Executable specification of static semantics

Thierry Despeyroux

INRIA

CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. 3) 954 90 20

Rapports de Recherche

N° 295

EXECUTABLE SPECIFICATION OF STATIC SEMANTICS

Thierry DESPEYROUX

Mai 1984

EXECUTABLE SPECIFICATION OF STATIC SEMANTICS

Thierry DESPEYROUX

I.N.R.I.A.
Route des Lucioles
Sophia-Antipolis
F-06565 Valbonne

Résumé:

Dans ce rapport, nous analysons comment des spécifications sémantiques écrites à l'aide de règles d'inférence peuvent être exécutées. Nous présentons un nouveau formalisme de spécification sémantique, appelé Typol, fondé sur les notions de recherche de motifs, de règle d'inférence et de sémantique opérationnelle. Les programmes Typol sont compilés en Prolog afin de générer des vérificateurs de types, des interpréteurs, etc..., directement à partir de leur spécifications. Nous présentons deux expériences réalisées à l'aide de Typol: la vérification des types dans un langage de type Algol (Asple), et l'inférence de types en ML.

Abstract:

In this report, we discuss how semantic specifications written using inference rules may be executed. We present a new formalism for specifying semantics, called Typol, based on notions of pattern-matching, inference rules and operational semantics. Typol program are compiled into Prolog to create executable type-checkers, interpreters, etc... directly from their specifications. We present two experiments carried out with Typol: static verification of an Algol-like language (Asple), and type inference in ML.

EXECUTABLE SPECIFICATION OF STATIC SEMANTICS

Thierry DESPEYROUX

I.N.R.I.A.
Route des Lucioles
Sophia-Antipolis
F-06565 Valbonne

1. Introduction

The Mentor system, developed at I.N.R.I.A., is a general syntax-directed editor ([Dhkl175], [Dhkl80]). It is interactive, programmable, and language-independent [Klmm82] and it allows the simultaneous manipulation of several formalisms. It is used for a large variety of activities such as developing of new languages, of programs and technical reports, building preprocessors, translation, porting programs, etc... The concrete and abstract syntax of new formalisms are specified in the meta-language Metal [Klmm82]. Up until now, however, tools such as type-checkers have been written in the low-level Mentor manipulation language Mentol. It is desireable to introduce into such a system a specification language for the semantic aspects of programming languages.

Our first experience in this area was a language named Formol [Des83]. Formol is an Ada-like specification language specially designed for writing denotational semantic definitions of programming languages. This language has been used to write the formal definition of Ada ([Inria80], [Don82]). However, specifications written in Formol are very large programs. They are heavy and exceedingly low-level to our taste, and we have been looking for a more convenient specification language.

In this paper, we present our second experience. The new language, called Typol, is based on notions of pattern-matching, inference rules and operational semantics [Plo81]. It is a powerful and compact formalism for specifying semantics. Typol programs are compiled into Prolog to create executable type-checkers, interpreters, etc... directly from their specifications.

2. Operational semantics and inference rules

In the most recent and more abstract approach to operational semantics, one uses inference rules to axiomatize the transitions of an abstract machine [Plo81]. We recall here the main aspects of this method, using the well known example of binary numbers. Binary numbers are sequences of **0** and **1** satisfying the following abstract syntax:

$$\begin{aligned} &\textit{number} ::= \textit{digit} \mid \textit{number number} \\ &\textit{digit} \quad ::= \mathbf{0} \mid \mathbf{1} \end{aligned}$$

We use two syntactic sets:

$$\begin{aligned} &\mathbf{NUMBER} = \textit{non-empty sequences of digits} \\ &\mathbf{DIGIT} \quad = \{\mathbf{0}, \mathbf{1}\} \end{aligned}$$

and one semantic set:

$$\mathbf{INTEGER} = I\!N \ .$$

The semantic value of the (syntactic) numeral **0** is the integer *0*, and the length of this string is *1*, so we write:

$$\mathbf{0} : 0, 1 \qquad \text{(A1)}$$

In the same way, we have:

$$\mathbf{1} : 1, 1 \qquad \text{(A2)}$$

A1 and A2 are axioms. An inference rule describes how to compute the semantic value of other numbers:

$$\frac{\mathbf{n} : v, l \qquad \mathbf{n'} : v', l'}{\mathbf{n\,n'} : v\ 2^{l'} + v' \ , \ l + l'} \qquad \text{(R1)}$$

$$\mathbf{n}, \mathbf{n'} \in \mathbf{NUMBER}$$
$$v, v', l, l' \in \mathbf{INTEGER}$$

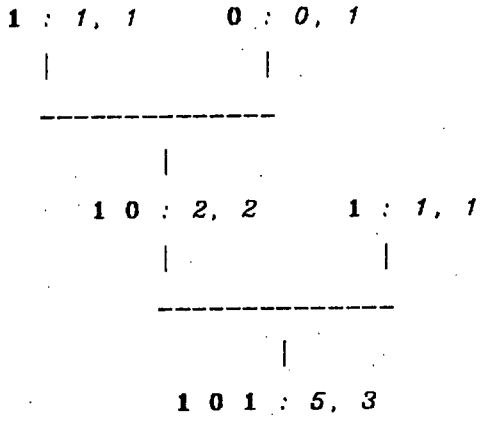
Rule R1 may be paraphrased as follows:

- if v is the value of **n**, and l its length
- if v' is the value of **n'**, and l' its length

then the semantic value of **n n'** is $v2^{l'}+v'$, and its length is $l + l'$.

Compared with formalisms used in attribute grammars ([Knu68], [Pau81], [Rep82]), inference rules are very compact and elegant. Their compactness comes from the fact that the propagation of information is done by pattern-matching. In the attribute grammar framework, this propagation is described explicitely. It seems to us more natural to describe and understand how things work using inference rules rather than attributes.

To derive the semantic value of any number in this deductive system we just need to build a proof tree. For example, the semantic value of **1 0 1** is:



There are several abstract representations for a given binary number, because of the abstract syntax we have chosen; but for each representation there is a single proof tree deriving its semantic value.

This method of associating a semantic value to a syntactic object is very general and elegant, and seems particulary intuitive in the context of type-checking. It has been used in many theoretical papers ([Dm82], [Mqs82]); here we have attempted a direct implementation.

3. Typol: a specification language for the Mentor system

To implement the method we have just described, we have defined a specification language, called Typol, for writing inference rules. Since Typol specifications are used to give the semantics of formalisms, we have to describe the syntactic constructs of these formalisms.

3.1. Handling of syntactic constructs

The objects manipulated by Mentor have a tree structure, and all primitives in the system are defined with respect to this structure. In other words, abstract syntax trees are the standard representation of programs in Mentor. Hence the basic objects manipulated by Typol programs are tree-patterns or schemes. We give here several examples of schemes, in which lowercase identifiers denote operators (nodes) of the abstract syntax, and uppercase identifiers denote tree-variables. **-atom** is a tree constructor for atomic trees (leaves), **-list** and **-pre** are tree constructors for lists.

fixed arity operator:	*ifthenelse(EXP, STM1, STM2)*	
nullary operator:	*void()*	
atomic operator:	*id-atom(X), id-atom('a'),*	
	number-atom(45)	
list operator:	*ids-list(())*	*(empty-list)*
	ids-list((ID1, ID2))	*(list of two elements)*
	ids-pre(ID, IDLIST)	*(ID followed by IDLIST)*

Schemes may be as deep and complicated as necessary. The abstract syntax of a given formalism is introduced in a Typol specification by a **use** clause that imports information from the relevant Metal specification. At this point, it is also possible to rename some of the operators to avoid ambiguities when more than one formalism is used. A dot notation may be also used to refer to the operator of a given abstract syntax, for the same reason.

With the following declaration:

use *PASCAL* **renaming** *program* **in** *pprog;*

both *pprog* and *program.PASCAL* denote the Pascal operator *program*.

3.2. Variables and types

The second class of objects in Typol is variables. These variables are typed variables and must be declared (although tree-variables in patterns do not need to be declared). Types of variables may be predefined (*INTEGER*, *BOOLEAN*) or private (these are implemented in another language, Pascal for example). A type may also be a formalism, if the values of this type are abstract syntax trees.

```
use PASCAL;
n    : INTEGER;   --predefined type
env  : ENV;       --private type
t    : PASCAL;    --formalism
```

3.3. Inference rules

The body of a Typol specification is a collection of inference rules. The upper and lower part of these rules are made of predicates.

$$\Rightarrow\frac{P1\ \&\ \ldots\ \&\ Pn}{Q} \qquad or \qquad P1\ \&\ \ldots\ \&\ Pn \Rightarrow Q$$

A rule with an empty upper part is an axiom. Predicates come in two kinds, as in the following example in which we describe the semantics of binary numbers.

```
program BINARY-SEMANTICS is
use BINARY-NUMBERS;
v, v', v'', l, l', l'' : INTEGER;

=> (digit-atom(0) : 0, 1);
=> (digit-atom(1) : 1, 1);

   (N : v, l) & (N' : v', l') & (v'', l'' = f(v, v', l, l'))
=>-------------------------------------------------------
                (number(N, N') : v'', l'');
end;
```

The predicate *(N : v, l)* may be read as: *using the inference rules we can deduce the value v and the length l from the tree N.*

The predicate *(v'', l'' = f(v, v', l, l'))* may be read as: *let <v'', l''> be the value of f(v, v', l, l').* In this predicate, *f* is an auxiliary function that has to be defined in another section of the Typol program: $f(v, v', l, l') = \langle v\,2^{l'}+v', l+l' \rangle$.

3.4. Modularity

An inference-rule system is a pretty way of specifying a semantic function. Within this system the semantic functions are not named, and are heavily overloaded. (This idea of not giving names to the semantic functions has been suggested in [Pra76].) However, if we want to use another (external) semantic function from inside a system, we need to name this semantic function. In other words, we need to switch to another inference system to prove a particular predicate.

In Typol, it is possible to give a name to a collection of rules and to switch from one set of rules to another.

```
set UPDATE is
  ...              --this set describes the update
                   --operation on environments
end;

UPDATE(env, id-atom(X), type : env1)          (P1)
```

Predicate P1 may paraphrased as follows: *Prove (env, id-**atom**(X), type : env1) using the set of rules called UPDATE*, or *switch to the set UPDATE to prove this predicate.*

3.5. Actions

Sometimes, we need to execute an action when some particular rule is used. Examples of such actions are printing error messages, debugging... These actions might be side effects of auxiliary functions called in the upper part of the rules, or special predicates. But such actions are not part of the semantic specification, so that we much prefer to attach them to the inference rules, to make it clear that they do not interfere with proof trees.

This stategy is already used in other systems, such as Yacc [Joh78], which is a system for generating parsers. In this system, it is possible to attach actions to the production rules of the grammar. These actions may be used, for example, to build the abstract syntax tree of a program during parsing.

We have included this mechanism in Typol, so that it is possible to attach an action to an inference rule. Such an action is executed whenever the rule is used. An action can take as parameters the objects used inside the rule, but it cannot modify them. Thus the pure rule is

separated from its side effects.

```
  (r = square(s))
=>--------------
    (s : r);     {if r<0 then
                    print('humm, there is a bug somewhere')}
```

3.6. The Typol compiler

To be executed, Typol programs are translated into Prolog [Bjm 83]. The two languages are very similar, but there are significant differences.

- Types

Typol objects are typed, while Prolog doesn't have any static type-checking. A further difference is that Typol distinguishes *in* and *out* parameters in a predicate. *Out*-parameters are objects which are deduced from *in*-parameters; this deduction may be done only if *in*-parameters are bound. In a Typol predicate, *in* and *out* parameters are separated by the colon sign: *in*-parameters are on the left of the colon, while *out*-parameters are on the right. Note that the parameters of a predicate are not only variables or constants, but may also be tree-patterns.

We call the *profile* of a Typol inference rule the types of the parameters of the predicate in the lower part of the rule, together with their kinds (*in* or *out* parameters). The profiles of the inference rules are not given explicitly by the users, but are deduced by the compiler. Since predicates may have different profiles, predicates and inference rules are objects which are heavily overloaded. In particular, the colon sign of the predicates is overloaded.

Let us show an example in which it is natural to use overloading. The "semantics" of an identifier (i.e., what has to be done with it) may depend on the context. If an identifier occurs in a declarative part, you have to declare it with its type in the current environment:

```
   (e' = declare('X', e, t))
=>------------------------                    (R2)
    (id('X'), e, t : e')
```

On the other hand, if an identifier occurs in an expression, you want to obtain its type in the current environment, (and perhaps declare it with a special type if it was not already declared):

$$\frac{(t,\ e' = type\text{-}of('X', e))}{(id('X'),\ e : t,\ e')} \quad (R3)$$

The profiles of rules R2 and R3 are:

$$R1: \quad TREE \times ENV \times TYPE \rightarrow ENV$$
$$R2: \quad TREE \times ENV \rightarrow TYPE \times ENV$$

Much of the elegance of Typol (and of the inference rule formalism) is based on this systematic use of overloading. However, to avoid ambiguities in the generated Prolog program, the Typol compiler must resolve this overloading, and generate different names for different occurences of the colon sign.

- Order of rules

We discuss now the problem of chosing which rule to apply to prove a particular predicate. Some schemes may be instances of more than one other scheme. And this may create ambiguous choices, as in the following example:

$$\frac{\ldots}{(f(X,\ int\text{-}atom(1)) : \ldots)} \quad (R4)$$

$$\frac{\ldots}{(f(int\text{-}atom(0),\ Y) : \ldots)} \quad (R5)$$

With these two rules, what is the semantic value of

$$f(int\text{-}atom(0), int\text{-}atom(1)) \quad ?$$

Both rules R4 and R5 can be applied. If the system of rules is not confluent, these two rules may give two different results. Thus, as the semantic value of $f(0, 1)$ is expected to be unique, we have to prove that the system is confluent. To keep clear of this problem, we add a third rule to our system:

```
...
=>-----------------------------------                    (R6)
  (f(int-atom(0), int-atom(1)) : ...)
```

Now, there is no more problem, because rule R6 is a best matching choice. In fact we have closed our set of patterns in the lower part of the rules under unification.

This strategy is general in Typol: we want to chose the *best matching choice*. Thus we extend the notion of closure under unification to all the *in*-parameters of the predicates in the lower parts of the rules. (We have to be careful in this extension because Typol variables are typed, and two variables with different types do not unify.) The Typol compiler checks for the existence of a best-matching rule. This may seem a strange constraint, but in fact it insures that the order of rules in a Typol specification is not significant.

In Prolog, the order of clauses is important because the first encountered matching rule is applied, even if more than one clause matches. Thus the Typol compiler has to sort the generated clauses, so that the best matching choices always occur first.

- An example

As an example, the translation of the Typol program given above is:

```
BINARY-SEMANTICS(digit(1), INTEGER(1), INTEGER(1))-//.
BINARY-SEMANTICS(digit(0), INTEGER(0), INTEGER(1))-//.
BINARY-SEMANTICS(number(*N1, *N2), INTEGER(*v), INTEGER(*l))
  -BINARY-SEMANTICS(*N1, INTEGER(*v1), INTEGER(*l1))
  -BINARY-SEMANTICS(*N2, INTEGER(*v2), INTEGER(*l2))
  -f(INTEGER(*v1), INTEGER(*v2), INTEGER(*l1), INTEGER(*l2),
     INTEGER(*v), INTEGER(*l))-//.
```

The first Typol compiler was written in Pascal. Then the compiler was rewritten in Typol itself. In fact, the formalism is quite adequate for this purpose.

4. Static semantics of Algol-like languages

Most experiments carried out with Typol until now have dealt with static verification of programs. Inference rules seem to be a very elegant and compact way to specify type-checkers. One of the reasons is that an important part of type-checkers consists of navigation in the abstract syntax tree. In Typol, control is achieved by pattern-matching and very little energy must be devoted by the programmer to navigation.

We give here examples of rules for the static semantics of the programming language Asple [Dkl78]. This language is a toy language with simple statements such as the while-statement, if-statement, assignment, and declaration of variables. Possible types for these variables are predefined types (integer, boolean) and references, as in Algol68. The complete specification of the static semantics of Asple in Typol is given in [Des83] and contains 23 rules or axioms.

An Asple program contains two parts: a declarative part and a statement part. During elaboration of the declarative part, an environment is built, which is then used to check the statement part. This is specified in Typol by the following rule:

```
    (e0 = init-env()) & (DECLS, e0 : e1) & (STMS, e1 :)
=>-------------------------------------------------
                (program(DECLS, STMS) :);
```

The nullary function *init-env* returns an empty environment.

Treatment of lists is also easy, since elaboration of declarations is linear in Asple. This is expressed with two rules:

```
    (e, FIRST-DECL : e1) & (e1, LIST-DECL : e2)
=>-------------------------------------------
    (e, decl-s-pre(FIRST-DECL, LIST-DECL) : e2);

=>(e, decl-s-list(()) : e);
```

Declaration of variables may be done in the following way:

```
  (e1, ok = declare(e, X, TYPE))
=>------------------------------                    (R7)
  (e, var-decl(X, TYPE) : e1);
       {if not ok
           then printerror('identifier already declared',X)}
```

In rule R7, *declare* is an auxiliary function returning a pair of values. The first is the result of declaring the identifier X with type *TYPE* in the environment e. The second is a boolean value that indicates whether the identifier was already declared or not. This function may be written apart from the Typol program, for example in Pascal or Prolog. Alternatively, one may describe environments, and operations on environments, in Typol with inference rules. In this case, the upper part of the rule given above will be:

$$DECLARE(e, X, TYPE : e1, ok).$$

This mean that we have to switch to the inference-rule system called *DECLARE*, and in this system prove that *(e, X, TYPE : e1, ok)*.

We give now the checking rule for a while-statement:

```
  (e, EXP : t, n) & (e, STATEMENT :)
=>--------------------------------
  (e, while(EXP, STATEMENT) :);
       {if t # 'bool'
           then printerror('type of expression must be boolean',EXP)}
```

Checking an expression returns a pair $<t,n>$, where n is the number of references (0 for a constant, 1 for an integer or boolean variable, 2 for a ref variable...) and t is the basic type of the expression. Checking expressions is specified by the following rules:

```
=>(e, number-atom(N) : 'int', 0);

=>(e, boolean-atom(B) : 'bool', 0);

  (t, n  = type-of(X, e))
=>-------------------------
  (e, ident-atom(X) : t, n);
       {if t = 'error'
           then printerror('undeclared identifier', X)}

  (e, EXP1 : t1, n1) & (e, EXP2 : t2, n2)
              & (t, n = result-type(t1, n1, t2, n2)
=>-------------------------------------------------
        (e, plus(EXP1, EXP2) : t, n);
      {if t = 'error' and t1 # 'error' and t2 # 'error'
          then printerror('bad mixture of types',plus(EXP1,EXP2))}
```

It is clear that the description of the static semantics of Asple by inference rules is very easy and natural. This is also the case for more complicated languages such as Pascal with blocks, procedures... Handling of environments is indeed more complicated, and the number of rules increases, but the description remains clear.

5. Type inference in ML

Most usual programming languages require the declaration of which objects will be used and what their types are. A large part of static semantics of these languages is type-checking. However, in ML, the meta-language of the LCF proof system [Gmw79], objects do not need to be declared. The ML interpreter must compute (deduce) the types of objects according to their use.

ML objects may be polymorphic. Thus ML types may be polytypes, i.e. may contain type variables. Since an ML expression may have many valid types, the types computed by the interpreter must be as general as possible ([Mil78], [Dm82]).

We discuss here the type inference problem for the applicative part of ML, which contains in essence the whole problem of typing in ML. Our sub-language contains the following expressions:

```
<exp> :=  <ident> | <number> | true | false
        | (<exp> <exp>)                          -- application
        | λ <ident> . <exp>                      -- abstraction
        | let <ident> = <exp> in <exp>
        | if <exp> then <exp> else <exp>

        where <ident> are the usual identifiers,
          and <number> are integers.
```

Possible type expressions for this language are:

```
<s> ::=  <t>
       | ∀ <v> <s>

<t> ::=  <v>
       | int | bool
       | <t>-list
       | <t> → <t>

          where <v> are type variables
            (noted with greek letters)
```

Let *exp* be an ML expression. If *exp* has type s (we write *exp* : s) then *exp* has every type obtained by substitution of types for bound type variables in s. Thus, if x has type $\forall\ \alpha\ (\alpha)$, it also has the following types:

$$int,\ bool,\ \beta \rightarrow \beta,\ \forall\ \beta\ (\beta \rightarrow \beta)\ldots$$

As examples, we give the types of some predefined identifiers:

```
nil  : ∀ α (α-list)
cons : ∀ α (α → (α-list → α-list)
head : ∀ α (α-list → α)
tail : ∀ α (α-list → α-list)
null : ∀ α (α-list → bool)
```

Let A be a set of assumptions of the form $\{x : \sigma\}$. We define $A_x = A - \{x:\sigma\}$.

Let $\sigma = \forall\alpha_i\ \tau$, and $\sigma' = \forall\beta_i\ \tau'$, where $\tau' = S\tau$ and S is a substitution of types for some of type variables α_i, and the β_i are not free in σ; then $\sigma' < \sigma$.

The problem is to give an inference system for deducing types of ML expressions. Let us consider the system given in [Dm82]:

$$TAUT:\ A \vdash x : \sigma \quad (x{:}\sigma \text{ in } A)$$

$$INST:\ \frac{A \vdash e : \sigma}{A \vdash e : \sigma'} \quad (\sigma > \sigma')$$

$$GEN:\ \frac{A \vdash e : \sigma}{A \vdash e : \forall\, \alpha\ \sigma} \quad (\alpha \text{ not free in } A)$$

$$COMB:\ \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\ e') : \tau}$$

$$ABS:\ \frac{A_x \cup \{x{:}\tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \to \tau}$$

$$LET:\ \frac{A \vdash e : \sigma \qquad A_x \cup \{x{:}\sigma\} \vdash e' : \tau}{A \vdash let\ x{=}e\ in\ e' : \tau}$$

$$IF:\ \frac{A \vdash e1 : bool \qquad A \vdash e2 : \tau \qquad A \vdash e3 : \tau}{A \vdash if\ e1\ then\ e2\ else\ e3 : \tau}$$

(The IF rule was added to this system.)

This system is adequate for proving the type of expressions, but it is not adequate for computing these types. In this system rules *INST* and *GEN* may be applied at any time, and there is an ambiguous choice between these two rules. Both of them have the same *in*-part in the bottom predicate. Thus, this system is not accepted in Typol.

We replace this system by another one in which instantiation and generalization are done explicitly where they are needed:

- The TAUT rule is always followed by an instantiation.
- The ABS rule is always followed by a generalization.

These modifications come from the fact that all occurrences of a λ-bound identifier must be ascribed the same type as their binding occurence, and that let-bound occurences of an identifier must be ascribed types which are possibly different instances of the type of the binding occurence. Thus, free variables coming from λ-bound identifiers must be bound outside λ-expressions.

$$TAUT'\colon\ A \vdash' x : \sigma' \quad (x:\sigma \text{ in } A \text{ and } \sigma' < \sigma)$$

$$COMB'\colon\ \frac{A \vdash' e : \sigma' \to \sigma \quad A \vdash' e' : \sigma'}{A \vdash' (e\ e') : \sigma}$$

$$ABS'\colon\ \frac{A_x \cup \{x:\tau\} \vdash' e : \sigma}{A \vdash' \lambda x.e : \forall \alpha_i \quad (\tau \to \sigma)} \quad (\text{free variables of } \tau \text{ are bound})$$

$$LET'\colon\ \frac{A \vdash' e : \sigma \quad A_x \cup \{x:\sigma\} \vdash' e' : \sigma'}{A \vdash' \text{let } x=e \text{ in } e' : \sigma'}$$

$$IF'\colon\ \frac{A \vdash' e1 : bool \quad A \vdash' e2 : \sigma \quad A \vdash' e3 : \sigma}{A \vdash' \text{if } e1 \text{ then } e2 \text{ else } e3 : \sigma}$$

These two systems are not, strictly speaking, equivalent, but if A |- ' e :σ then A|-e:σ (the new one is *sound*) and if A |- e :σ then A |-' e : σ' with σ'>σ (and it is *complete*). Notice that the new system computes a more general type that the first one because types are generalized. For example, we have:

$$\vdash \lambda x.x : \alpha \to \alpha$$
$$\vdash' \lambda x.x : \forall \alpha (\alpha \to \alpha)$$

Now this new system is an acceptable Typol program. The Typol compiler produces mechanically an ML type-checker that works as expected. In fact, this type-checker works in a manner that is similar to the type

assignment algorithm "W" given in [Dm82]. In the algorithm "W", instantiations are done by the unification algorithm of Robinson [Rob65], which is used in Prolog systems too, but it seems easier to write and understand an inference system than an algorithm like "W".

6. Other applications of Typol

We have written in Typol a type-checker for a substantial sublanguage of Pascal. In an interactive programming environment such as Mentor, it is not reasonable to check a whole program after each modification. So we are interested in making the type-checker incremental. To make a type-checker incremental, we must know how far program modifications propagate. This may be computed dynamically, as in the Synthesizer, a system using attribute grammars [Rep82]. We choose to compute the propagation statically in the Typol program, using a dependence graph on parameters of the predicates in the inference rules. Nothing needed to be changed in the original Typol program to make it incremental. This is a highly desirable feature.

Further experiments have been carried out and will be described in a forthcoming paper:

- specification of the dynamic semantics of Asple, and generation of an interpreter for this language,
- specification of the dynamic semantics of a subset of P–code and generation of a P–code interpreter,
- specification of Asple → P–code translation, and generation of an Asple → P–code translator.

7. Conclusion

In this paper we have presented three main ideas:

- Semantic specifications written using inference rules may be executed. The method may be compared with the approach taken in the SIS system of Peter Mosses [Mos78].
- A Prolog system is adequate for executing such a specification.
- Intensive use of overloading make semantic specifications easier and more elegant.

Typol seems to be a very intuitive way to introduce semantic specifications into a system like Mentor. We would like to use more of Prolog's power (backtracking for example) to specify the static

semantics of programming languages with overloading, such as Ada. Other examples will be needed to improve the Typol formalism and to see exactly what its limits are, both from a theoretical and a practical standpoint.

Acknowledgements

I would like to thank Gilles Kahn for helpful suggestions and extensive reviewing of this paper, and John Reynolds for improving my english.

References

Bjm 83 G. Barberye, T. Joubert, M. Martin, *Manuel d'utilisation de Prolog/Cnet - version Pascal/Multics 2.00* C.N.E.T. NT/PAA/CLC/ICS/1058, Paris, October 1983

Des 83 Th. Despeyroux, *Spécifications sémantiques dans le système Mentor,* Thèse de 3ème cycle, Université Paris-XI, Orsay, October 1983

Dhkl 80 V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, *Programming environments based on structured editors: the MENTOR experience,* Rapport de recherche no 26, Inria, July 1980

Dhkll 75 V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, J.-J. Lévy, *A structure oriented program editor: a first step toward computer assisted programming,* International Computing Symposium, North-Holland Publishing Company, 1975

Dkl 78 V. Donzeau-Gouge, G. Kahn, B. Lang, *A complete machine-checked definition of a simple programming language using denotational semantics,* Rapport de recherche no 330, Inria, October 1978

Dm 82 L. Damas, R. Milner, *Principal type-schemes for functional program,* ACM, 1982

Don 82 V. Donzeau-Gouge, *Les raisons des choix dans la définition formelle du langage Ada,* Thèse d'état, Université Paris-7 July 1982

Gmw 79 M. Gordon, R. Milner, C. Wadsworth, *Edinburg LCF,* Lecture Notes in Computer Science, Springer-Verlag, 1979

Gor 79 M.S.C. Gordon, *Descriptive technique for denotational semantics,* Springer Verlag, 1979

Inria 80 *Formal definition of the Ada programming language,* Honewell Inc., CII-Honewell Bull, Inria, November 1980

Joh 78 S. C. Johnson, *Yacc: Yet Another Compiler-Compiler* Bell Laboratories, July 1978

Klmm 82 G. Kahn, B. Lang, B. Melese, E. Morcos, *Metal: a formalism to specify formalisms,* Science of Computer Programming, vol. 3, no 2, p. 151-188, August 83

Knu 68 D. E. Knuth, *Semantics of context-free languages*, Mathematical Systems Theory, 2, 2, p. 127-145, 1968, correction in vol. 5, 1, p. 95-96, 1971

Mos 78 P. Mosses, *SIS: a compiler generator system using denotational semantics*, DAIMI, University of Aarhus, 1978

Mqs 82 D. B. MacQueen, R. Sethi, *A semantic model of types for applicative languages*, ACM symposium on LISP and functional programming, p. 243-252, 1982,

Mil 78 R. Milner, *A theory of type polymorphism in programming*, JCSS 17,3, p. 348-375, 1978

Pau 81 L. Paulson, *A compiler generator for semantic grammars*, PhD Thesis, Stanford University, December 1981

Plo 81 G.D. Plotkin, *A structural approach to operational semantics*, DAIMI FN-19, Aarhus university, September 1981

Pra 76 V. Pratt, *Semantical considerations on Floyd-Hoare logic*, 17th Focs, 109-121, Huston, 1976

Rep 82 T. W. Reps, *Generating language-based environments*, PhD Thesis, Cornell University, August 1982

Rob 65 J. A. Robinson, *A machine-oriented logic based on the resolution principle* JACM 12,1, p. 23-41, 1965

Set 77 R. Sethi, *Semantics of computer programs: overview of language definition methods*, Bell Laboratories, September 1977