348 GIPE. Third annual review report", January 1988.

[8] J. DESPEYROUX, "Theo: an interactive Typol theorem prover", Inria Research Report 887, August 1988.

[9] TH. DESPEYROUX, "Typol: a formalism to implement Natural Semantics", Inria Technical Report 94, mars 1988.

[10] A. FELTY, D. MILLER, "Specifying theorem provers in a higher-order logic programming language", Ninth Conference on Automated Deduction, 1988, and Report MS-CIS-88-12, University of Pennsylvania, February 1988.

[11] G. GENTZEN, "The Collected Papers of Gerhard Gentzen", edited by M. E. Szabo, in studies in logic and the foundations of mathematics, North-Holland, Amsterdam, 1969.

[12] T. G. GRIFFIN, "An Environment for Formal Systems", Report 087-022005, Cornell University, August 1987.

[13] R. HARPER, F. HONSELL, G. PLOTKIN, "A Framework for defining Logics", Proc. of the second ACM-IEEE Symp. on Logic In Computer Science, Cornell, USA, 1987.

[14] L. HASCOËT, "A tactic-driven system for building proofs", Inria Research report 770, Dec. 1987. also in the proc. of the 7th seminar "Programmation en Logique", Tregastel, May 1988.

[15] G. HUET, "A uniform approach to Type Theory", Inria Research Report 795, February 1988.

[16] G. KAHN, "Natural Semantics", Proc. of Symp. on Theorical Aspects of Computer Science, Passau, Germany, February 1987, also Inria Research Report 601, Feb. 1987.

[17] D. MILLER, G. NADATHUR "A logic programming approach to manipulating formulas and programs", Report MS-CIS-87-113, University of Pennsylvania, December 1987.

[18] G NADATHUR, "A higher-order logic as the basis for logic programming", Ph.D. dissertation, University of Pennsylvania, Dec. 1986, also Report MS-CIS-87-48, University of Pennsylvania, June 1987.

[19] L.C. PAULSON, "Logic and computation. Interactive proof with Cambridge LCF", Cambridge Tracts in Theoretical Computer Science 2, 1987.

[20] L.C. PAULSON, "The representation of logics in higher-order logic", Cambridge Technical Report 113, 1987.

[21] L.C. PAULSON, "The foundation of a generic theorem prover", Cambridge Technical Report 130, march 1988.

[22] G.D. PLOTKIN, "A structural approach to operational semantics", Aarhus Report DAIMI FN-19, 1981.

[23] D. PRAWITZ, "Natural Deduction, a Proof-Theoretical Study", Almqvist & Wiksell, Stockholm, 1965.

[24] D. PRAWITZ, "Ideas and results in Proof Theory", Proc. of the 2nd. Scand. Logic Congress, North Holland, 1971.

Table of Contents

1. Introduction. Original motivations and Natural Semantics background . . . 2
2. The theorem prover . . . 2
   2.1. Developing a proof under Theo. General organization of the system . . . 2
   2.2. Representation of a Proof . . . 3
3. A simple example of an object logic . . . 3
4. An example of a proof development . . . 4
5. Two Resolution Rules . . . 5
   5.1. General proofs . . . 5
   5.2. Particular proofs . . . 5
6. Tactics . . . 6
   6.1. Control . . . 6
   6.2. The user tactic . . . 6
   6.3. The prolog tactic . . . 7
   6.4. Proof of a derived rule, adding a rule . . . 7
   6.5. Using an instanciation of a rule . . . 7
   6.6. Editing manipulation: cut and paste . . . 7
   6.7. Special tactics for CC . . . 7
   6.8. Proof trees . . . 7
   6.9. Writing your own tactic . . . 8
7. Conclusion . . . 8
8. Annex A: abstract syntax of a proof . . . 9
9. Annex B: Basic tools that can be used by a tactic . . . 10
10. Annex C: How to install the system. . . . 14

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique