Theo: An interactive proof development system.

Joëlle Despeyroux

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

This paper [1] presents an interactive, tactic-driven, proof development system -a Theorem Prover- called *Theo*. 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. Tactics and tacticals are 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.

Subject classification: CR D.1, F.3, F.4.1.

Keywords: Software: programming languages, Theory of computation: logics and meanings of programs, Mathematical logic and foundations: mechanization of proofs, logic of programming, Theory of computing: symbolic computation, semantics, Artificial intelligence: theorem proving.

1. Introduction. Original motivations and Natural Semantics background

A Natural Semantics specification [7][18] is a formal system, that may be a logic or not, depending on the example. The idea of using a formal system to describe semantics of languages comes both from G. Plotkin's Operational Semantics [24]and unpublished notes of Robin Milner. A Natural Semantics specification is defined by means of axioms and inference rules, using Gentzen's sequents [12][3]. Natural deduction style is not handled, except as a pretty-print of a logic defined by a Gentzen-type sequent system -i.e. with explicit hypothesis. 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 [10]. For the moment, Typol is a first order language. A Typol program can be type-checked and compiled -presently into Prolog or Lisp. It is usually 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

[1] Received February 1990. Revised April 91.

Gerard Huet, for example. For these examples, we need an *interactive theorem prover*. In fact, the need of an interactive theorem prover even goes back to [7], 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.

The paper consists of seven parts. We first present 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 presented in the fifth part. The sixth part presents various examples of object logics. Finally, we discuss the connections with related logical frameworks.

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. Tactics are functions or relations from proofs to proofs. As they are parts of the meta-logic, it seems natural 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. As typol is for the moment a first order language, the meta logic of Theo is a first order logic, that -beside tactics- mainly contains the definition of a proof term, and the resolution rule used to develop proofs. 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. The first versions of Theo were presented in [8][9].

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

Theo first asks for a sequent to prove, 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 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. You are allowed to stop to build 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 into 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 object logics are defined by a Gentzen-type sequent system, and 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 [9]. [2] A proof is recursively defined by:

- $R : T$, where T is a sequent of the object logic -of the form $\phi \vdash \psi$, with ϕ and ψ ranging over formulae. *axiom or theorem of name R of the object logic - 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.

---

[2] These two styles of definition seem to be quite standard now. Even using a resolution rule seems to be the natural way to build proofs. Th. Coquand, for example was easily convinced, and implemented these two ideas in his theorem prover for the Calculus of Constructions. [5]. The combinators are the nodes used for the application and a term to prove, together with the names of the object inference rules.

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 later on.

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, together with its pretty-printer. The abstract syntax of the language, given as an algebraic abstract data type, is as follows:

```
sort  prop  is
id,  true,  false   : () → prop;
not                 : (prop) → prop;
implies, and, or    : (prop, prop) → prop;

sort  props  is
props               : (prop, props) → props;
eprops              : () → props;
end;
```

The inference rules we chose are the propositional part of the intuitionist Gentzen's Sequents System for First Order Logic, called 'System LJ' [12], 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 props $\vdash$ prop. The inference rules of Fig.1 are the pretty-print version of the Typol inference rules.

$$\Rightarrow_s: \frac{a, p \vdash b}{p \vdash a \Rightarrow b} \qquad \Rightarrow_a: \frac{p \vdash a \quad b, p \vdash c}{a \Rightarrow b, p \vdash c}$$

$$and_s: \frac{p \vdash a \quad p \vdash b}{p \vdash a \wedge b} \qquad and_{al}: \frac{a, p \vdash c}{a \wedge b, p \vdash c} \qquad and_{ar}: \frac{b, p \vdash c}{a \wedge b, p \vdash c}$$

$$or_{sl}: \frac{p \vdash a}{p \vdash a \vee b} \qquad or_{sr}: \frac{p \vdash b}{p \vdash a \vee b} \qquad or_a: \frac{a, p \vdash c \quad b, p \vdash c}{a \vee b, p \vdash c}$$

$$not_{sj}: \frac{a, p \vdash \mathit{false}}{p \vdash \neg a} \qquad not_{aj}: \frac{p \vdash a}{\neg a, p \vdash \mathit{false}}$$

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

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-deterministic 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 an '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 built 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 displays 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:

$$\begin{aligned}(\Rightarrow_s (or_a \ \ (and_s \ \ (or_{sl} \quad ax{:}x \quad \vdash \quad x) \quad (or_{sl} \quad ax{:}x \quad \vdash \quad x))\\ (and_s \ \ (or_{sr} \ (and_{al} \ ax{:}y \vdash y)) \ \ (or_{sr} \ (and_{ar} \ ax{:}z \vdash z)))))\end{aligned}$$

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, where a and b are variables whereas y is a closed term:

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

5. A Resolution Rule

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 consists in two Typol rules, which can be pretty-printed as follows (technical details are omitted):

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

The rules above describe the application of an inference rule R to a sequent c in a proof P. Remember that all these terms are -*formal*- Typol terms, with Typol identifiers denoting 'variables'. In the first rule, the sequent c, the proof P and the rule R are 'generalized'. (This means that Typol identifiers are replaced by logical variables, which can later on become any Typol term.) [3]

Then the new proof resulting from the application is 'instantiated' -i.e. logical variables are replaced by new Typol identifiers- (only possibly new variables appearing in the premises of rule R are still logical variables in the resulting proof, and are to be instantiated).

The second rule realizes the application of an inference rule $\frac{p}{\phi}$, of name n, to a sequent ψ in a proof P. First ϕ and ψ are unified in c -this unification is or is not just matching, depending on the case. Note that the unification affects the all proof P, thus possibly instantiating free variables in it. 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 p of new terms to prove, 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.

6. Tactics.

The basic implemented tactics are *user*, and the direct call to *Prolog* to prove a goal. You may add a *derived rule*, obtained by compactifying a proof tree previously built. You may also ask to apply a particular *instantiation* of an object inference rule. Finally, you may perform some 'editing manipulation': you may *cut* or *paste* a proof. A function *complete?* tells you if your proof is complete or not. Other tactics are special purpose tactics for the Calculus of Constructions. They are described later on. Notice that all our tactics are *valid tactics*. Unvalid tactics just do 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 [10]. [4] 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*. There are two cases. Either we can specify the cases where a tactic fails, dor we cannot, or we could but this would cost too much. In the first case, there is no problem. In the second case, there is an inference rule in the code whose order of the premises is relevant. 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 [11].

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:

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

[3] The generalization of a Typol identifier, written in the above rule $gen(ident\ I)$, is specified by the following Typol sequent: $e \vdash ident\ I \rightarrow x$, where e is an environment mapping Typol identifiers to Typol terms. Of course, c and P must be generalized in the same environment. In order to optimize this process a little bit, we only generalize in P those identifiers that appear in c.

[4] 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.

The predicate *get* consists in fact in two predicates. A first predicate asks the user to point out with the mouse on a sequent c in the proof, and gets it. Then the user is asked to choose a rule R in the object logic. The set *apply* contains the resolution rule presented in the previous section. Of course the system may fail to apply the rule.

The function *Help user* uses the predicate set_of_rules to search for the inference rules that are applicable to the selected goal. 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 special purpose window.

6.3. *The Prolog tactic.*

In some cases where the 'depth first' strategy is convenient to reach a goal, we would like to use it in a 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 problem. This only requires that the user has compiled 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 perform some arithmetic calculus, or 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 call the Prolog tactic on some specific judgements of its object logic, may be by putting annotations on them. Most of the calls to Prolog will then be automatic.

The *exact* function uses the Prolog tactic to check the validity of a (possibly partial) theorem given by the user. The theorem, eventually instantiated, is added to the object logic concerned.

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 proof-tree into a derived rule, taking the premises and the conclusion of the proof-tree. The tactic *add proof* asks for a proof. The system 'compactifies' the proof into a rule that it adds into the object logic involved. In the current version of Theo, the rule is 'asserted' in the Prolog data base. It should be added in a list attached to the object logic. In order to use the new rule later on, the user should add it to the object logic concerned.

6.5. *Using an instantiation of a rule.*

This tactic, named *inst rule*, enables the user to use an instantiation of an object inference rule, in order to guide a little bit more the construction of its proof. This tactic asks for three arguments: a variable of an object inference rule, an expression that can be used to instantiate this variable and a sequent on which to apply the instantiated rule. A little technical difficulty here, for the user, is that the expression must apply 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. 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 eq and CC. 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, or undo, and paste.*

To 'cut' or *undo* a sub-proof means first to build a new proof where the proof cut has been replaced by a sequent whose values of variables are the most general possible, depending both on the rest of the proof and on the initial goal. Then other parts of the proof may depend on the cut part. These terms are also replaced by variables. A second version of the undo, *undo1*, is undo, except that theorems remain unchanged, even if some terms in them are depending on the cut proof. In both versions, the resulting proof of an undo is one of the possibly various proofs that the user would have obtained by playing the same proof, except the cut part.

To *paste* a partial or complete proof for a sequent to prove -the goal- is equivalent to first evaluating the derived rule corresponding to the proof to paste, then applying this rule to prove the goal, except that the body of the pasted proof is kept in the resulting proof. Keeping the body of the pasted proof entails renaming of variables.

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 later on.

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:

$$\mathrm{NAMED}, \mathrm{PROOF\_ENV} \vdash \mathrm{tactic} \rightarrow \mathrm{PROOF\_ENV};$$

For example, $i, p \vdash T(a, b) \rightarrow p'$ describes a tactic T that takes a and b as parameters, takes a proof p as input and gives a proof p' as output, where i is the initial goal. A user defined tactic is valid, as long as it does not use a basic function as generalizing a proof, or substitutes a term for another in a proof. To ensure the validity of a proof, the user may evaluate its corresponding proof-tree. This is not done automatically by the theorem prover as this operation costs a little bit in time. But it would be easy to enforce this check on a result of a tactic, before giving it. We hope that the user of Theo will find the task of writing a new tactic an easy one. We also hope that Theo is an *open system.*

7. Tacticals.

A mini-language has been defined to write tacticals. Constants of the language are tactics. However most tactics are non-constant operators, as user, undo or proof-tree. Other operators are the usual operators *sequence, try, if-then-else* and *repeat.* An operator *do* allows to apply a tactic and then forget the Prolog proof performed, in order to save Prolog memory. This is just an implementation trick. The language has no concrete syntax as most operators take as arguments Typol expressions. For example, *user* takes two arguments: a sequent to prove and a rule to apply.

8. Some examples of object logics

We present here various examples of object logics, that are implemented under the system.

8.1. *The Calculus of Constructions and the Edinburgh Logical Framework*

A minimum proof development environment for the Calculus of Constructions (CC) of Gérard Huet and Thierry Coquand [4][16]was presented in [9]. We have now implemented some special purpose tactics for CC. Together with our general purpose tactics, they represent most of the tactics described by Thierry Coquand in [5]. The first tactic, *exact,* is in fact a general purpose tactic. It asks for a theorem, in CC in this case. Theo verifies the validity of the theorem, eventually instantiating it, then uses it to prove the current goal. The second tactic, called *var,* searches for a variable in the current context having the type of the current goal. A third tactic, *prod,* is a *repeat* 'tactical'. It consists in the application of the abstraction rule as long as possible. This tactic is used as a starting point when building a proof of a product. Adding those tactics for CC was a very easy task, suggesting that our basic tools begin to be 'sufficiently complete'. Finally we have implemented the declaration of local and global constants in CC. Local constants are produced by a *let* rule in the type-checker of CC. Global constants are declared in a *signature.* Global constants are given by their definition, eventually together with a type. The validity of a signature must be checked. Then the signature must be 'compiled', producing some code that Theo will automatically use whenever it encounters a constant in an expression, or whenever a tactic requires to look for a constant as a possible expression having the desired type of the current goal. A theorem to prove in CC contains as arguments both a context and a name of a signature.

The Edinburgh Logical Framework (LF) [14][2]has different aims than CC. It is a Logical Framework whereas CC is a Mathematical Framework, and is not to be confused with it. However, they are for Theo very similar languages, with similar type-checkers. So we defined a common language for them, a λ-calculus with dependant product and a let construct. Very quickly a minimal environment for LF was obtained from its type-checker written in Typol. As the tactics written for CC are naturally useful for LF, we now have an environment for LF similar to the one for CC. The main difference are signatures (global constants): a constant in CC is given by its definition and type while a constant in LF is only given by its type. We should be able to handle any 'Generalized Type System'.

8.2. *Propositional Logic*

The propositional part of First Order Logic was easy to implement. On the contrary, First Order Logic is the typical example that Theo could handle, but other systems like LF or λ-Prolog handled in a more elegant way. The point is that in Theo, as in Isabelle for example, we would have to encode the side conditions of the 'quantifiers' rules. LF-like systems avoid these side conditions, as they are based on a λ-calculus specially designed to avoid them. We plan to base Typol on a λ-calculus too, with higher-order matching, but we have to do that with much care. The 'rules of proof' presented in [1]for example, are not handled in an elegant way by LF.

8.3. *Proving a derived rule in a logic*

We said that we can prove a derived rule within Theo. This 'tactic' is a general purpose tactic, that can be used in any object logic written in Typol. A very simple example would be to compare the static semantics of the 'lambda' and the 'let' construct in ML. Another example is to derive inference rules defining the 'repeat' construct from its definition with respect to the 'while' and from the inference rules defining the while. But we guess the user will have many more interesting examples to deal with.

8.4. *Execution of non deterministic languages*

The integration of Theo and the debugging facility of Typol that we have mentioned above, when we discussed control in tactics, seems to give rise to many possibilities for developing a proof, especially when dealing with true non-deterministic languages. We intend to work on an example in this area very soon.

8.5. *A Rule-base definition of Equational Unification*

This is the typical example that Theo handles well, as the inference rules form a formal system written with sequents, with no side conditions on free variables [17].

9. Related work

Related work are numerous, and most of them are quite recent systems. LCF [21]was among the first theorem provers. ML was defined as its meta-language for writing tactics. Terms, formulae and theorems are typed data. An inference rule is defined as a function from theorems to theorems. Its 'inverse' is a tactic mapping a goal to subgoals. *Tacticals* express functions on tactics, such as repetition of a given tactic. In order to obtain the proof just constructed, a *validation function* has to be evaluated; this is now universally recognized as a serious fault. Theorems can only be constructed by tactics. Still, a tactic can be invalid. In this case the validation function will not return the theorem asked for. LCF has its own object theory, called PPλ, and does not in principle enable the user to enter its own object logic.

Nuprl [6]was among the first systems being implemented and is still the most sophisticated system. It is based on a Constructive Type Theory, that mainly extends Martin-Löf's Intuitionistic Type Theory with recursive types and partial functions. So a proof in Nuprl is a term of a higher-order λ-calculus. Tactics are expressed by ML functions, as in LCF. Nuprl provides a user interface, less powerful than Centaur.

λ-prolog [19][20]is currently developed by the D. Miller team. It is a very interesting extension of Prolog to allow λ-terms and applications inside a term of a clause. λ-terms are evaluated by the usual β-reduction. Unification has been extended to higher order unification. λ-prolog has recently been used to implement a theorem prover [11]. Proofs are represented exactly as our proofs. But the theorem prover is not viewed as a meta-logic manipulating an object logic. An inference rule of the object logic is defined as a tactic, more precisely a λ-prolog clause saying that the proof of the conclusion will be such provided proofs of the premises are such and such. Tactics have exactly the same shape. In a certain sense, there is only a meta-logic. We are not following this approach.

Just after we begun to design Theo, Laurent Hascoët[15], from our team, tried to do some experiments in theorem proving under Centaur. His theorem prover now consists in prolog clauses that formally manipulates Typol rules. So far the differences between his system and Theo could be seen as minor, as far as design is concerned, as Theo consists of Typol rules that formally manipulates other Typol rules. Note, however, that writing Theo in Typol enables us to fit in with the idea that the theorem prover is nothing else but the meta-logic in which object -or meta- proofs can be performed. Another difference is the form chosen for representing a proof. His system keeps the full proof tree, which can be very large, whereas Theo defines its own form of a proof, as all others systems do. Our systems have been implemented independently. He mostly used his system to perform 'partial evaluation' on Typol program.

The Calculus of Constructions, Isabelle, and the Edinburgh Logical Framework are three recent systems. CC, the Calculus of Constructions [4][16], is aimed at formalizing mathematical reasoning, while ELF, the Edinburgh Logical Framework, and Isabelle are general frameworks to represent logics. While a proof in CC and ELF is a term of a higher-order λ-calculus, a proof in Isabelle is a proposition of higher-order logic, representing a derived rule of the object logic.

ELF [14] is aimed at allowing uniform presentations of logics. Such operations as introduction and discharge of hypothesis, instantiation of schematic variable, or certain variable-occurrence conditions are 'built in' the framework. According to designers of ELF [2], 'it is an 'ELF thesis' that well-behaved natural-deduction formalisms are those that can be directly encoded in the ELF'. The price to pay is to prove that the object logic has been correctly translated. This can mean for example that 'there is a compositional bijection between proofs in the natural deduction system and proof terms in ELF'. On the other hand, it seems that ELF does not succeed naturally to represent 'rules of proof'. The rule of necessitation of modal logic, for example, seems to be more naturally presented in a sequent calculus framework. A prototype version of ELF has been implemented under the Cornell's Synthesizer by T. Griffin.

EFS [13] is a theorem prover implemented under the Cornell Synthetizer, which was initially designed to build proofs in LF and CC. EFS has in common with Theo the fact that it is implemented under a meta-compiler that provides a nice man-machine interface. But object logics must be defined by means of attribute grammars.

Isabelle [22][23] is based on higher-order logic. A proof, or more exactly the 'assertion of provability in the object logic', is a proposition of the meta logic -higher-order logics-, that represents a derived rule of the object logic. The application of a rule is defined by a Resolution Rule. This rule, as well as the 'lifting' rule can be derived in the meta logic. A rule of the object logic is represented by a term in the typed λ-calculus. Tactics are expressed by ML functions. It is due to the use of unification in the process of building proofs that no validation function is needed, despite the use of ML. Maybe we should say here that we designed Theo just before reading the last reports on Isabelle. If we ever choose Isabelle proofs, our resolution rule could be the same than the one of Isabelle (the one presented p.28 in [23]). The only remaining significant difference will then be on the form of tactics, which, in Theo, are expressed in Typol. However, we do not think that we will change our representation of proofs to the one of Isabelle, because of its following drawback: as Isabelle does not keep the full proof, it seems a-priori not possible to do some calculus on the proof.

10. Conclusion

We presented here the first version of a Typol, tactic driven, theorem prover. The main advantage of Theo is its general nature: the user has only to give the object logic's concrete and abstract syntax to Centaur and to write the inference rules in Typol to get a theorem prover for an object logic. The only proviso is that Typol is well suited for describing the rules. So the typical examples that Theo handles well, are formal systems written with sequents, with no side conditions on free variables.

An evident advantage of Theo is that it is built on top of Centaur, using the man-machine graphic interface of the system. Beside this, we think that Theo has two major advantages. The first one is the very compact form used to represent proofs. The second is its design: a logic, presented as a Typol program, that manipulates formally an object logic, also given by a Typol program. Note that, contrary to 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 does 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 to be 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 [7].

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] Th. Coquand, *The tactics Theorem Prover, User's guide", in the "Calculus of Constructions. Documentation and Users' guide, Version 4.10", July 1st 1989.*

[6] R. L. Constable *et al., Implementing Mathematics with the Nuprl Proof Development System, Prentice-Hall, 1986.*

[7] 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.*

[8] 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.*

[9] J. Despeyroux, *"Theo: an interactive proof development system", Inria Research Report 887, August 1988.*

[10] Th. Despeyroux, *Typol: a formalism to implement Natural Semantics, Inria Technical Report 94, mars 1988.*

[11] 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.*

[12] 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.*

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

[14] 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.*

[15] 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.*

[16] G. Huet, *A uniform approach to Type Theory, Inria Research Report 795, February 1988.*

[17] J.P. Jouannaud, C. Kirchner, *Solving Equations in Abstract Algebras: A Rule-Bases survey of Unification. March 31, 1990*

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

[19] D. Miller, G. Nadathur *A logic programming approach to manipulating formulas and programs, Report MS-CIS-87-113, University of Pennsylvania, December 1987.*

[20] 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.*

[21] L.C. Paulson, *Logic and computation. Interactive proof with Cambridge LCF, Cambridge Tracts in Theoretical Computer Science 2, 1987.*

[22] L.C. Paulson, *The representation of logics in higher-order logic, Cambridge Technical Report 113,*

1987.

[23] L.C. Paulson, *The foundation of a generic theorem prover, Cambridge Technical Report 130, March 1988 and Journal of Automated Reasoning, vol 5, 1989, pp. 363-397.*

[24] G.D. Plotkin, *A structural approach to operational semantics, Aarhus Report DAIMI FN-19, 1981.*

[25] D. Prawitz, *Natural Deduction, a Proof-Theoretical Study, Almqvist & Wiksell, Stockholm, 1965.*

[26] D. Prawitz, *Ideas and results in Proof Theory, Proc. of the 2nd. Scand. Logic Congress, North Holland, 1971.*

Table of Contents