TYPOL : a formalism to implement natural semantics

Thierry Despeyroux

Rapports Techniques

N° 94

TYPOL

A FORMALISM TO IMPLEMENT NATURAL SEMAMTICS

Thierry DESPEYROUX

MARS 1988

RT94

TYPOL

A Formalism to Implement Natural Semantics

Un Formalisme pour Implémenter la Sémantique Naturelle

Thierry Despeyroux

INRIA – Sophia-Antipolis
2004, Route des Lucioles
F-06565 Valbonne Cedex (France)

Abstract

CENTAUR is an interactive programming environment generator. It allows the specification not only of the syntactical aspects of a programming language, but also of its semantical aspects. Those aspects are described using a specification formalism called TYPOL. The specifications written in TYPOL may be compiled into Prolog to be executed. This report is the TYPOL (version 2) reference manual for the version 0.5 of CENTAUR.

Résumé

Le système CENTAUR est un générateur d'environnement de programmation interactif. Il permet de spécifier non seulement les aspects syntaxiques d'un langage de programmation, mais aussi ses aspects sémantiques. Ces derniers sont décris à l'aide d'un langage de spécification sémantique, TYPOL. Les spécifications écrites en TYPOL peuvent être compilées en Prolog afin d'être exécutées. Ce rapport est le manuel de référence TYPOL (version 2) de la version 0.5 de CENTAUR.

TYPOL

A Formalism to Implement Natural Semantics

Thierry Despeyroux

INRIA – Sophia-Antipolis
2004, Route des Lucioles
F-06565 Valbonne Cedex (France)

♠

Manual for the version **0.5** of CENTAUR.

Contents

Introduction

To date, four main lines have been explored in the formal specification of programming languages: *denotational semantics* – especially to specify dynamic semantics –, *attribute grammars* – mostly for static semantics and translations –, *algebraic abstract data types* and *structural operational semantics*.

All these semantics (except the last one) have a great default: they lead to very large specifications, not very easy to manage, not even to write.

We have developed a new semantics, calling it *natural semantics*. The main idea of it is that computing the semantic value of an expression of a language is no more than proving a particular theorem in a theory (made of axioms and inference rules) that express the semantics of this formalism. Examples of such predicates are: *"this expression has this type"*, or *"this expression in language L may be translated into that expression in language L' "*.

• *Natural semantics* retains the best aspects of earlier methods. It has its origin in the structural operational semantics [13], but focus on the pure logical part of it. The name *natural* comes from the fact that we write our semantic definitions in the *natural deduction style* [14,15], using Gentzen's sequents [8]. As in denotational semantics and in structural operational semantics, semantics may be defined recursively on the structure of the formalism. But we can also write non structural definitions. The style of this definition does not need to be operational (but it can be operational), and it is no longer functional, but relational. Notice that conditional rewriting rules may be expressed using natural semantics, but rewriting is a particular example of what can be expressed in this semantics. As in attribute grammars, algebraic data types, and in B.N.F., the definition style is declarative. Natural semantics also uses very powerful concepts: pattern matching, unification and overloading.

We would like natural semantic specifications to be as close as possible to traditional mathematical style, short, readable and elegant. We want also these specifications to be executable. Notice that the SIS system of Peter Mosses [10,7] proves some years ago that these goals were reachable. Trying to reach our goals results in a computer formalism called Typol. The language is still under development but it has already been experimented with extensively. Typol programs may be executed from inside the Mentor meta-syntactical editor [6,5], or from inside the new Centaur[16] system. So, Mentor and Centaur with Typol are two complete meta-compilers. Natural semantics is close enough to mathematical logic to allow proofs on Typol programs. One example has been the proof of correctness of a translation from a subset of ML into CAM (Categorical Abstract Machine [3]) [4]. This experiment shows such proofs could be computer driven.

This report wants to be a complete reference manual and a guide to Typol and Natural Semantics. It refers to version 2 of Typol that is implemented in the version 0.5 of Centaur. It contains numerous examples of semantic specifications written in Typol and running under this system. In the technical parts of this report we will assume that the reader has a good knowledge of Centaur.

Typol programs are entered in the computer using a traditional ASCII representation. These programs may be unparsed with a nice pretty-printer that produces TEX inputs. This pretty-printer is run after type-checking, so it is possible to associate automatically different fonts to different types of objects, in accordance with standard mathematical practice. Abstract syntax trees are also unparsed using their concrete representation. All this encreases the readability of semantic definitions. A lot of examples in the rest of this document are produced using this strategy. In this manual, we will sometimes use the ASCII representation of Typol programs, sometimes a computed form of output, using or not appropriate unparsers for the manipulated formalisms. Different outputs may be the external representations of the same Typol program.

A Typol type-checker is now available. It is a major component of the Typol programming environment. A number of processors in this environment must be executed after type-checking. It is the case of course for the code generator, to produce Prolog code, but also for the pretty-printer that produces input for $\TeX$, as we said above. Of course, all these components are written in Typol. The Typol type-checker has (at least) two goals. The first one is to solve ambiguity when two languages that use the same name for an operator are involved (in a translation for example). Most of the time the context will be sufficient to resolve the ambiguity, but we may have to specify what operator we really mean. The second goal is to infer the type of variables. In Typol, the scope of variables is limited to the rule where they appear. But variables declarations are allowed. The scope of such declarations is the set of rules where they appear. It is not necessary to declare in this way all variables, because often their type may be inferred. In particular, it is almost never useful to declare variables that stand for abstract syntax fragments when these variables occur in the subjects of the rules.

Typol specifications are compiled into Prolog code. When executing these specifications, Prolog is used as the motor of our deductive system. We have tried hard to keep the semantics of our rules independent of Prolog features. In particular the semantic of a Typol rule is independent of the order of the sequents in the numerator of the rule (as it is in logic). For efficiency reasons we would like conditions occurring in rules to be evaluated as soon as possible, to avoid building useless proof-trees. This means that we need a Prolog system with build-in coroutines. The first version of Typol used a C-Prolog interpreter [12]. The current implementation of Typol, uses MU-Prolog [11] with success.

Now, as other formalisms, Natural Semantics makes use of special notations, with a very precise meaning associated to these notations. The tree first chapters describe these notations and provide many examples. The formal description of the syntax is given in Appendix A.

Chapter 4 explains the error messages produced by the Typol type-checker.

The three last chapters form a practical guide to use Typol under Centaur. It explains how to compile and debug Typol programs, and how to interface Typol with the rest of the system.

The appendix B gives a list of problems or bugs that may be encountered when using the current version of Centaur.

1 Rules

The basic notion of Typol is inference rules. Let us see first what are the components of a Typol rule.

1.1 Tree Patterns

It seems that in all specification formalisms (not only denotational but also in abstract algebraic specifications) the way of specifying tree navigation is heavy. In usual semantic specifications we have to give two redundant pieces of information:

- What is the structure of the abstract syntax tree of a formalism,
- How to select a branch of this tree, and how to build such trees.

This is usually done by giving constructors and destructors (selectors) acting on trees. Using tree patterns will simplify this problem, because patterns contain in themselves all the information we want.

Tree patterns may contain variables, denoting subtrees. We can refer to these subtrees, just naming the corresponding variable, and we can rebuild a tree just writing a new pattern. So there is no longer a distinction between constructive and destructive objects, and the way of selecting one component is self-contained.

1.1.1 Fix-arity Operators

Let us take as example the Asple programming language [7], that will help us all over this report. The imperative part of an Asple program may contains while-loops. As usual, while loops are made of two parts: the first one is a (boolean) expression, the second one is a list of statements. In the Metal [9] definition of Asple, this will be expressed by the following abstract syntax declaration:

```
while -> EXP STMS;
```

This means that the `while` constructor is a fix-arity operator having two sons, the first one from phylum `EXP`, the second from phylum `STMS`. The Typol notation for fix-arity operator is a prefix notation. We can talk about a `while` node, writing:

```
while(X,Y)
```

In this pattern, `X` and `Y` are two variables, denoting respectively the first and the second son of our `while` node. So we can refer to these two components, just saying `X`, or `Y` (See section 1.5 page 10 for details on variables).

By convention, we will very often use as name for a variable the name of the corresponding phylum in the abstract syntax. So, we will write:

```
while(EXP,STMS)
```

Using the appropriate pretty-printer, this will be unparsed as follow:

while EXP do STMS end

Here are some more examples for fix-arity operators:

```
if(EXP,STMS1,STMS2)
#program(DECLS,STMS)
output(EXP)
bool()
plus()
```

The "#" character in front of the operator name **program** is not meaningful. It is necessary here because "program" is a keyword of Typol. Notice that **bool** and **plus** are nullary operators. The empty pair of parenthesis is useful here to avoid ambiguity with variables. In Metal, **bool** and **plus** are seen as singletons. They are defined using the following syntax:

```
bool -> implemented as SINGLETON;
```

The pretty-printed version of these examples will be:

if EXP then STMS_1 else STMS_2 fi
begin DECLS STMS end
output EXP
bool
+

1.1.2 Atomic Operators

Atomic operators are nullary operators. They usually contain some value. Typically, identifiers are represented in the abstract syntax using atomic operators. There are three kinds of atomic operators: they may be implemented as IDENTIFIER, STRING or INTEGER. Here are some examples of atomic operator definitions, in Metal:

```
id -> implemented as IDENTIFIER;
number -> implemented as INTEGER;
```

In Typol, it is possible to talk about the value of atomic operators, using variables:

```
id X
number N
boolean V
```

In these patterns, X, N and V are variables denoting the values of atomic operators. It is possible to write patterns using constants. These constants are integers for operators implemented as INTEGER, or strings for operators implemented as IDENTIFIER or STRING.

```
id "foo"
number 24
boolean "true"
```

These patterns denote respectively an identifier with internal value "foo", the integer 24, and the boolean "true". They will be pretty-printed in the following manner:

```
foo
24
true
```

1.1.3 List Operators

List operators may have an arbitrary number of sons. All these sons, we will say elements, must belong to the same phylum. In Metal, list operators will be defined using the $*$ or $+$ notations:

```
decls -> DECL + ...  ;
 stms -> STM * ...  ;
idlist -> ID * ...  ;
```

These declarations mean that the `decls` operator denotes a list of declarations, each one belonging to the `DECL` phylum, that the `stms` operator denotes a list of statements, each one belonging to the `STM` phylum, and that the `idlist` operator denotes a list of identifiers.

In Typol, there is a special notation for lists. It is possible to write patterns denoting a list with a fix number of elements. The first example is for empty lists:

```
idlist[]
```

The following examples are for lists with a given number of elements denoted by variables:

```
idlist[A]
idlist[A,B,C,D]
```

It is also possible to isolate some elements in the beginning of a list from the rest of this list, using the dot notation:

```
idlist[A.Q]
idlist[A,B,C,D.Q]
```

In these examples, the variables `A`, `B`, `C` and `D` stand for elements of the list (say identifiers), and the variable `Q` stands for a (sub)list of identifiers. There is no way to match a sublist elsewhere that at the end of a list.

1.1.4 More about Patterns

Typol patterns are as general as possible, and you can write very complicated patterns, if you want. Here are some examples of patterns. The first one denotes a list of identifiers, the first of which is the identifier "foo":

```
idlist[id "foo".Q]
```

The following pattern represents an if-statement, with an empty else-part, and where the expression on the right hand side is a sum:

```
if(bop(EXP1,plus(),EXP2),STMS,stms[])
```

It can be unparsed in the following manner:

$\boxed{\text{if } \mathrm{EXP}_1 + \mathrm{EXP}_2 \text{ then } \mathrm{STMS} \text{ fi}}$

One should be careful when reading pretty-printed patterns. The external representation may be ambiguous while it is of course not the case for the abstract representation. So the following patterns are distinct:

```
id "foo"
idlist[id "foo"]
```

but they may have a single concrete representation:

$\boxed{\text{foo}}$

There is also a problem with empty lists that are almost always represented by nothing. All those problems are not Typol problems, but only pretty-printer's problems. It only means that sometimes some patterns should not be pretty-printed.

1.1.5 Technical Details

- Operator names are always in lowercase. They are those used in the Metal definition of the formalism. We will see later that renaming is possible (see section 3.1 page 16 for details).
- When an operator name is also a reserved keyword of Typol, it must be prefixed with the "#" character. *ex:* `#program`
- To disambiguate operators it is possible to postfix the name of an operator with the name of a formalism (in uppercase). *ex:* `if::ASPLE`
- Values for an atomic operator must be exactly the same as the internal representation used by Centaur for this atom, that is lowercase if the internal representation is in lowercase *etc...* .

1.2 Propositions

In natural semantics, computing the semantic value of an expression in a language is no more than proving that a particular proposition holds. Very often, this proposition is a relation between two or more objects. Typol offers various way of writing propositions: propositions, relations, labelled relations, and anonymous propositions.

1.2.1 Propositions

This is the more traditional style of propositions. Its general form is:

```
P(x)
```

where P is a name (let's say a predicate), and x is a list of Typol expressions, separated by commas. Here are some examples of such propositions:

```
IS_BOOLEAN(V)
IS_WELL_FORMED(if(EXP,STMS1,STMS2))
GREATER(25,0)
```

1.2.2 Relations

Relations are infix propositions. Some binary infix relational operators are predefined as predicates. There is no special meaning associated with these symbols. Here is the list of these predefined operator symbols:

```
|-> -> => < > << >> = <= <- >= <=> |= <-> (- -) ? & :
```

You can decide on your own that the meaning of

```
x : t
p -> c
v <= v'
```

is

x *has type* t
p *can be translated into* c
v *is a prefix of* v'

Both arguments of these binary relational operators may be lists of Typol expressions.

1.2.3 Labeled Relations

It is possible to attach labels to a relational operator. This means that it is possible to distinguish some parameters among all the parameters of a predicate. Labels are list of Typol expressions. Any relational operator may labeled. In the following example, the relational operator -> is labeled by two labels i and o.

```
{i -> o}
```

When the Typol pretty-printer is used, left and right labels are respectively unparsed above and under the relational operator. For example, if we want to say that the state of an abstract machine will change to an other state, producing some output o and consuming some input i, we may write:

```
s {i -> o} s'
```

and this will be unparsed as:

$$\boxed{\sigma \xrightarrow[o]{i} \sigma'}$$

1.2.4 Anonymous Propositions

Anonymous propositions are propositions that are not named. Let's say that these propositions don't use a predicate symbol. As for relations, the meaning of such propositions is not fixed. The general form of these predicates is:

$$\boxed{\mathtt{x}}$$

where **x** stands for a list of Typol expressions.

1.2.5 Technical Details

- The names of predicates are always in uppercase.
- A list of arguments may be empty.
- When the list of arguments of a predicate is empty, the empty parentheses must be present to avoid ambiguity with variables.
- Predicates may be overloaded on the number and types of their arguments.

1.3 Sequents

Sequents express the fact that some hypotheses are needed to prove a particular proposition. A sequent has two parts: the first part contains the *hypotheses*; the second one, called *consequent*, is a proposition. These two parts are traditionally separated by the turnstile symbol "⊢". An hypothesis should be a set of propositions. In the current implementation of Typol hypotheses are a list of Typol expressions. The first expression in the right part of a sequent in called the *subject* of the sequent. Here are some examples of sequents:

```
|- #program(DECLS, STMS)
   env[] |- DECLS -> e
      e |- EXP : t
    |- IS_BOOLEAN(t)
```

how they may be pretty-printed:

$$\vdash \text{begin}\ \mathrm{DECLS}\ \mathrm{STMS}\ \text{end}$$
$$\rho_{\emptyset} \vdash \mathrm{DECLS} \rightarrow \rho$$
$$\rho \vdash \mathrm{EXP} : \tau$$
$$\vdash \mathit{IS\_BOOLEAN}(\tau)$$

and their intuitive meaning:

-*The program* `#program(DECLS, STMS)` *is well typed.*
-*Given an empty environment, the list of declarations* DECLS *produces the environment* e *(ρ).*
-*The expression* EXP *as type* t *(τ) given the environment* e *(ρ).*
-t *(τ) is boolean.*

In the first example the proposition is an anonymous proposition, and the subject is `#program(DECLS, STMS)`. The second and third examples use relations (*resp.* `->` and `:`), and the subjects are `DECLS` and `EXP`. The last example use a traditional proposition, and the subject is `t`.

Typol offers another kind of sequents: the named sequents. See section 2.4 page 14 for more details.

1.3.1 Judgements

The various forms of sequents participating in the same semantic definition are called *judgements*. These different judgements are used without being given explicit names. We can say that in a semantic definition, the turnstile symbol is overloaded.

1.3.2 Abbreviated Sequents

When a list of hypotheses is empty, it is possible to omit the turnstile symbol in a sequent. In this case, the consequent must be enclosed into a pair of parentheses. So the following sequents are equivalent:

`|- P(x)`	`(P(x))`
`|- x -> y`	`(x -> y)`
`|- x {i -> o} y`	`(x {i -> o} y)`

The semantics of a sequent and its abbreviated form is the same (in particular the generated code is the same), excepted when the consequent is an anonymous proposition. So `|- x` is not equivalent to `(x)`. See section 5.2 page 25 for more details.

1.4 Inference Rules & Axioms

An inference rule explains when a sequent can be deduced from other sequents. It has two parts, named *numerator* and *denominator*. These two parts are separate by an horizontal line. The denominator consists of a sequent, and the numerator consists of a collection of sequents, also called *premises*, separated by ampersands. A rule with an empty numerator is called an *axiom*. The subject of the sequent in the lower part of the rule (denominator) is called the subject of the rule. Intuitively, an inference rule states that if all the sequents in the numerator hold (i.e. they can be proved), then the denominator holds. Here are some examples of inference rules and axioms:

```
   env[] |- DECLS -> e & e |- STMS
   --------------------------------
       |- #program(DECLS, STMS);

           e |- decls[] -> e;

 e |- DECL -> e1 & e1 |- DECLS -> e2
 -----------------------------------
     e |- decls[DECL.DECLS] -> e2;
```

They can be pretty-printed in the following manner:

$$\frac{\rho_{\emptyset} \vdash \text{DECLS} : \rho \qquad \rho \vdash \text{STMS}}{\vdash \textbf{begin}\ \text{DECLS STMS}\ \textbf{end}}$$

$$\rho \vdash \text{decls}[] : \rho$$

$$\frac{\rho \vdash \text{DECL} : \rho_1 \qquad \rho_1 \vdash \text{DECLS} : \rho_2}{\rho \vdash \text{DECL; DECLS} : \rho_2}$$

The first rule explains when an Asple program is well typed: if, given an empty environment, the declarative part produces the environment ρ, and if using this environment the statement part is well typed, then the whole program is well typed. The second rule (in fact an axiom) says that an empty list of declarations does not modify the environment. The third rule explains that in Asple, the elaboration of declarations is linear.

The horizontal line between the numerator and the denominator must contain at least three dashes, and the whole inference rule may be typed in a single line for parsing.

1.5 Variables

Variables are very similar to Prolog variables. The main difference is that Typol variables are typed. There is a strong difference between a variable and a variable name. A variable name is a reference to something that may be a variable or not. A variable name is local to an inference rule or an axiom, while a variable survives outside an inference rule. Variable names stand for abstract syntax trees or constants. Using the same variable name in different occurrences (but in the same rule) means that values denoted by these variables must be unified. In fact, the name "variable" is used both for variables and variable names.

Typol variables are typed. There types must be declared by the user unless they are used inside a pattern so they can be deduced by the type-checker. Declared variables are written using lowercase identifiers, while others are written using uppercase identifiers. Here are some possible variable names:

```
DECLS STMS STMS1 STMS2 STMS' ro e #use x x' x''
```

Typol keywords may not be used as variable names (in lowercase), but it is possible to use the "#" prefix if necessary. When parsing, only the case of the first letter is meaningful. So you can write:

```
Decls Stms r0
```

1.5.1 Reserved variables

The variable names `subject` and `this_rule` are reserved. The value of these variables are respectively the path from the root of the abstract syntax tree to the subject of the rule and the path to the current rule in the source. These variables are used in the interface with Centaur. See section 5.3.4 page 29 for more details.

1.5.2 Anonymous Variables

Sometimes, the name of a variable appears only once in an inference rule. In this case, this variable may not be explicitly named, using an anonymous variable: the variable name is replaced, as in some Prolog implementations, by an underscore:

```
_
```

Notice that each occurrence of an anonymous variable in a rule denote a different variable. Anonymous variables are not typed.

1.6 Typol Rules

To make semantic specifications cleaner, and the connection between semantic definitions and the rest of the world easier, some additional information may be hooked to the inference rules. A Typol rule is a triple made of an optional rule name, an inference rule (or an axiom) and this information:

```
<rule_name> <inference_rule> <infos>
```

The rule name may be any any Typol expression followed by a semicolon, and is optional. Here is an example of named rule with an empty infos part:

```
"empty declaration list":  e |- decls[] -> e;
```

The <infos> part is divided in three parts which are described in the following subsections:

```
<where> <provided> <actions>
```

1.6.1 Where

This part is not yet implemented. It is supposed to allow factorizing very similar rules, saying that a variable is in fact one in a collection of patterns.

1.6.2 Provided

The provided part contains a list of conditions, that are sometimes called side-conditions. These conditions are checked as soon as possible, when the rule is used. They are part of the semantics, and are used to write some constraints that cannot be expressed in a structural manner, using patterns. This is in particular the case when it is necessary to say that two objects must be different, as in the following example:

```
provided DIFF(x,y);
```

From the syntactic point of view, a provided part is a list of premises. But these premises are most of the time simple propositions. In the example, we use a Prolog predicate (See section 5.2 page 25 for details).

1.6.3 Actions

As it is done in other systems, such as Yacc for example, it is possible to attach actions to semantic rules. Of course these actions are not part of the semantics, and cannot interfere with the formal system. These actions may only read the variables of the inference rule to which they are attached. Actions never fail. So it is possible, by using actions, to isolate pure semantic definitions from their side effects. These actions are very useful for extracting some pieces of information from the formal system. They can print messages, call the redisplay procedure of Centaur to show the current expression (i.e. the subject) of the current rule, give control to the user, etc...

The actions are done after that the whole numerator of the rule has been proved. But we should be careful. If it is possible that for some reason the sytem will backtrack after doing an action, this action will not be undone. What is done is done.

From the syntactic point of view, an action is a list of premises. But these premises are most of the time simple propositions. Here is the general form of an action:

```
do <premise_s> ;
```

1.7 Semantic Specifications

A semantic specification consists in an unordered collection of inference rules or axioms, together with a declarative part (see section 3 page 16). Theses rules define a formal system in which it is possible to prove that some particular propositions hold. No hypotheses are made about the unicity of proofs in those systems.

The order of the rules in the specification is not meaningful. Changing this order will not affect the semantics of the specification. If at one time, more than one rule may be used in building the proof tree, the most specific rule will be tried first. This does not mean that other possible choices are not legal. In fact, this implies that inference rules are sorted by the compiler. This is not done in the current implementation of Typol.

2 Modularity

Modularity is an important feature of Typol. It is very useful to handle large semantic specifications, and to make them easier to read (overloading is a very nice feature, but abuse of overloading leads to obscurity).

2.1 Programs & Bodies

A Typol program is the unit of compilation of Typol specifications. It is made of three parts.

The first part is the name of the program. This name is used for generating the names of Centaur and Prolog files that are created by the system. Assuming that the name of the program is `FOO`, these files will be `FOO.po`, `FOO.ty` and `foo.pg` for, respectively, the polish form, the textual form of the program, and the Prolog code generated by the Typol compiler. The name of a program must be in uppercase.

The second part is a list of global declarations. These declarations may be use declarations (see section 3.1 page 16), or importation of sets of inference rules (see section 2.5 page 15).

The third part is a body. The body of a program is again made of three parts. The first part contains a list of (local) variable declarations (see section 3.2 page 16). The second part is a list of inference rules, or sets (see section 2.3 page 14). The third part is reserved for future extensions.

Here is the general form of a Typol program:

```
program <UCID> is
    <global_s>
    <body>
end <UCID> ;
```

The repetition of the program name after the keyword **end** is optional. The general form of a body is:

```
<local_s>
<rule_s>
<reserved>
```

2.2 Braces

It is possible to collect some rules using braces. These braces don't have any semantic meaning. There are pure syntactic sugar. Here is an example of such braces:

```
{
e |- decls[] -> e;

e |- DECL -> e1 & e1 |- DECLS -> e2
-----------------------------------
e |- decls[DECL.DECLS] -> e2;
}
```

2.3 Sets

A set is a named collection of inference rules. This collection of rules is a complete formal system. In a set, an (anonymous) sequents in the numerators of a rule refer to the same set of rules. This is also the case in a program, which is a particular sort of set. A set is made of two parts. The first part is the name of the set. As for programs, this name must be in uppercase. The second part is a body, again as for programs. The general form of a set is very similar to programs:

```
set <UCID> is
    <body>
end <UCID> ;
```

As for programs, the repetition of the set name is optional. We give now a complete example of set, which is part of the static semantics of Asple.

```
set TYPE_OF is
var x :  ASPLE::ID;
var m :  ASPLE::MODE;

env[type(x, m).L] |- x :  m;

L |- x :  m
--------------
env[E.L] |- x :  m;

end TYPE_OF;
```

set TYPE_OF is

$x:\mu \cdot \mathrm{L} \vdash x:\mu$

$$\frac{\mathrm{L} \vdash x:\mu}{\mathrm{E} \cdot \mathrm{L} \vdash x:\mu}$$

end TYPE_OF;

2.4 Named Sequents

Sequents in the numerator of an inference rule refer to the local set or program. It is possible to name explicitly an other set to be used. This allows switching from one formal system to an other one. For example, assuming that in the type-checker of Asple there is a set named TYPE_OF, that specifies how to find the type of an identifier in the environment, we can explicitly switch to this system writing:

```
TYPE_OF(e |- id X : m)
```

This will be unparsed in the following manner:

$$\rho \overset{\text{type\_of}}{\vdash} \mathrm{id}\,\mathrm{X}:\mu$$

Here is an other example. The predicate IS_BOOLEAN is defined in the set TYPE_-COERCION which contains all predicates on Asple types. In an if-statement, the type of the expression must be boolean. So the following rule appears in the Asple type-checker specification:

$$\boxed{\frac{\rho \vdash \mathrm{EXP} : \mu \qquad \overset{\text{type\_coercion}}{\vdash} \; IS\_BOOLEAN(\mu) \qquad \rho \vdash \mathrm{STMS}_1 \qquad \rho \vdash \mathrm{STMS}_2}{\rho \vdash \text{if } \mathrm{EXP} \text{ then } \mathrm{STMS}_1 \text{ else } \mathrm{STMS}_2 \text{ fi}}}$$

2.5 Importing Sets

Set definitions are local to the current program. But it is possible to import sets defined in an other Typol program, renaming them if it is necessary, using import clauses. Import clauses must appear in the second part of Typol programs. The syntax of import clauses is:

```
import <renaming_list> from <UCID> ;
```

where <UCID> is the name of the Typol program from which sets are imported, and <renaming_list> contains set identifiers or renaming declarations of the form:

```
<OLD_NAME> as <NEW_NAME>
```

Here are some examples of such import clauses:

```
import TYPE_OF from ASPLE_ENV;
import A, B, C as X, D as Y from FOO;
```

3 Types

To be compiled, a Typol program must contains some declarations. The abstract syntax of every syntactic construction must be define, and some variables must be declared. The possibility of defining new types is under development.

3.1 The USE declaration

A formalism must be declared before it is used. The use clause imports all the abstract syntax of a formalism, and possibly renames some operator names (to limit overloading). A use clause is global to a Typol program. It syntax is :

```
use <UCID> <renaming_list> ;
```

where <UCID> is the name of a formalism and <renaming_list> contains a list of renaming declarations beginning by the keyword **renaming** and separate by colons:

```
<OLD_NAME> as <NEW_NAME>
```

Here are some examples of such use clauses:

```
use ASPLE;
use TYPOL renaming lcid as ident, ucid as bigident;
```

3.2 Declaration of variables

In a Typol rule, variables in uppercase may be used only when their types may inferred from the context by the type-checker. Variables in lowercase must be used when it is impossible to infer the types of these variables, or when one wants to restrict the type of an expression. Declarations of variables are local to a set. The following syntax must be used:

```
var <LCID_S> : <TYPE> ;
```

where <LCID_S> is a list of identifiers in lowercase, and <TYPE> is a type. Allowed types are: `string`, `integer`, path, every phylum from a formalism (as ASPLE::EXP), and private types. A private type is only represented by an identifier in lowercase. We do not yet check that used types are legal (in particular, we do not check that a particular phylum does exist in the definition of a formalism). Here are some examples of variable declarations:

```
p :  path;
x, y :  TYPOL::IDENT;
name :  string;
e :  env;
n, n', n'' :  integer;
```

3.3 Creating a new operator

It is possible to create new operators. This possibility is still under development. These operators are not encoded when compiled into Prolog (but list operator are implemented as binary trees). So it is possible to use in Typol some Prolog terms. Definitions of new operators are global to a program. The following syntax must be used:

```
define <LCID_S> : <OP_TYPE> ;
```

where <LCID_S> is a list of identifiers in lowercase, and <OP_TYPE> is of the form:

```
( <SONS_TYPE> ) → <LCID>
```

where <LCID> is the name of a private type, and <SONS_TYPE> is a list of types that are the allowed types for the corresponding sons (as in METAL definition). The allowed types for sons are the same as for variables.

It is possible to create list operators, using the following notation for <SONS_TYPE>:

```
<TYPE> *
```

and it is possible to build free terms, saying that <SONS_TYPE> is either *, for a fix arity operator, or * * for a list operator. In this case, the type-checker will never check what is under the defined operator. Here are some examples of operator definitions:

```
define foo : ( TYPOL::EXP, string) → bar;
define fop : ( * ) → bar;
define foq : ( TYPOL::LCID *) → bar;
```

Creating free operator may be dangerous. The Typol type-checker cannot use a free term to infer the type of an undeclared variable. Schemes may be incorrect under a free operator, and ambiguities may occur. In these cases the generated code may be different from what is expected by the programmer.

4 Errors in a Typol program

When running the type-checker under Centaur (see section 6.1 page 33 to know how it is called), the exact point where the error occurs becomes the current expression, while the message is displayed. The type-checker stops after each error, and the following menu is displayed.

```
Continue
 Abort
 Help
```

By selecting one of the items, the user may continue or abort the type-checking process.

The following sections contain an example for each kind of error in Typol programs. Error messages produced by the Typol type-checker appear as comments in these programs. In each case we include some explanations. We discuss not only why there is an error, but also what are the consequences, if any, for the rest of the type-checking process.

4.1 Multiple declaration of a formalism

Multiple declaration of a formalism is allowed in Typol[1]. As it may be an error, the user is given a warning, as in the following example:

```
program E_1 is
use TYPOL renaming rule as typol_rule;
use TYPOL renaming lcid as ident;
-Warning: The language TYPOL is used twice.

|- typol_rule(A,B) ;

|- ident X ;

end E_1;
```

Here, two use-clauses import the same formalism (TYPOL). A warning is printed, but renaming-clauses are valid. So, in the example above, `typol_rule` and `ident` are correct node operators.

4.2 Unknown formalism

When a formalism is used, the Typol type-checker attempts to locate the abstract syntax of that formalism. If "L" is the name of a formalism, its abstract syntax will be found in the "l.t.pl" file, in the formalism directory. If searching the definition of a formalism does not succeed, an error occurs.

[1]The Typol type-checker prefers to give a warning instead of an error message when multiple non-contradictory declarations exist.

```
program E_2 is
use FOO;
-Fatal error: I don't know anything about FOO.

end E_2;
```

The language FOO is unknown to the system (i.e. no abstract syntax definition of FOO may be located). As this may generate a large number of irrelevant errors in the rest of the program, the type-checker stops. This is the only case where type-checking aborts. Section 6.2 page 33 explains how to generate the abstract syntax tables used by the type-checker.

4.3 Renaming of an unknown operator

Renaming an operator means that the "standard" name of that operator is made unavailable. The new name must be used instead. The type-checker checks that the old name is legal.

```
program E_3 is
use TYPOL renaming foo as bar;
-Error: "foo" is not a TYPOL operator.

|- foo(A) ;
-Error: I can't infer any type for this rule.

|- bar(A) ;
-Error: I can't infer any type for this rule.

end E_3;
```

The abstract syntax of TYPOL does not contain any operator named foo, so it is impossible to rename it. No declaration for bar is done either, because we do not know what type it should be given. This could be improved.

4.4 Multiple renaming of an operator

For convenience, it is possible to give more than one name to an operator. But this may be an error, so the user is given proper warning.

```
program E_4 is
use TYPOL renaming lcid as ident, lcid as identifier;
-Warning: the TYPOL operator "lcid" is already renamed.

|- ident X ;

|- identifier X ;

|- lcid X ;
-Error: I can't infer any type for this rule.

end E_4;
```

The TYPOL operator lcid is renamed twice. Both ident and identifier are new names for TYPOL::lcid. But lcid will not mean TYPOL::lcid.

4.5 Importing or defining an existing set

It is not permitted to reimport or to redefine an already defined set.

```
program E_5 is
import FOO as BAR;
import FOO;
import BAR;
-Error: the set BAR is already imported or defined.

set FOO is
-Error: the set FOO is already imported or defined.

end FOO;

end E_5;
```

When importing FOO as BAR, BAR is defined, but not FOO. So FOO can be imported or defined, but not BAR. When importing FOO again, FOO is defined, and may not be redefined.

4.6 The variables "subject" and "this_rule" are predefined

The variables "subject" and "this_rule" are two reserved variables in Typol. They give, when it is possible, an access to the subject of the rule, and to the rule itself in the source program. These variables do not need to be declared and they may not be redefined.

```
program E_6 is
var subject :  string;
-Error: The variable "subject" is predefined.

set FOO is
var this_rule :  string;
-Error: The variable "this_rule" is predefined.

end FOO;

end E_6;
```

4.7 Multiple declaration of a variable

Multiple declaration of a variable is not permitted within a set or a program.

```
program E_7 is
var a :  string;
var a :  integer;
-Error: The variable "a" is already declared.

|- lcid a ;

|- integer a;
-Error: I can't infer any type for this rule.

set FOO is
var a :  integer;

|- integer a ;

end FOO;

end E_7;
```

In a set or a program, if one attempts to redefine an already declared variable, the old definition remains unchanged. So in E_7, a is a variable of type `string`, not **`integer`**.

Remember that declarations of variables are local to the program or to the surrounding set, in the Algol style.

4.8 Missing definition of sets

A set that is used must be either imported or defined in the same program.

```
program E_8 is
use TYPOL;

FOO(A) & BAR()
-Error: The set FOO is neither imported nor defined.
--------------------
|- number A;

set BAR is
end BAR;

end E_8;
```

Here the set FOO is not defined, and this an error. On the other hand the set BAR is used before it is defined and this is allowed. Note that the message is printed only at the first occurrence of the name of the undefined set.

4.9 Not completely instanciated type

The type of a rule must be completely inferred by the type-checker. This means that every variable that is in a rule must receive a type.

```
program E_9 is
use TYPOL;

|- integer A, B;
-Error: I can't instanciate the type of B.

end E_9;
```

Variables in lowercase must be declared. But variables in uppercase may not be declared[2]. There types must be inferred from the context (the rest of the rule). If it is not possible, an error occurs. Here, the type of A (integer) may be deduced from the type of the operator integer. It is not possible to deduce the type of B.

4.10 Unresolved overloading

Only one type must be possible for a rule. Ambiguous types are not permitted.

[2]Variables in uppercase may be used when their types may be inferred from the context. Variables in lowercase must be used when it is impossible to infer the types of these variables, or when one wants to restrict the type of an argument.

```
program E_10 is
use TYPOL;
use METAL;

|- node_id(PHYLUM, lcid A) ;

|- lcid A ;
-Error: I can infer more than one type for this rule.

end E_10;
```

The operator `lcid` exists both in the `TYPOL` and the `METAL` abstract syntax. In the first rule, the type-checker can deduce that `lcid` stands for `TYPOL::lcid`, because `node_id` is solely a `TYPOL` operator. But in the second rule, it is not possible to deduce from the context whether `lcid` stands for `TYPOL::lcid` or `METAL::lcid`, and this is an error.

4.11 Missing declaration of variables

Variable in lowercase must be declared. If they are not declared, some messages are emitted, and they are treated as uppercase variables.

```
program E_11 is
use TYPOL;

|- lcid a ;
-Error: The variable "a" must be declared.

end E_11;
```

4.12 Other type errors

All other errors are type errors that occur within a single Typol rule. For the moment only error message covers all these errors:

-Error: I can't infer any type for this rule

It is the case in the following situations:

- When a variable in uppercase is used with different incompatible types within a rule.
- When unknown operators are used.
- When an operator is used with a wrong arity.
- When an operator has a son with an improper type.

Most of the time, the type-checker provides a more accurate message:

-Error: Type error possible here.

This means that they may be a problem at the current expression (or an incompatibility between the current expression and its father).

Further versions of the Typol type-checker will provide more accurate information, as it is to be expected.

5 Interfacing Typol with the environment

Of course a Typol program must be able to communicate with Prolog and Centaur. Typol has been extended to achieve this goals.

5.1 Defining Prolog predicates

A set of rules in the body of a set or program may define not only sequents, but also Prolog predicates. The syntax of these predicates is the same as for anonymous predicates. They may be used as premise (see section 1.4 page 9). In this case all occurrences of this sort of predicate in place of a premise in the same set will refer to the predicate defined in this set. As usual, these predicates may be overloaded on the number and types of arguments. Some extra parenthesis are needed around the list of arguments.

This sort of predicates must be manipulated with caution. The generated Prolog code does not contain any type information. In fact, defining a Prolog predicate in this way, is like writing Prolog code (but Typol objects are automatically encoded while compiling).

Here is an example of set defining such a predicate:

```
set IS_IN_LIST is
var x :  ASPLE::ID;

(X, idlist[X.L]);

(x,L)
----------------
(x, idlist[Y.L]);

end IS_IN_LIST;
```

As for sequents, we can refer to a Prolog predicate defined in a set from outside this set by naming this predicate. In this cases the extra parenthesis must not appear.

```
IS_IN_LIST(id "foo", idlist[id "foo", id "bar"])
```

5.2 Interface with Prolog

It is possible to use from inside a Typol program a predicate written in Prolog. A Prolog predicate may be called as described in the previous section. The predicate must be in lowercase in Prolog, but will be used in uppercase in Typol. This predicate must be declared in the Typol program, using an import clause, without the "from" part. Notice that renaming is possible.

Here are some examples of declarations of Prolog predicates:

```
import WRITE, NL, PLUS;
import A, B as X, D;
```

When using a Prolog predicate, be sure that types are compatible between Typol and Prolog. That is possible only for simple objects (integers, strings and variables) unless this Prolog predicate is part of the Typol environment (i.e. is a Typol predefined predicate). The orders of the parameters are the same in Typol and in Prolog, but don't forget the empty brackets in Typol, when the arity is zero as in: `NL()`.

The following table shows how simple Typol objects are translated into Prolog:

TYPOL		PROLOG	
integer	*ex:* 23	integer	*ex:* 23
string	*ex:* "foo"	identifier	*ex:* 'foo'
variable	*ex:* A x _	variable	*ex:* _a _x _

When more complicated objects are used, a set defining an anonymous predicate must be used. Let us see now a complete example of interface between Typol and Prolog.

The TYPOL side:

```
import ADD;
var a, b, c :  ASPLE::EXP;
...
...|-...  & SOMME(a,b,c)
------------------------
...;

set SOMME is
ADD(A,B,C)
------------------------------
(number A, number B, number C);
end SOMME;
```

The PROLOG side:

```
add(P,Q,R) :- R is P + Q.
```

It is not necessary to know how Typol trees are encoded into Prolog. The interface is done automatically by the set ADD.

5.3 Interface with Centaur

Communication with the host system is of course very important. You can avoid using the interface between Prolog and Typol, but it is impossible not to use the interface with Centaur.

This interface has three goals:

- Coercion of Centaur objects into Prolog ones, and back.
- Calling Le_lisp functions and the messages management system from within a Typol program.
- Keeping trace of a Typol program execution in the Centaur data structure.

It is made of a collection of predefined Prolog predicates, which will probably found in actions, or in provided clauses.

5.3.1 Moving objects around

GETVAR

Syntax: GETVAR(var,p)

Function: Give a pointer to Centaur tree stored in Le_lisp variable. If "var" is "k", give a pointer to the current expression in the current view.

Arguments: "var" must be instanciated to a string that is the name of the variable. "p" must be a variable that will be instanciated to an integer, which is a pointer to the tree stored in the variable "var".

GETTREE

Syntax: GETTREE(lang,p,term)

Function: Coerce a Centaur tree into a Typol object.

Arguments: "lang" must be instanciated to a string that is the name of a formalism. "p" must be instanciated to an integer that is a pointer to a tree. "term" must be a variable that will be instanciated to a Typol object.

Example: GETVAR("k",a) & GETTREE("typol",a,t) gives in t a Typol representation of the abstract syntax tree that is the current expression of the current view.

SENDTREE

Syntax: SENDTREE(lang,term,p)

Function: Coerce a Typol object into a Centaur tree.

Arguments: "lang" must be instanciated to a string that is the name of a formalism. "term" must be instanciated to a Typol term of type "lang". "p" must be a variable that will be instanciated to an integer, which is a pointer to the tree corresponding to "term".

Note: term must not contains non-instanciated variables.

SENDVAR

Syntax: SENDVAR(lang,var,p)

Function: Affect a Centaur tree to a Le_lisp variable.

Arguments: "lang" must be instanciated to a string that is the name of a formalism. "var" must be instanciated to a string that is the name of a variable. "p" must be instanciated to an integer that is a pointer to a tree.

Example: SENDTREE("typol",t,a) & SENDVAR("typol","k",a) affects the abstract syntax tree corresponding to the Typol object t to the current expression of the current view.

TREE_SEND

Syntax: TREE_SEND(lang,p,ext)

Function: Coerce a Typol object into a Centaur tree, and send it in a view.

Arguments: "lang" must be instanciated to a string that is the name of a formalism. "p" must be instanciated to an integer that is a pointer to a tree. "ext" is a string that is used as an extension. The name of the view in which the Centaur tree is send is built using the name of the current view and this extension.

EXISTF

Syntax: EXISTF(f,l,p)

Function: Find the file "f" and return the complete path "p". If "l" is the integer 0, the search is done in all "semantics" directories. If "l" is a formalism name, the search is done only the root directory of this formalism.

Arguments: "f" must be instanciated to a string. "p" must be a variable that will be instanciated to a string. "l" may be either the integer 0, or a string that is the name of a formalism in the system.

5.3.2 Calling the message facilities

PRMESS

Syntax: PRMESS(f,l,n,args)

Function: Print a message.

Arguments: "f" must be instanciated to a string that is the name of the file of messages to be used (without extension .mess). If "l" is a string that is the name of a formalism, the file is search in the root directory of this formalism. If "l" is "resources", the file is searched in the resources directory. "n" must be instanciated to an integer, which is the number of the message to be printed. "argi" are optional parameters to the message. Only integers and strings may be used as parameters. At most three parameters may be given.

5.3.3 Calling Le_lisp

GETSYM

Syntax: GETSYM(arg1,arg2)

Function: Get the symbolic address of a Le_lisp function.

Arguments: "arg1" must be instanciated to a string that is the name of a Le_lisp function. "arg2" must be a variable that will be instanciated to an integer, which is a Le_lisp pointer.

PUSHARG

Syntax: PUSHARG(type,val)

Function: Push one argument on top of the Le_lisp execution stack.

Arguments: "type" must be instanciated to a string that is the name of a type allowed by the C to Le_lisp interface (i.e. fix, float, string, vector, pointer). "val" must be instanciated to a value of type "type".

Note: Arguments must be pushed one at a time.

LISPCALL

Syntax: LISPCALL(type,nargs,ll_name,res)

Function: Call a Le_lisp function.

Arguments: "type" must be instanciated to a string that is the name of the type od the resulting value. "val" must be a instanciated to an integer that is the number of arguments of the Le_lisp function. "ll_name" must be instanciated to the symbolic address of the Le_lisp function. "res" must be a variable that is unified with the result of the call.

5.3.4 Pointing back to the Centaur data structure

While the Prolog structure manipulated by Typol is, in some sense, a copy of the Centaur structure, it is not possible to know directly to which node in the original structure correspond a node in the Prolog one. But if we have a path from the root to a certain point in the Prolog structure, we can retrieve, following this path, the corresponding point in the original structure.

The Typol compiler generates Prolog code in such a way that a path from the root of the structure to the subject of the "current" inference rule is maintained when it is possible. This path is accessible through the Typol variable "subject", which must be not used for other purposes. The following collection of predefined predicates permits to manipulate the information contained in the variable "subject".

REDISPLAY

Syntax: REDISPLAY(path,lang,proc)

Function: Move the current expression that becomes the subject of the current rule, and execute a named procedure.

Arguments: "path" must be instanciated with a path (very often, it will be the Typol variable subject), "lang" must be instanciated to a string that is the name of the formalism of the subject, proc must be instanciated to a string that is the name of a Le_lisp function.

TRACK

Syntax: TRACK(path,lang), TRACKV(path,lang,var), TRACK(path,lang,p), TRACK()

Function: Move the current expression that becomes the subject of the current rule, or update the given variable, or give an integer that is a pointer to the corresponding structure, without updating the current expression. Without parameters, update the current expression using the last path saved by KEEP (see below).

Arguments: "path" must be instanciated with a path, and "lang" must be instanciated to a string that is the name of a formalism.

KEEP

Syntax: KEEP(path,lang)

Function: Save path and lang in a global variable.

LAST_K

Syntax: LAST_K(path,lang)

Function: Give the last pair path-lang saved.

Some comments are necessary to use these predefined predicates:

- If you want to call the redisplay procedure after each semantic rule, use REDISPLAY.
- Sometimes you would like to memorize a particular point, and then redisplay it later on. In this case, use KEEP to memorize the path, then TRACK & LISPCALL to update the current expression and call the redisplay procedure.
- The redisplay procedure costs in time. The same remark may be applied to TRACK. So, if you don't need a redisplay every time, don't use TRACK to often, but use KEEP instead.

- It is possible to redefine the procedure that is called, to suppress the redisplay. This will not, of course, suppress the updating of the current expression, which continue slowing down the Prolog execution.

The functionalities described in this section may change or disappear in the future. See section 7.4 page 37.

5.4 Executing a Typol program from within Centaur

To execute a Typol program one have to construct a Prolog goal, and then ask Prolog to solve this goal. To do that, the correspondance between a Prolog predicate and it correspondant into the Typol program must be known. This correspondance may be complicated in general, but is simple in the case of a Typol rule with an anonymous predicate as conclusion. The following table shows this correspondance:

TYPOL	PROLOG
`program FOO is`	
`...`	
`...|-... & ...|-...`	
`------------------------`	
`(a,b);`	`foo(_a,_b) :- ...`
`set BAR is`	
`...`	
`......`	
`------------------------`	
`(a,b);`	`foo$bar(_a,_b) :- ...`
`end FOO;`	

Goals are sent to Prolog using the {prolog}:send Lisp function:

```
({prolog}:send "foo(12,'bozo').")
```

It is easy to pass simple objects as parameters, as shown in the above example. To pass more complicated objects, or to get back a result into Le_Lisp world, one must use a special interface rule, as shown in the following example:

```
env[] |- p -> q
----------------
();
        provided GETVAR(k,subject), GETTREE("foo",subject,p);
        do TREE_SEND("bar",q,".res");
```

In that example, the current expression of the current view is used as argument p. The name of its formalism is foo. The result q of the computation (may be a translation), is a tree in language bar, and is sent in an other window.

A more sophisticated protocol will be developed in the future.

6 The Typol environment

6.1 The Centaur-Typol environment

As it is usual in Centaur, the file Typol.env define some specific menus to be used by the Typol user. The root menu for Typol is the following:

File	⇒
Display	⇒
Edition	⇒
Navigation	⇒
Typol Navi.	⇒
Typol	⇒

The first four submenus are the standard ones. The menu called "Typol Navi." provides some quick procedures to navigate in a Typol program.

File	⇒
Display	⇒
Edition	⇒
Navigation	⇒
Typol Navi.	FindRule
Typol	Search
	UpRule
	FindSet
	UpSet

FindRule: Prompt the user for the subject of a rule, and find a rule with that subject.

Search: Prompt the user for a Typol expression, and find a rule using it. This Typol expression may be a rule name.

UpRule: Go up to the enclosing Typol rule.

FindSet: Prompt the user for a Typol set name, and find this set.

UpSet: Go up to the enclosing set of rules.

The last menu allows to call the Typol type-checker and the compiler:

File	⇒
Display	⇒
Edition	⇒
Navigation	⇒
Typol Navi.	⇒
Typol	Type-Check
	Compile
	Compile (Db)

Type-Check: Type-Check a Typol program.

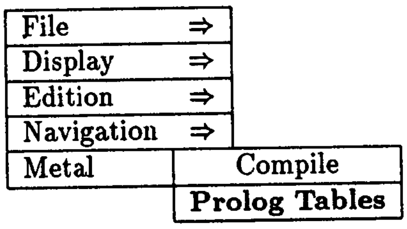
Compile: Compile a Typol program into Prolog, in the "normal" mode.

Compile (Db): Compile a Typol program into prolog in debug mode (see section 7.2 page 35).

6.2 The Centaur-Metal environment

Every Typol program must be type-checked, before it is compiled into Prolog. The smallest Typol declaration (i.e. the **use** declaration) produces the biggest effect: it imports all syntactic type information concerning a particular formalism.

When the Typol type-checker evaluates a **use** declaration, it must find somewhere the specification of the formalism that is used. This information is found in a file named `x.t.pl`, where `x` is the name of the formalism. This file is built by executing a Typol program, which is a translator from Metal to Prolog, on the Metal specification of the formalism. The following picture shows how to generate this file. The selected item is displayed in boldface.



6.3 The MU-Prolog environment

Two facilities offered by MU-Prolog are used in the Typol environment. First, an error handler has been written to load automatically undefined predicates, when possible. Second, the logic preprocessor (lpp) may be used when coroutines are needed.

6.3.1 Naming conventions and autoloading

In MU-Prolog, an attempt to use an undefined predicate is considered as an error. An error handler has been written to catch that error; when an undefined predicate is call, it tries to find a definition of that predicate in a canonical way: if the predicate `foo` or `foo$bar...` is called it tries to load the file `foo.pl`, and if this file does not exists it tries with `foo.pg`.

The Typol compiler follows of course this naming convention. If you define a set `FOO` in the typol program `BAR`, the corresponding Prolog predicate is named `foo$bar....` If you need to write your own Prolog predicates directly in Prolog, it should be tedious to create a file for each predicate. You can use the same naming convention as the Typol compiler. In this case it is necessary to say from which file a particular Prolog predicate is imported, using the `from` clause. If the file `foo.pg` contains the prolog predicate `foo$open(Name)`, use the following declaration:

```
import OPEN from FOO;
```

If you want to load or to reload some files directly in Prolog, use `load(Name)` and `reload(Name)` instead of the standard predicates `consult` and `reconsult` (tt [...] [-...]).

6.3.2 The Logic Preprocessor

The MU-Prolog logic preprocessor (lpp) may be used to generate automatically wait declarations[11]. We suggest that Prolog files generated by this pre-processor should be named `*.pl`. So these files will be chosen for auto-loading.

If you use lpp, the semantics of your Typol rules will no longer depend on the order of the premises, in the numerator of the rules.

It is possible that lpp takes some wrong decisions with non-defined predicates that are not tests (this will only slow down the speed of execution). It may be preferable not to run lpp on separated files. It is better to concatenate first all the files you need, and to run lpp on the result, as in the following example:

```
cat typol_tc.pg typol_env.pg typol_test.pg typol_aux.pg > foo
lpp <foo >typol_tc.pl
```

It is possible that in the future, the functionality of lpp will be included in the Typol compiler.

6.3.3 The Prolog menu

The standard menu bar may be modified to allow easier access to Prolog. This may be done by adding the two following lines in the startup file of Centaur (file ".centaur" in your home directory):

```
({menubar}:add-menu #:interface:the-menu-bar #:interface:prolog-menu 1)
(send 'redisplay #:interface:the-menu-bar ())
```

The Centaur menu bar is modified as follows:

System	Prolog	Views
	Call	
	Restore	

Call: Enter a Prolog session. The input is taken in the shell window. To go back to Centaur, type ctrl-D.

Restore: Restore the initial Prolog state as it was at the beginning of the Centaur session. This is useful to clean up all the Prolog data base, when too much Prolog clauses have been loaded or when some Typol programs have been recompiled. Information to check the consistency of the system is preserved.

7 The Typol debugger

7.1 Why a Typol debugger?

For a long time, debugging Typol programs was performed using the trace facilities of Prolog. Three main difficulties, mostly due to the fact that the user was obliged to debug something different from what he had written, were encountered with this strategy:

- The connection between the Typol rules and the generated Prolog clauses is not obvious. The names of the clauses are synthesized using the name of the Typol program, the name of the enclosing set, and some type information. The number of parameters in a Prolog predicate is greater than in the corresponding Typol sequent because access chains to different objects are generated by the compiler. The order of the parameters in a sequent may be not the same in the generated Prolog predicate. In the future, these problems can only get worse as there will no longer be a one-one mapping between Typol rules and generated Prolog clauses.
- Centaur trees are coerced into Prolog terms, and these are very difficult to read: lists are represented as binary terms; the abstract syntax operators are encoded; the terms are very large.
- In MU-Prolog, it is very difficult to control the execution process. Tracing is much too verbose. Clauses that do not match the current goal are tried, if no index table exists. When a goal succeeds, the bindings after the success are not shown.

This is why a real Typol debugger was necessary. By real we mean that the debugger must speak the same language as the user (that is Typol rules, and concrete syntax of formalisms), and that some control on the execution process is possible (skipping some details, going backwards, selecting one rule, etc...). The debugger presented here is a first step toward a ideal Typol debugger.

7.2 Compiling Typol programs in debug mode

As it is usually done, to be debugged, a Typol program must be compiled in a special mode, called the debug mode. The following picture shows how to select this mode in Centaur. The selected item is displayed in boldface.

File ⇒	
Display ⇒	
Edition ⇒	
Navigation ⇒	
Typol	Type-Check
	Compile
	Compile (Db)

In this mode two predicates, called `$pre_action` and `$post_action` are generated respectively before and after the code produced by the compiler in the normal mode.

These predicates get as parameters the path to the corresponding Typol rule in the source code and the path to the subject of this rule. In the future, an environment containing every variables of the Typol rule will also be passed to these predicates. A last parameter consists in a new variable (i.e. a variable that does not already appear in the generated code for the rule), which is the same for the two predicates. This variable is a link that binds the `$pre_action` and the `$post_action` together for each rule, and is used by the "Skip" command of the debugger (see below).

The execution of a Typol program compiled in the debug mode is a bit slower than the execution of the same program compiled in normal mode. It also takes some extra memory space. Of course, it is possible to use some modules that are compiled in debug mode and some other that are not compiled in this mode. The behaviour is as expected.

7.3 The role of $pre and $post actions

The `$pre` and `$post` actions have two purposes. The first one is to send a signal to Lisp, telling that something happens (the value of this signal indicates what happens). The second is to make the environment (i.e. the path to the current rule, the path to the subject, and all other variables in the future) accessible to Lisp.

The signal emitted by the `$pre` and `$post` actions may have four different values that correspond to the traditional ports of a Prolog debugger:

TRY: This means that a rule is chosen (because the conclusion matches the current sequent to be proved). This signal is emitted by the `$pre` actions only, when entering a rule.

PROVED: This mean that the conclusion of a rule is proved (i.e. all the premises have been proved). It is emitted by the `$post` actions only, when exiting normally a rule.

BACK: This mean that we are backtracking, and an other proof for a sequent must be founded. This signal is emitted by the `$post` action when re-entering a rule during backtracking.

FAIL: This mean that a rule is finally not applicable (i.e. no satisfactory proof may be built). It is emitted by the `$pre` actions when exiting a rule while backtracking.

When the signal is emitted, the `$pre` and `$post` actions wait until an answer coming from Lisp tells how to proceed. Five different answers are currently implemented:

Continue: Let the execution proceed normally. That is the action will succeed if the value of the signal was **TRY** or **PROVED**, and it will fail if the value was **BACK** or **FAIL**. When the answers are always **Continue**, `$pre` and `$post` actions do not affect the execution at all.

Abort: Abort the current execution.

Fail: Force backtracking. The action will fail. This is possible only when the value of the signal was **TRY** or **PROVED**. This may be used to choose an other rule if any (when used at a **TRY** point), to find an other proof for a particular rule (when used at a **PROVED** point) , or to go backwards in the execution (To go backwards in the execution (i.e. to undo), the answer must be always **Fail** when it is possible, otherwise **Continue**). Note that it is possible, using **Fail** to force a failure, but it is not possible to force a success as in the C-Prolog trace mode.

Retry: Force to go forwards again. This is possible only when the value of the signal was **FAIL** or **BACK**. It is be used to go forwards again after undoing, or to replay something that has been already done.

Skip: May be used only when the signal was **TRY**. Go ahead directly to the corresponding **PROVED** or **FAIL**. No signal is emitted until the right point is reached. If after a **Skip** that drives you to a **FAIL** you want to see the details, use **Retry**, then **Continue** normally. As we explained before, the **`$pre`** and **`$post`** actions get as one of their parameters a special variable that is the same for the two actions. Normally this variable is free. When the answer is **Skip**, this variable is bound (in the **`$pre_action`**), and therefor also in the **`$post_action`**. Until the corresponding **`$post_action`** is reached, no signals are emitted. We know that the right point is reached when the skip variable of an action is bound.

The answers must be elaborated in the Lisp world, as it is explained in the following section.

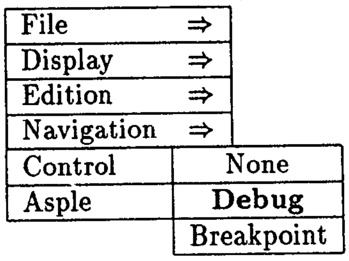
7.4 The semantics managers

A special program, called the semantics manager, must decide what answer must be returned when a signal is emitted by a **`$pre`** or **`post`** action. This (Lisp) program may access everything in the environment saved by the action, before taking its decision. So the final decision may depend on many information: the rule involved, the subject of the rule (and in the future everything in the environment of the rule). It may look a some annotations hooked somewhere in the semantics or on the subject. It may prompt the user, etc...

Three different managers have already been written in Lisp. As these managers are in fact automata, it may later be possible to describe their behaviour in Esterel[1,2], and then produce the Lisp code by compiling these specifications.

In the current implementation, the name of the manager is kept as an annotation of the Centaur views, so different views of a Centaur session may use different managers. And it is possible to switch from a manager to an other one at any time.

The following figure shows how to choose the Typol debugger as manager in the Asple environment. Again, the selected item is displayed in boldface.



7.4.1 The simplest manager: None

The manager called "None" is the default manager of a view. It is very difficult to imagine a simpler manager: it does nothing (it doesn't look at any information), but always answers **Continue**. The behaviour of a Typol program compiled in debug mode and run using this manager is the same as that of the same program compiled without the debug option. It will only be a bit slower and use more space in the Prolog stack.

7.4.2 The Typol debugger

This manager may be used to debug Typol programs. It systematically shows the current Typol rule, and the subject (when it is possible), and prompts the user with a menu, depending on the value of the signal received. The entries of these menus are exactly the answers described in section 7.3 as we can see in the following figure.

TRY	**PROVED**	**BACK**	**FAIL**
Continue	Continue	Continue	Continue
Abort	Abort	Abort	Abort
Skip	Fail	Retry	Retry
Fail	Help	Help	Help
Help			

In the future, it will be possible to get the value of any Typol expression in the current rule.

7.4.3 The Asple debugger

This is the most sophisticated manager written so far, and it gives a good idea of what is possible. Two menus are used by this manager to interact with the user.

Redisplay	X
Semantics	
Breakpt	
Done	

(Un)Set		Done	
≪	<	>	≫

The first menu allows to customize the debugger: the user may ask to see the subject (Redisplay) and/or the current semantic rule. He may ask to take breakpoints into account or not.

The second menu permits to drive the computation, and to set (or unset) breakpoints somewhere in the Asple program, or on a particular Typol rule. Breakpoints are implemented as annotations of Centaur trees. If the toggle "Breakpt" is on in the first menu, the computation stops whenever a breakpoint is encountered, otherwise breakpoints are ignored. The buttons marked ">" and "<" are for stepping forwards, or backwards. Those marked "≫" and "≪" are for running forwards or backwards until a breakpoint is encountered (if the "breakpt" toggle is on, of course).

When the user clicks on one of these four buttons, the adequate sequence of answers to Prolog signals are generated. For example, for going backwards, the generated answers will be **Fail** when it is possible, **Continue** otherwise. The environment is accessed only when it is necessary. In particular, if the toggles "Breakpt", "Redisplay" and "Semantics" are off, no access to the environment is done.

7.5 How to call the Typol debugger

The following steps must be followed:

- Compile the Typol program in debug mode (see section 7.2 page 35).
- Select the view where the Typol program must be executed.
- Set the manager of the Centaur view to be the Typol debugger. The following lisp command must be executed:

```
(let ((control-block (send 'control-block (current-view))))
     ({control-block}:manager control-block 'debugger)
)
```

- Execute the Typol program as usual.

  The following piece of code disables the Typol debugger:

```
(let ((control-block (send 'control-block (current-view))))
     ({control-block}:manager control-block ())
)
```

This may be done in a menu driven mode. The file `ASPLE.env` contains an example of a control menu that allows to select a manager.

A The complete syntax of Typol

This appendix gives the complete syntax of Typol. In the implemented version, the abstract syntax of Typol differs from the following. Some more operators exist, that must not be used. There are either obsolete (but allow compatibility with polish files coming from Mentor), or reserved for future development. Two phyla are extended: PREMISE, with the operator rule, and PROPOSITION with type_s). These features are here for technical reason and must not be used.

A.1 TYPOL.metal

```
definition of TYPOL is
   chapter AXIOM
      rules
         <AXIOM> ::= <TYPOL_PROGRAM> ;
            <TYPOL_PROGRAM>
   end chapter ;

   chapter PROGRAM
      rules
         <TYPOL_PROGRAM> ::=
            #program <UCID> #is <GLOBAL_S> <BODY> #end <UCID_OPTION> #;
;
            program (<UCID>, <GLOBAL_S>, <BODY>)

      abstract syntax
         program  -> UCID GLOBAL_S BODY ;
         PROGRAM  ::= program ;
   end chapter ;

   chapter GLOBALS
      rules
         <GLOBAL_S> ::= ;
            global_s-list (())
         <GLOBAL_S> ::= <GLOBAL_S> <GLOBAL> ;
            global_s-post (<GLOBAL_S>, <GLOBAL>)
         <GLOBAL> ::= <USE> ;
            <USE>
         <GLOBAL> ::= <IMPORT> ;
            <IMPORT>
         <GLOBAL> ::= <DEFINE> ;
            <DEFINE>

      abstract syntax
         global_s  -> GLOBAL * ... ;
      GLOBAL_S  ::= global_s ;
      GLOBAL  ::= use import define ;
end chapter ;

chapter USE
   rules
      <USE> ::= #use <UCID> <RENAMING_PART> #; ;
         use (<UCID>, <RENAMING_PART>)
      <RENAMING_PART> ::= ;
         renaming_s-list (())
      <RENAMING_PART> ::= #renaming <RENAMING_S> ;
         <RENAMING_S>
      <RENAMING_S> ::= <RENAMING> ;
         renaming_s-list ((<RENAMING>))
      <RENAMING_S> ::= <RENAMING_S> #, <RENAMING> ;
         renaming_s-post (<RENAMING_S>, <RENAMING>)
      <RENAMING> ::= <LCID> #as <LCID> ;
         operator_renaming (<LCID>.1, <LCID>.2)
      <RENAMING> ::= <UCID> #as <UCID> ;
         phylum_renaming (<UCID>.1, <UCID>.2)

   abstract syntax
      use  -> UCID RENAMING_S ;
      renaming_s  -> RENAMING * ... ;
      operator_renaming  -> LCID LCID ;
      phylum_renaming  -> UCID UCID ;
      RENAMING_S  ::= renaming_s ;
      RENAMING  ::= operator_renaming phylum_renaming ;
end chapter ;

chapter IMPORT
   rules
      <IMPORT> ::= #import <IMPORT_ITEM_S> #from <UCID> #; ;
         import (<IMPORT_ITEM_S>, <UCID>)
      <IMPORT> ::= #import <IMPORT_ITEM_S> #; ;
         import (<IMPORT_ITEM_S>, void ())
      <IMPORT_ITEM_S> ::= <IMPORT_ITEM> ;
         import_item_s-list ((<IMPORT_ITEM>))
      <IMPORT_ITEM_S> ::= <IMPORT_ITEM_S> #, <IMPORT_ITEM> ;
         import_item_s-post (<IMPORT_ITEM_S>, <IMPORT_ITEM>)
      <IMPORT_ITEM> ::= <UCID> ;
         <UCID>
      <IMPORT_ITEM> ::= <UCID> #as <UCID> ;
         set_renaming (<UCID>.1, <UCID>.2)
   abstract syntax
      import  -> IMPORT_ITEM_S FROM ;
      import_item_s  -> IMPORT_ITEM + ... ;
      set_renaming  -> UCID UCID ;
      IMPORT_ITEM_S  ::= import_item_s ;
      IMPORT_ITEM  ::= set_renaming ucid ;
      FROM  ::= ucid void ;
end chapter ;

chapter DEFINE
   rules
      <DEFINE> ::= #define <LCID_S> #: <OP_TYPE> #; ;
         define (<LCID_S>, <OP_TYPE>)
      <OP_TYPE> ::= #( <SONS_TYPE> #) #-> <LCID> ;
         op_type (<SONS_TYPE>, <LCID>)
      <SONS_TYPE> ::= #* ;
         star ()
      <SONS_TYPE> ::= <LIST_OF> ;
         <LIST_OF>
      <SONS_TYPE> ::= <TYPE_S> ;
         <TYPE_S>
      <LIST_OF> ::= #* #* ;
         list_of (star ())
      <LIST_OF> ::= <TYPE> #* ;
         list_of (<TYPE>)
      <TYPE_S> ::= ;
         type_s-list (())
      <TYPE_S> ::= <TYPE> ;
         type_s-list ((<TYPE>))
      <TYPE_S> ::= <TYPE_S> #, <TYPE> ;
         type_s-post (<TYPE_S>, <TYPE>)

   abstract syntax
      define  -> LCID_S OP_TYPE ;
      op_type  -> SONS_TYPE LCID ;
      type_s  -> TYPE * ... ;
      star  -> implemented as SINGLETON ;
      list_of  -> LIST_TYPE ;
      OP_TYPE  ::= op_type ;
      SONS_TYPE  ::= star type_s list_of ;
      LIST_TYPE  ::= star TYPE ;
end chapter ;

chapter BODY
   rules
```

```
      <BODY> ::= <LOCAL_S> <RULE_S> <AUXILIARY_FUNCTION_S> ;
         body (<LOCAL_S>, <RULE_S>, <AUXILIARY_FUNCTION_S>)

   abstract syntax
      body  -> LOCAL_S RULE_S AUX_FUNCTION_S ;
      BODY  ::= body ;
end chapter ;

chapter LOCALS
   rules
      <LOCAL_S> ::= ;
         local_s-list (())
      <LOCAL_S> ::= <LOCAL_S> <LOCAL> ;
         local_s-post (<LOCAL_S>, <LOCAL>)
      <LOCAL> ::= <VAR_DECL> ;
         <VAR_DECL>
      <VAR_DECL> ::= #var <LCID_S> #: <TYPE> #; ;
         var_decl (<LCID_S>, <TYPE>)
      <TYPE> ::= <PHYLUM_ID> ;
         <PHYLUM_ID>
      <TYPE> ::= <IDENT> ;
         <IDENT>

   abstract syntax
      local_s  -> LOCAL * ... ;
      var_decl  -> LCID_S TYPE ;
      LOCAL_S  ::= local_s ;
      LOCAL  ::= var_decl ;
      TYPE  ::= phylum_id IDENT ;
end chapter ;

chapter RRULES
   rules
      <RULE_S> ::= ;
         rule_s-list (())
      <RULE_S> ::= <RULE_S> <RULE> ;
         rule_s-post (<RULE_S>, <RULE>)
      <RULE> ::= #{ <RULE_S> #} ;
         brace (<RULE_S>)
      <RULE> ::= <EXPRESSION> #: <INFERENCE_RULE> #; <INFOS> ;
         rule (<EXPRESSION>, <INFERENCE_RULE>, <INFOS>)
      <RULE> ::= <INFERENCE_RULE> #; <INFOS> ;
         rule (void (), <INFERENCE_RULE>, <INFOS>)
      <RULE> ::= #set <UCID> #is <BODY> #end <UCID_OPTION> #; ;
         set (<UCID>, <BODY>)
      <INFERENCE_RULE> ::= <PREMISE_S> <LINE> <CONCLUSION> ;
         inf_rule (<PREMISE_S>, <CONCLUSION>)
      <INFERENCE_RULE> ::= <CONCLUSION> ;
         inf_rule (premise_s-list (()), <CONCLUSION>)
      <LINE> ::= %LINE ;
         lcid-atom (%LINE)

   abstract syntax
      rule_s  -> RULE * ... ;
      brace  -> RULE_S ;
      set  -> UCID BODY ;
      rule  -> RULE_NAME INF_RULE INFOS ;
      inf_rule  -> PREMISE_S CONCLUSION ;
      RULE_S  ::= rule_s ;
      RULE  ::= rule set brace ;
      INF_RULE  ::= inf_rule ;
      RULE_NAME  ::= EXP void ;
end chapter ;

chapter INFOS
   rules
      <INFOS> ::= <WHERE> <PROVIDED> <DO> ;
         infos (<WHERE>, <PROVIDED>, <DO>)
      <PROVIDED> ::= ;
         provided (premise_s-list (()))
      <PROVIDED> ::= #provided <PREMISE_S> #; ;
         provided (<PREMISE_S>)
      <DO> ::= ;
         do (premise_s-list (()))
      <DO> ::= #do <PREMISE_S> #; ;
         do (<PREMISE_S>)

   abstract syntax
      infos  -> WHERE PROVIDED DO ;
      provided  -> PREMISE_S ;
      do  -> PREMISE_S ;
      INFOS  ::= infos ;
      PROVIDED  ::= provided ;
      DO  ::= do ;
end chapter ;

chapter PREMISES
   rules
      <PREMISE_S> ::= ;
         premise_s-list (())
      <PREMISE_S> ::= <PREMISE> ;
         premise_s-list ((<PREMISE>))
      <PREMISE_S> ::= <PREMISE_S> #& <PREMISE> ;
         premise_s-post (<PREMISE_S>, <PREMISE>)
      <PREMISE> ::= <SEQUENT> ;
         <SEQUENT>
      <PREMISE> ::= <SET_NAME> #( <SEQUENT_OR_CONSEQUENT> #) ;
         named (<SET_NAME>, <SEQUENT_OR_CONSEQUENT>)
      <PREMISE> ::= #( <CONSEQUENT> #) ;
         <CONSEQUENT>
      <CONCLUSION> ::= <SEQUENT> ;
         <SEQUENT>
      <CONCLUSION> ::= #( <CONSEQUENT> #) ;
         <CONSEQUENT>
      <SEQUENT_OR_CONSEQUENT> ::= <SEQUENT> ;
         <SEQUENT>
      <SEQUENT_OR_CONSEQUENT> ::= <CONSEQUENT> ;
         <CONSEQUENT>

   abstract syntax
      premise_s  -> PREMISE * ... ;
      named  -> SET_NAME SEQUENT_OR_CONSEQUENT ;
      PREMISE_S  ::= premise_s ;
      CONCLUSION  ::= SEQUENT_OR_CONSEQUENT ;
      PREMISE  ::= PROPOSITION sequent named ;
      SEQUENT_OR_CONSEQUENT  ::= sequent PROPOSITION ;
end chapter ;

chapter SEQUENTS
   rules
      <SEQUENT> ::= <HYPOTHESES> #|- <CONSEQUENT> ;
         sequent (<HYPOTHESES>, <CONSEQUENT>)
      <HYPOTHESES> ::= <EXPRESSION_S> ;
         <EXPRESSION_S>
      <CONSEQUENT> ::= <PROPOSITION> ;
         <PROPOSITION>
      <CONSEQUENT> ::= <RELATION> ;
         <RELATION>
      <CONSEQUENT> ::= <EXPRESSION_S> ;
         <EXPRESSION_S>
      <PROPOSITION> ::= <PREDICATE> #( <EXPRESSION_S> #) ;
         proposition (<PREDICATE>, <EXPRESSION_S>)
      <RELATION> ::= <EXPRESSION_S> <REL_OP> <EXPRESSION_S> ;
         relation (<EXPRESSION_S>.1, <REL_OP>, <EXPRESSION_S>.2)
      <PREDICATE> ::= <UCID> ;
```

```
            <UCID>

      abstract syntax
         sequent  -> HYPOTHESES CONSEQUENT ;
         proposition  -> PREDICATE EXP_S ;
         relation  -> EXP_S REL_OP EXP_S ;
         HYPOTHESES  ::= EXP_S ;
         SEQUENT  ::= sequent ;
         CONSEQUENT  ::= PROPOSITION ;
         PROPOSITION  ::= proposition exp_s relation ;
         PREDICATE  ::= ucid ;
   end chapter ;

   chapter REL_OP
      rules
         <REL_OP> ::= <SIMPLE_REL_OP> ;
            <SIMPLE_REL_OP>
         <REL_OP> ::= <LABELLED_REL_OP> ;
            <LABELLED_REL_OP>
         <LABELLED_REL_OP> ::=
            #{ <EXPRESSION_S> <SIMPLE_REL_OP> <EXPRESSION_S> #} ;
            lab_rel_op (<EXPRESSION_S>.1, <SIMPLE_REL_OP>, <EXPRESSION_S>.2)

      abstract syntax
         lab_rel_op  -> EXP_S SIMPLE_REL_OP EXP_S ;
         REL_OP  ::= lab_rel_op rel_op ;
         SIMPLE_REL_OP  ::= rel_op ;
   end chapter ;

   chapter EXPRESSIONS
      rules
         <EXPRESSION_S> ::= ;
            exp_s-list (())
         <EXPRESSION_S> ::= <EXPRESSION> ;
            exp_s-list ((<EXPRESSION>))
         <EXPRESSION_S> ::= <EXPRESSION_S> #, <EXPRESSION> ;
            exp_s-post (<EXPRESSION_S>, <EXPRESSION>)
         <EXPRESSION> ::= <VALUE> ;
            <VALUE>
         <EXPRESSION> ::= <SCHEME> ;
            <SCHEME>
         <VARIABLE> ::= #_ ;
            anonymous ()
         <VALUE> ::= <NUMBER> ;
            <NUMBER>
      <VALUE> ::= <STRING> ;
         <STRING>
      <VALUE> ::= <VARIABLE> ;
         <VARIABLE>
      <VARIABLE> ::= <TYPE> ;
         <TYPE>

   abstract syntax
      exp_s  -> EXP * ... ;
      anonymous  -> implemented as SINGLETON ;
      STRING  ::= string ;
      EXP_S  ::= exp_s ;
      EXP  ::= VALUE SCHEME ;
      VARIABLE  ::= TYPE anonymous ;
      VALUE  ::= VARIABLE integer string ;
end chapter ;

chapter SCHEMES
   rules
      <SCHEME> ::= <NODE_ID> #( <EXPRESSION_S> #) ;
         node (<NODE_ID>, <EXPRESSION_S>)
      <SCHEME> ::= <NODE_ID> <VALUE> ;
         #atom (<NODE_ID>, <VALUE>)
      <SCHEME> ::= <NODE_ID> #[ <LIST_DESC> #] ;
         #list (<NODE_ID>, <LIST_DESC>)
      <LIST_DESC> ::= <EXPRESSION_S> ;
         <EXPRESSION_S>
      <LIST_DESC> ::= <EXPRESSION_S> #. <VARIABLE> ;
         #pre (<EXPRESSION_S>, <VARIABLE>)

   abstract syntax
      node  -> NODE_ID EXP_S ;
      #atom  -> NODE_ID VALUE ;
      #list  -> NODE_ID LIST_DESC ;
      #pre  -> EXP_S VARIABLE ;
      SCHEME  ::= node #atom #list ;
      LIST_DESC  ::= exp_s #pre ;
end chapter ;

chapter WHERE
   rules
      <WHERE> ::= ;
         where-list (())

   abstract syntax
```

```
      where  -> LCID * ... ;
      WHERE  ::= where ;
end chapter ;

chapter AUXILIARY_FUNCTIONS
   rules
      <AUXILIARY_FUNCTION_S> ::= ;
         function_s-list (())

   abstract syntax
      function_s  -> LCID * ... ;
      AUX_FUNCTION_S  ::= function_s ;
end chapter ;

chapter IDENTIFIERS
   rules
      <NUMBER> ::= %NUMBER ;
         integer-atom (%NUMBER)
      <STRING> ::= %STRING ;
         string-atom (%STRING)
      <IDENT> ::= <LCID> ;
         <LCID>
      <IDENT> ::= <UCID> ;
         <UCID>
      <LCID> ::= %LCID ;
         lcid-atom (%LCID)
      <UCID> ::= %UCID ;
         ucid-atom (%UCID)
      <UCID_OPTION> ::= ;
         void ()
      <UCID_OPTION> ::= <UCID> ;
         <UCID>
      <NODE_ID> ::= <LCID> ;
         <LCID>
      <NODE_ID> ::= <UCID> #:: <LCID> ;
         node_id (<UCID>, <LCID>)
      <PHYLUM_ID> ::= <UCID> #:: <UCID> ;
         phylum_id (<UCID>.1, <UCID>.2)
      <SET_NAME> ::= <UCID> ;
         <UCID>
      <SET_NAME> ::= <SET_NAME> #$ <UCID> ;
         set_name (<SET_NAME>, UCID>)
      <LCID_S> ::= <LCID> ;
         lcid_s-list ((<LCID>))
      <LCID_S> ::= <LCID_S> #, <LCID> ;
```

```
         lcid_s-post (<LCID_S>, <LCID>)
      <SIMPLE_REL_OP> ::= #: ;
         rel_op-atom (':')
      <SIMPLE_REL_OP> ::= #-> ;
         rel_op-atom ('->')
      <SIMPLE_REL_OP> ::= %RELOP ;
         rel_op-atom (%RELOP)

   abstract syntax
      void  -> implemented as SINGLETON ;
      lcid  -> implemented as IDENTIFIER ;
      ucid  -> implemented as IDENTIFIER ;
      integer  -> implemented as INTEGER ;
      string  -> implemented as STRING ;
      node_id  -> UCID LCID ;
      phylum_id  -> UCID UCID ;
      set_name  -> SET_NAME UCID ;
      lcid_s  -> LCID + ... ;
      rel_op  -> implemented as IDENTIFIER ;
      UCID  ::= ucid ;
      LCID  ::= lcid ;
      NODE_ID  ::= node_id lcid ucid ;
      PHYLUM_ID  ::= phylum_id ;
      SET_NAME  ::= ucid set_name ;
      IDENT  ::= lcid ucid ;
      NUMBER  ::= integer ;
      LCID_S  ::= lcid_s ;
end chapter ;

abstract syntax
   meta  -> implemented as IDENTIFIER ;
   undef  -> implemented as IDENTIFIER ;
   comment  -> implemented as STRING ;
   comment_s  -> COMMENT + ... ;
   COMMENT_S  ::= comment_s ;
   COMMENT  ::= comment ;

   EVERY  ::=
      undef lab_rel_op lcid ucid integer meta comment #pre import
      import_item_s set_renaming void list_of brace var_decl rule
      inf_rule provided do node #atom set_name #list phylum_renaming
      node_id where lcid_s use local_s rule_s premise_s exp_s
      function_s define comment_s infos sequent program body relation
      rel_op set named global_s renaming_s operator_renaming string
      op_type type_s anonymous phylum_id star proposition ;
```

```
end definition
```

A.2 TYPOL.lex

```
"|->"|"->"|"=>"|"="|"<="|"<-"|">="|"<=>"|"<"|">"|"<<"|">>"|"|="|
"<->"|"(-"|"-)"|"?"          return(RELOP);

"---"[-]*                    return(LINE);

[A-Z][A-Za-z0-9_]*'*         {lowercases(); return(UCID);} ;

[a-z#][A-Za-z0-9_]*'*        {lowercases(); return(LCID);} ;

\"([^\"\n]*|\"\")*\"         {shiftleft(); suplast(); return(STRING) ;}
;

[0-9][0-9_]*              return(NUMBER);

--[^-^\n]*[^\n]*             {shiftleft();shiftleft();
                              fprintf(yaccout,"-1\n%d\n%s\n",yyleng,yytext);

\n                           fprintf(yaccout,"-2\n");

.                            {fprintf(yaccout,"-3\n%s\n",yytext);};
```

B Known problems or bugs

Here is a list of problems that may appear while using Typol in Centaur v0.5. Of course this list is not complete.

- The operators **no_tree**, **meta**, **comment** and **comment_s** may not be used in a Typol specification. There do not appear in the prolog table extracted from the Metal specification.
- Type-checking a Typol program may be very slow when many **define** declarations are used.
- Type inclusion is not yet implemented.

```
use TYPOL;
var x :  TYPOL::UCID;
|- type_s[x];
```

  The operator type_s is declared in the Metal specification of Typol as a list of TYPOL::TYPE. The variable x is declared with type TYPOL::UCID that is included in TYPOL::TYPE. But the type-checker detects a type error.

  The same problem may appear when an undeclared variable occurs more than once in a Typol rule. In this case, it may be necessary to use different variables, and to force them to be equal by using a predicate (EQ) that must be define to be the Prolog predicate "=".
- Type restrictions are not (yet) checked dynamically.
- The phylum EVERY must be explicitly declared in the Metal specification of a given formalism before the generation of the corresponding Prolog tables.
- The name of a file containing a Typol program must be the name of that program, and the polish file must exists (with a name in uppercase) if you want to use the Typol debugger.
- Don't forget to reload the generated Prolog file when a Typol program is recompiled. This is not yet done automatically, and may be done by using the "**Restore**" button in the Prolog menu (see section 6.3.3 page 34).
- If the Typol debugger is used, the polish file and the Prolog generated code must be coherent. So don't forget to save the polish file when a Typol program is recompiled.
- The syntax of Typol allows to use phylum identifiers as variable names, but this is not yet accepted by the type-checker.

Index

References

[1] G. Berry, Ph. Couronne, G. Gonthier, *Synchronous Programming of Reactive Systems: An Introduction to ESTEREL*, Proceedings of the First France-Japan Symposium on Artificial Intelligence and Computer Science, Tokyo, North-Holland, October 1986, and INRIA Research Report n^o 647.

[2] G. Berry, F.Boussinot, Ph. Couronne, G. Gonthier, *ESTEREL v2.2 System Manuals*, Collection of Technical Reports, Ecole des Mines de Paris, Sophia Antipolis, 1986.

[3] G. Cousineau, P.-L. Curien, M. Mauny, *The Categorical Abstract Machine*, LITP Report 85-8, University of Paris VII, January 1985, also in Proc. of the IFIP conference on "Functional Programming Languages and Computer Architecture", LNCS 201, Nancy, France, September 1985.

[4] J. Despeyroux, *Proof of Translation in Natural Semantics*, in Proc of "Symposium on Logic in Computer Science", Cambridge, Massachussets, June 1986, and INRIA Research Report n^o 514, Rocquencourt, France, April 1986.

[5] Th. Despeyroux, *Spécifications sémantiques dans le système Mentor*, Thèse de 3^o cycle, Université Paris-XI, Orsay, France, October 1983.

[6] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, *Programming environment based on structured editors: The Mentor experience*, INRIA Research Report n^o 26, Rocquencourt, France, July 1980.

[7] V. Donzeau-Gouge, G. Kahn, B. Lang, *A complete machine checked definition of a simple programming language using denotational semantics*, INRIA Research Report n^o 330, Rocquencourt, France, October 1978.

[8] G. Gentzen, *The Collected Papers of Gerard Gentzen*, E. Szabo, North-Holland, Amsterdam, 1969.

[9] B. Mélése, *Métal, un language de spécification pour le système Mentor*, T.S.I., AFCET, October 1982.

[10] P. Mosses, *SIS: Reference Manual and User guide / Operating Notes / Tested Examples*, Report DAIMI MD-30/32/33, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[11] L. Naish, *Negation and Control in Prolog*, LNCS 238, Springer-Verlag, 1986.

[12] F. Pereira, D. Warren, D. Bowen, L. Byrd, L. Pereira, *C-Prolog User's Manual, Version 1.2*, EdCAAD, Department of Architecture, University of Edinburgh, U.K., 1983.

[13] G. D. Plotkin, *A structural approach to operational semantics*, Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

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

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

[16] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, *CENTAUR : The system*, INRIA Research Report n° 777, Rocquencourt, France, December 1987.

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