UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 1218

Programme 1
Programmation, Calcul Symbolique et Intelligence Artificielle

FUNCTIONAL EVALUATION OF NATURAL SEMANTICS SPECIFICATIONS

Isabelle ATTALI
Jacques CHAZARAIN

Mai 1990

RR_1218

FUNCTIONAL EVALUATION OF NATURAL SEMANTICS SPECIFICATIONS[1]

EVALUATION FONCTIONNELLE DE SPECIFICATIONS EN SEMANTIQUE NATURELLE

Isabelle ATTALI
INRIA - Sophia-Antipolis
Route des Lucioles
06565 Valbonne Cedex, France
e-mail: ia@trinidad.inria.fr

Jacques CHAZARAIN
URA I3S & Math. Dpt
University of Nice
06034 Nice Cedex, France
e-mail: jmch@cerisi.cerisi.fr

ABSTRACT:
Typol is a language to implement Natural Semantics inside the Centaur Programming Environment. We explain why Natural Semantics can be considered as a generalization of the Attribute Grammars formalism. This relationship suggests introducing in the Typol setting some concepts from the Attribute Grammars framework. With these concepts, we can provide specialized evaluators to some classes of Typol programs, instead of the general Prolog machine currently used to execute such programs. More precisely, we prove that under acceptable conditions, the unification process is no longer required and can be replaced by a simple pattern-matching mechanism. From this proof method, we deduce a functional implementation of Typol programs. We also propose a language construct called CondMatch, that makes it easier to automatically translate a Typol program into a Lisp function.

RESUME:
Dans le système Centaur, le langage Typol sert à implémenter les spécifications en Sémantique Naturelle. Nous expliquons pourquoi la Sémantique Naturelle peut être considérée comme une généralisation des Grammaires Attribuées. Ce lien suggère l'introduction dans le domaine du langage Typol de concepts issus des Grammaires Attribuées. Ainsi, nous pouvons pouvons décrire des classes de programmes Typol pour lesquelles on peut remplacer l'évaluateur standard utilisant Prolog par des évaluateurs plus spécialisés. Notre résultat principal concerne la définition d'une large classe de programmes Typol pour laquelle on démontre que l'on peut construire la preuve d'un but sans utiliser l'unification mais seulement le filtrage. De cette preuve, nous extrayons, sous certaines hypothèses, une implémentation fonctionnelle des spécifications Typol. Nous proposons également une construction appelée CondMatch qui rend plus systématique la traduction d'un programme Typol en une fonction Lisp.

[1]A shorter version of this paper is to be published in the proceedings of the first International Workshop on Attribute Grammars and their Applications, Paris, Sept. 90

1 Introduction

The description of semantic properties in the natural deduction style advocated by Plotkin [21], has been used by Damas and Milner [8] and developed by Kahn and his group inside the Centaur project [11,17,12,5]. We must also indicate the seminal work of Warren [23].
A semantic specification is represented by a set of inference rules of the form

$$\frac{H_1 \vdash T_1 : S_1 \qquad \cdots \qquad H_n \vdash T_n : S_n}{H \vdash T : S}$$

which constitutes, together with type information, a Typol program [11]. To compute the semantic value, say S_0, of an abstract syntax term T_0, given some initial environment H_0, means to prove the goal sequent $H_0 \vdash T_0 : S_0$ in this deductive system.

In this paper, we study the structure of natural semantics specifications in order to associate functional evaluators to some classes of Typol programs. The natural deduction formalism can be considered, in a sense, as a generalization of Attribute Grammars (AG). More precisely, we explain how to translate a given AG specification into a Typol program. The generated Typol programs verify some specific properties, such as:

- the proof tree of a goal is always isomorphic to the abstract syntax tree of the term T_0;
- there is no link nor constraint within input attributes because they are composed of pairwise distinct variables;
- there is one rule for each abstract syntax operator.

This explains why, in [3,2], some restrictive hypotheses, due to the AG framework, were necessary. In other words, since the usual domains of AG and logic programming are different (values vs terms), the mandatory hypotheses concerned the absence of links and constraints within input attributes. In the present work, these unnatural restrictions are simply released because proofs are entirely established in the logical setting of natural semantics; here, Attribute Grammars are only used as a guideline to suggest new concepts but not as a tool for proofs. This makes it possible to obtain new results under natural hypotheses.

We adapt for the Typol setting some classical notions from AG: input/output attributes, minimal argument selector, Strongly Non Circular programs. We prove the feasibility of evaluating Typol programs in successive passes. Our major result is the definition of a rather large class of Typol programs in which the unification process can be replaced by a less costly pattern-matching process (Theorems 1 & 2). This generalizes the results of [3,2] to a larger class of programs and is closely related to [9,20] in spirit, but not in scope (Prolog evaluation vs functional evaluation). Furthermore, we give, under acceptable hypotheses, a functional implementation (instead of the usual Prolog machine) for Typol programs evaluation.

This paper belongs to a recent field of investigations: relationship between Attribute Grammars and Logic Programming [9,15,22,1].

Section 2 formally defines, with a number of illustrative examples, the natural semantics formalism and outlines a comparison with Attribute Grammars. Section 3 gives an operational semantics to the Typol formalism, defines various notions of circularities, and namely the Strongly Non Circularity, describes a reduction technique useful to transform a circular program (requiring an unification) into separate Strongly Non Circular programs. In Section 4, we present our theoretical results about the unification-free execution of non trivial classes of Typol programs. Finally, in Section 5 we discuss implementation issues for a functional evaluator of Typol programs based on the proof technique of the theorems. Section 6 concludes the paper.

2 Some definitions about the Natural Semantics formalism

A Typol program is essentially a collection of inference rules. Each inference rule is composed of a finite set of premises (which is empty for an axiom) and a conclusion. Premises and conclusion of a rule are relations represented by sequents in the Gentzen Natural Deduction style [12].

The object languages are manipulated via their abstract syntax, defining a many-sorted algebra. This presentation allows quite compact and readable specifications of semantic properties such as static semantics, dynamic semantics, and translations. Moreover, it is an executable formalism because each rule can be translated into an Horn clause.

In order to illustrate our definitions and results, we give in appendix some simplified version of well-known examples of Typol programs such as the dynamic semantics of a small Pascal-like language, the static and dynamic semantics of Mini-ML [5], and a translator from regular expressions to deterministic automata [3,2].

In this section, we describe more formally the notions of abstract syntax, sequents, rules, and we compare the Typol formalism with the Attribute Grammars one.

2.1 Abstract syntax

The abstract syntax of a program deals with the structure of the language constructs, and not with their textual form.

Definition 1 *Given a set P of sorts (or types), a set O of operators with their signature on P, and a set X of variables, we denote by $M(O,X)$ the many-sorted algebra of well-typed terms; we call it* **abstract syntax.**
If p is an operator of signature $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, we say that τ is the **type** *of p.*

Example 1 *Some operators with their signature*
The signature of the **while** *operator from Appendix A is: $EXP \times STMS \rightarrow STM$.*
The signature of the **letrec** *operator from Appendix C is: $PAT \times EXP \times EXP \rightarrow EXP$.*

2.2 Sequents and attributes

Roughly speaking, a sequent expresses the fact that some hypotheses are needed to prove a particular property about an abstract syntax term.

Definition 2 *A* **sequent** *$H \vdash T : S$ is composed of:*

a term T in $M(O,X)$ named the **subject** *of the sequent;*

a tuple $H = (h_1, \cdots, h_m)$ of **inherited attributes** *expressions (using a terminology coming from the relationship with Attribute Grammars).*

a tuple $S = (s_1, \cdots, s_n)$ of **synthesized attributes** *expressions.*

The attribute expressions can also be elements of some term-algebra. We denote by VAR(T) (resp. VAR(H), VAR(S)) the set of variables that occur in the term T (resp. the tuples of terms H, S).

The *names* of the attributes used in a sequent depend only on the type τ of its subject; we note $\mathrm{inh}(\tau)$ the set of attribute names in H and $\mathrm{syn}(\tau)$ set of attribute names in S. We also note $\mathrm{attr}(\tau) = \mathrm{inh}(\tau) \cup \mathrm{syn}(\tau)$ the set of all attribute names for a given sort τ.

Remark 1 *In the rest of the paper, we use the terms "attribute" or "attribute name" to reference the name of an attribute; to talk about the expression associated with an attribute, we say "attribute expression".*

Example 2 *Some attributes*
The following sequent comes from Appendix A:

$$s \vdash \text{while}\,\text{EXP}\ \text{do}\,\text{STMS}\,\text{end} : s'$$

We often write the subject in a concrete syntax, a more readable way of denoting terms.
In this Typol specification, all sequents with a subject of sort STM have the same attribute names:

$$\text{inh(STM)} = \{\text{env}_{in}\} \quad \text{and} \quad \text{syn(STM)} = \{\text{env}_{out}\}$$

Definition 3 *A sequent is said to be* **reduced** *if H and S have at most one element.*

For instance, the previous sequent is reduced as it is usually the case for most of Typol programs.

2.3 Rules and attributes

Typol rules are reminiscent of the Gentzen formalism for logical systems. They indicate how a sequent may be deduced from other sequents.

Definition 4 *A Typol* **rule**

$$\frac{H_1 \vdash T_1 : S_1 \qquad \cdots \qquad H_n \vdash T_n : S_n}{H \vdash T : S} \qquad (r)$$

is composed of:

- *a sequent $H \vdash T : S$ named the* **conclusion** *of the rule;*
- *sequents $H_i \vdash T_i : S_i$ named* **premises** *of the rule;*

The subject T of the conclusion is also the **subject** *of the rule;*
A rule without premises is called an **axiom**.
A rule without conclusion is called a **goal**.

Example 3 *Some Typol rules*

$$r1 : \quad \frac{\rho \vdash \text{E}_1 : \alpha \qquad (\rho, \text{X} \mapsto \alpha) \vdash \text{E}_2 : \beta}{\rho \vdash \text{let}\,\text{X} = \text{E}_1\ \text{in}\,\text{E}_2 : \beta} \qquad (App.\ C)$$

$$r2 : \quad \frac{(\rho, \text{X} \mapsto \alpha) \vdash \text{E}_1 : \alpha \qquad (\rho, \text{X} \mapsto \alpha) \vdash \text{E}_2 : \beta}{\rho \vdash \text{letrec}\,\text{X} = \text{E}_1\ \text{in}\,\text{E}_2 : \beta} \qquad (App.\ C)$$

$$r3 : \quad \frac{s \vdash \text{EXP} : true \qquad s \vdash \text{STMS} : s_1 \qquad s_1 \vdash \text{while}\,\text{EXP}\ \text{do}\,\text{STMS}\,\text{end} : s_2}{s \vdash \text{while}\,\text{EXP}\ \text{do}\,\text{STMS}\,\text{end} : s_2} \qquad (App.\ A)$$

$$r4 : \quad \frac{s \vdash \text{EXP} : false}{s \vdash \text{while}\,\text{EXP}\ \text{do}\,\text{STMS}\,\text{end} : s} \qquad (App.\ A)$$

Definition 5 *We define the* **signature** *of a given Typol rule as (r) by $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, where τ is the type of the term T and each τ_i is the type of the term T_i.*
We call **type** *of the rule the type of its subject.*

In the general case, the signature of a given Typol rule r can be different from the signature of the abstract syntax operator subject of the rule (see rule r4). On the other hand, the subjects of the premises are not necessarily proper subterms of the subject of the rule (see rule r3).

Example 4 *Some signatures of Typol rules*
The signature of r1 is: $EXP \times EXP \to EXP$;
The signature of r2 is: $PAT \times EXP \times EXP \to EXP$;
The signature of r3 is: $EXP \times STMS \times STM \to STM$;
The signature of r4 is: $EXP \to STM$;

As shown in rule r3 with s_1 and s_2, two attributes in different locations in the rule can have the same expression. Thus, we use a sequence (instead of a set) to also take care of the sequent (conclusion or premise) the attribute belongs to, and keep track of its **position**.

Definition 6 *We denote by ATTR(r) the set of attribute positions; it is represented by the the sequence of all attribute names involved in a given rule r of signature* $\tau_1 \times \cdots \times \tau_n \to \tau$:

$$ATTR(r) \;=\; \{attr(\tau), attr(\tau_1), \cdots, attr(\tau_n)\}$$

Remark 2 *A premise of a Typol rule can also be a* **predicate condition** *of the form* $C(\theta_1, \cdots, \theta_n)$ *where each* θ_i *belongs to the same term-algebra than attribute expressions. Such predicate conditions express a restriction on the applicability of a rule.*
For clarity and without loss of generality, we do not take in account such predicate conditions; we only indicate as remarks the modifications necessary to handle the presence of predicate conditions.

2.4 A comparison with Attribute Grammars

We briefly recall the description of semantic specification using Attribute Grammars [18]. See the monograph [9] for more information about Attribute Grammars. The objective is similar to Natural Deduction: given an abstract syntax term T_0 and some initial value H_0 (inherited attributes) we have to compute a semantic value, the associated synthesized attributes S_0, using a system of local equations.

For each operator p of the abstract syntax is given a functional description of the *output attributes* in terms of the *input attributes* (we assume here it is in Bochmann normal form):

$$S = \phi(H, S_1, \cdots, S_n)$$

$$H_k = \psi_k(H, S_1, \cdots, S_n), \qquad k = 1 \cdots n$$

where each attribute H_j, S_k, H and S can be a tuple.

The intuitive meaning of these equations is given in Figure 1 where input attributes have an incoming arrow in the rule box and the output ones have an outcoming arrow:

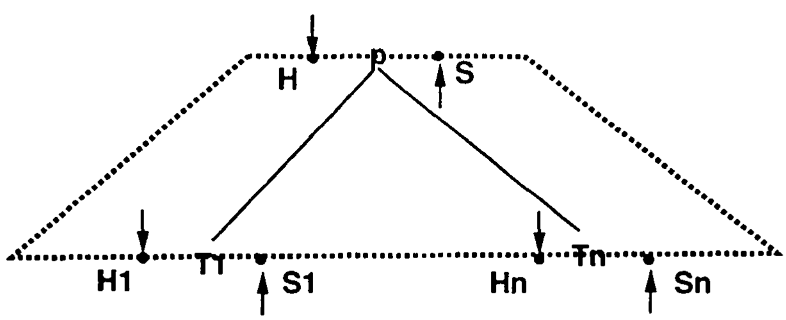

Figure 1: A rule in Attribute Grammars

In the Typol style, we could represent the same local equations by the following (AGr) Typol rule:

$$\frac{\psi_1(H, S_1, \cdots, S_n) \vdash T_1 : S_1 \qquad \cdots \qquad \psi_n(H, S_1, \cdots, S_n) \vdash T_n : S_n}{H \vdash p(T_1, \cdots, T_n) : \phi(H, S_1, \cdots, S_n)} \qquad (AGr)$$

where $H, S_1, \cdots, S_n$ are pairwise distinct variables.
This is sufficient to explain the use of some Attribute Grammars terminology, but in fact, as we shall see later, this kind of Typol rules is very specific. In fact, Typol programs are closer to the so-called *Relational Attribute Grammars* in [8] with an additional built-in splitting of attributes in a sequent.

3 Dependency properties in Typol programs

We are concerned in this section with the evaluation of Typol programs. From a given goal, we build a proof tree yielding the proof of the goal.

We shall study the graph $D(t)$ of information flow in the proof tree t. This graph gives a dynamic way of computing each attribute, starting from given data, going on with the variables depending only on the latter, and so on until all attributes are computed. Note that the graph $D(t)$ defines a dynamic notion of circularity: these successive computations are possible when $D(t)$ is acyclic.

But we are particularly interested by a *static* notion of acyclicity. For that purpose we adapt, from the AG field, the minimal argument selector topic. With this argument selector Γ, we can associate to each Typol rule r a graph $D_{r,\Gamma}$ which includes all the potential dependencies between the attributes of this rule. As a consequence, a Typol pg is called *Strongly Non Circular* if, for each rule r, the graph $D_{r,\Gamma}$ has no cycle, and, of course, we can statically check such a property.

Finally, we build another graph structure $D_{P,\Gamma}$ coming from the combination of all induced dependency graphs $D_{r,\Gamma}$ for all Typol rules in a given program P. With this graph, we define a partition that makes it possible to transform a Typol program into separate Typol programs, each of them yielding a separate phase of the computation.

3.1 Evaluating Typol programs

Evaluate a Typol program means try to prove a goal (a sequent) within the logic defined by the Typol program itself. For instance, one can prove that a given Mini-ML program is well-typed in a given predefined environment.
Such an evaluation leads to the construction of a proof tree. We recall the usual definition of proof trees from logic:

Definition 7 *Let P a Typol program, a* proof tree *t for P is a tree labeled with sequents such that:*

- *the sequent A is the label of a leaf if there exists a substitution σ and an axiom $\frac{}{\alpha}$ of P such that $\sigma(A) = \sigma(\alpha)$.*
- *let A the label of an internal node and let $t_1, \cdots, t_n$ its sons, with respective labeled roots $B_1, \cdots, B_n$, then there exists a substitution σ and a rule $\frac{\beta_1 \quad \cdots \quad \beta_n}{\alpha}$ of P such that $\sigma(A) = \sigma(\alpha)$ and $\sigma(B_i) = \sigma(\beta_i), i = 1, \cdots, n$.*

We say that t is a proof tree for a sequent G if it is a proof tree with a root labeled with G.

In this paper, we focus on the following kind of goals $H_0 \vdash T_0 : S_0$ to be proved:

- H_0 is given;
- T_0 is a *closed term* (an abstract syntax term);
- S_0 is a result to be computed.

Example 5 *A goal*
The sequent

$$\rho_0 \vdash \textit{letrec fact} = \lambda x \cdot \textit{if } x = 0 \textit{ then } 1 \textit{ else } x \star \textit{fact}(x\text{-}1) : \alpha$$

computes the type α of the recursive definition of the factorial function in Mini-ML, in a predefined environment ρ_0. As usually, the subject is written here in concrete syntax to be more readable.

So, in a goal, H_0 and T_0 are input and S_0 is an output. More generally, we extend this notion of input/output to the attributes of a rule because we want to study the proof process and data-flow rules between attributes. Here also, the terminology is consistent with Attribute Grammars.

Definition 8 *For a given Typol rule*

$$\frac{H_1 \vdash T_1 : S_1 \qquad \cdots \qquad H_n \vdash T_n : S_n}{H \vdash T : S} \qquad (r)$$

we define the set input(r) of **input attribute positions** *composed from the attribute positions of H, $S_1, \cdots, S_n$ and the set output(r) of* **output attribute positions** *composed from the attribute positions of S, $H_1, \cdots, H_n$.*
The sets input(r) and output(r) define a partition of the set of all attribute positions $ATTR(r)$.

We also need the sets of input / output variables of a rule r in order to define the flow of informations in a proof tree.

Definition 9 *The sets input_var and output_var of a rule r are defined as follows:*

- $\textit{input\_var}(r) = VAR(T) \cup VAR(H) \cup VAR(S_1) \cup \cdots \cup VAR(S_n)$
- $\textit{output\_var}(r) = VAR(S) \cup VAR(T_1) \cup \cdots \cup VAR(T_n) \cup VAR(H_1) \cup \cdots \cup VAR(H_n)$.

We say that a Typol rule r is **safe** *if:*

- $\textit{output\_var}(r) \subset \textit{input\_var}(r)$;
- $VAR(T_i) \subset VAR(T)$, $i = 1, \cdots, n$.

These input and output sets, together with the notion of safety, induce a local dependency graph on the whole set of attribute positions within a Typol rule r. Roughly speaking, input attribute positions are computed by the outer context and used in r to compute the output attribute positions which are then transmitted to the outer context. This kind of flow ensures all output attributes are computable.
Most of usual Typol programs are composed with safe inference rules. Nevertheless, here is a non safe Typol rule:

Example 6 *A non safe Typol rule from Appendix B*

$$\frac{\rho \cdot [\mathrm{X} : \tau'] \vdash \mathrm{E} : \tau}{\rho \vdash \lambda \mathrm{X}.\mathrm{E} : \tau' \rightarrow \tau}$$

This rule is non safe because the τ' variable has no occurrence among input attributes expressions.

3.2 The local dependency graph of a rule

For a given Typol rule r, we define on the set ATTR(r) of attribute positions a dependency relation D_r as follows:

Definition 10 *Let $a \in input(r)$ and $b \in output(r)$, we say that $a \underset{D_r}{\rightarrow} b$ if the expressions of the attribute positions a and b share a common variable.*

Example 7 *For the rule r2 of Example 3, we have the following local dependency graph:*

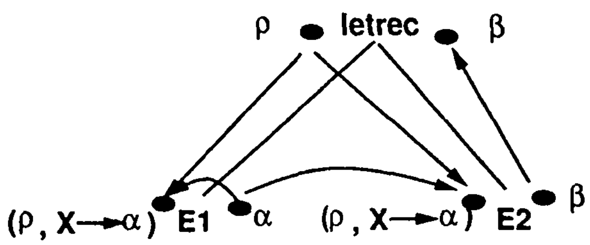

Figure 2: A local dependency graph D_r

By definition, the local dependency graph is a bipartit graph on ATTR(r): the arrows are from input positions to output positions. Intuitively, this relation means that the value of attribute a is needed to compute the value of attribute b.

We also have to take in account possible predicate conditions, so we extend the dependency relation with the links induced by the predicate conditions:

Definition 11 *Two attributes a and b are said* **linked by a predicate condition** $C(\theta_1, \cdots, \theta_n)$ *if there exists indices i and j such that:*

- *the expressions of θ_i and a share a common variable and*
- *the expressions of θ_j and b share a common variable.*

Then, the definition of dependency relation can be naturally extended as follows:

Definition 12 *Let $a \in input(r)$ and $b \in output(r)$, we say that $a \underset{D_r}{\rightarrow} b$ if:*

- *the expressions of the attribute positions a and b share a common variable or*
- *a and b are linked by a predicate condition.*

There can also be common variables between two input attribute positions. This kind of symetrical link is at the origin of the use of unification in some proof process; so we introduce new notions to define a subclass of Typol programs.

Definition 13 *A Typol program is* **link-less** *if, in every rule, neither two input attribute expressions share a common variable, nor are linked by a predicate condition.*

Another feature of Typol programs non compatible with usual Attribute Grammars is the possibility of constraining input attribute expressions to structured values:

Definition 14 *A Typol program is* **constraint-less** *if, in every rule, input attributes are bound to variables.*

Notice that the Typol rule (AGr) associated with an Attribute Grammar verifies these two previous properties; we call such rules "linear" rules according to the following definition:

Definition 15 *A Typol program is* **linear** *if it is link-less and constraint-less.*

The case of linear Typol programs has been more particularly studied in [2,1].

Example 8 *Non linear rules*
The rule r3 of Example 3 does not verify the constraint-less condition because of the constant **true**. *The rule*

$$\frac{\rho \vdash \mathrm{E}_1 : \tau' \rightarrow \tau \qquad \rho \vdash \mathrm{E}_2 : \tau'}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 : \tau}$$

describes the type checker for the **apply** *operator of Mini-ML. It does not verify the link-less condition between the types of E_1 and E_2 (because of the common τ' variable) nor the constraint-less condition (because of the structured term $\tau' \rightarrow \tau$).*

As Typol programs are mainly used to describe semantic properties, we are particularly interested in a unique solution (when exists) for a given goal. This is not surprising in domains such as static semantics, dynamic semantics, and translations. In all these cases, the expected behavior is intrinsically deterministic.

Definition 16 *We say that a Typol program is* **non ambiguous** *if any goal has at most one proof.*

The non ambiguity of a Typol program can be usually determined in a static manner, just looking at the rules; for instance, a sufficient condition is the following: all rules have distinct subjects or two rules with the same subject have exclusive premises. See for instance the two Typol rules describing the behavior of the *while* statement in Example 3: the first premise is either evaluated to *true* or *false*.

3.3 The Strongly Non Circularity property

In this section, we are concerned with the dependencies between attributes in a given proof tree t. Intuitively, these dependencies come from the combination of the local dependency graph D_r within the proof tree. Such a combination is based on the structure of the proof tree and defines a **compound dependency graph** $D(t)$.
The nodes of this graph are in the set ATTR(t) of all attribute occurrences of the proof tree t, and the arrows are those given by the D_r relations for all rules r used to build the proof tree t.
We define the circularity property as follows:

Definition 17 *A Typol program is non circular if, for any proof tree t, the compound dependency graph $D(t)$ is acyclic.*

The non-circularity property of attribute dependencies in a proof tree provides an evaluation order (at least if the program is link-less) for the computation of all attributes.

We also need in the following some properties independent from any proof tree but verified for all potential proof trees because the order of computation of two attributes may depend on the proof tree itself.

We want a dependency relation which includes all the local dependency relations D_r and also all the dependencies induced by any possible proof tree. More precisely, for each sort τ, we define

a relation R_τ on $\mathrm{inh}(\tau) \times \mathrm{syn}(\tau)$ such that $(a, b) \in R_\tau$ if it may be necessary to know the value of a to compute the value of b in a particular proof tree. This dependency may be direct (by any D_r) or indirect (via the transitive closure of D_r).
We call **closed** such a family (R_τ) of relations. More formally:

Definition 18 *Given a Typol program, a family* (R_τ) *of relations is said to be* **closed** *if, for any rule* r*, we have:*

$$[D_r \cup R_{\tau_1} \cup \cdots \cup R_{\tau_n}]^+ \mid_{attr(\tau)} \subset R_\tau \qquad (I)$$

where $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$ *is the signature of the rule* r.
As usual, the notation $[\ ]^+$ *means the transitive closure and* $\mid_{attr(\tau)}$ *means the restriction to the set* $attr(\tau)$.

Given (R_τ) a family of relations and r a rule of signature $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$, we associate a **induced dependency graph** $D_{r,R}$ on the set of attribute positions of the *premises* of r, with the relation:

$$D_r \cup R_{\tau_1} \cup \cdots \cup R_{\tau_n} \mid_{\{\mathrm{attr}(\tau_1), \cdots, \mathrm{attr}(\tau_n)\}} \qquad (D_{r,R})$$

We can define a static notion of non circularity:

Definition 19 *Given a Typol program, a family* (R_τ) *of relations is said to be* **non circular** *if, for each Typol rule* r*, the graph* $D_{r,R}$ *is acyclic.*

Now, we are interested in the existence of a smallest element among the closed family of relations (R_τ). Such a family would contain only useful dependencies between attribute positions and thus avoid useless ones. For that purpose, we interprete the closure condition as a fix-point equation:

$$[D_r \cup R_{\tau_1} \cup \cdots \cup R_{\tau_n}]^+ \mid_{\mathrm{attr}(\tau)} = R_\tau \qquad (II)$$

In fact, we define the application Φ from the set of family relations (R_τ) into itself by:

$$\Phi(R_\tau) \mid_\tau \ = \ [D_r \cup R_{\tau_1} \cup \cdots \cup R_{\tau_n}]^+ \mid_{\mathrm{attr}(\tau)}$$

For any rule r, equation (II) now means: $\Phi(R) = R$.
The set of sorts and the set of rules of a given Typol program are finite and the function Φ is increasing because of the union operator. Therefore, we can apply the fix-point theorem to get the minimal solution (for the inclusion) of equation $\Phi(R) = R$. We can replace the condition (I) by the equation (II) because it is easy to verify that the minimal solution of $\Phi(R) \subset R$ is also the minimal solution of $\Phi(R) = R$.
Moreover, we know that this minimal solution Γ is the limit of sequence $\Gamma^{(n)}$ defined by:

$$\Gamma^{(0)} = \emptyset \ \text{ and } \ \Gamma^{(n+1)} = \Phi(\Gamma^{(n)})$$

Here, the sequence is convergent in a finite number of steps because the manipulated sets are finite. So, we have the following algorithm to compute Γ, starting from $\Gamma^{(1)}$.

We construct a sequence $\Gamma^{(n)}$ of subsets included in the following set:

$$\{(\tau, h, s) / \tau \in \mathcal{P}, h \in \mathrm{inh}(\tau), s \in \mathrm{syn}(\tau)\}$$

Building the minimal relation Γ
$i = 1$
$\Gamma^{(1)} = \{(\tau, h, s)$ / there exists a rule r of type τ and attributes h, s verifying $h \xrightarrow[D_r]{} s$ $\}$

repeat
$\Gamma^{(i+1)} = \{(\tau, h, s)/$there exists a rule r of type τ and attributes h, s verifying $h \xrightarrow[D_{r,\Gamma^i}]{+} s$ $\}$
until $\Gamma^{(i+1)} = \Gamma^{(i)}$ $\diamond$

From the minimal solution Γ, we define the family of relations indexed by the sorts

$$\Gamma_\tau = \{(h, s)/(\tau, h, s) \in \Gamma\}$$

In analogy with Attribute Grammars, we introduce the following definitions:

Definition 20 *A synthesized attribute $s \in syn(\tau)$ is said* **purely synthesized** *if there exists no h in $inh(\tau)$ such that $\Gamma_\tau(h, s)$.*

We naturally extend the definition of purely synthesized attributes to Typol programs:

Definition 21 *A Typol program P is said to be* **purely synthesized** *if, each synthesized attribute of the program P is purely synthesized.*

By analogy with a notion introduced in the Attribute Grammars field by [17,6], we define a larger class of Typol programs as follows:

Definition 22 *The minimal relation Γ of a Typol program is called the* **minimal argument selector**; *when this relation is non circular, we say that the Typol program is* **Strongly Non Circular** *(SNC in short).*

Remark 3 *If it exists a non circular closed relation R for a Typol program, the minimal relation Γ is obviously non circular.*

Example 9 *Here are the induced dependency graphs $D_{r,\Gamma}$ associated with the* let *and the* letrec *operators of the Mini-ML type-checker. The first one is acyclic but the second one contains a cycle (see Figure 3):*

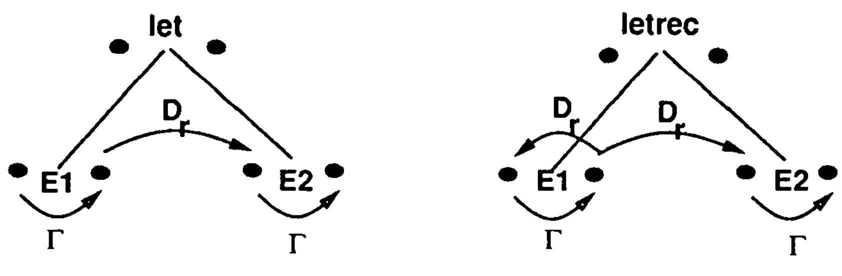

Figure 3: Non circular and circular graphs $D_{r,\Gamma}$

3.4 Reduction of a Typol program

In this section, we give a way to transform a Typol program, possibly circular, into separate Typol programs, all of them Strongly Non Circular. Speaking about semantics, this transformation takes as input a monolithic Typol program and produces as many Typol programs as separate phases in the semantic description.

Our transformation is based on an appropriate splitting of the sets H and S of inherited and synthesized attributes. We compute first the combination of the graphs $D_{r,\Gamma}$ for all rules of a given Typol program P. Such a combination defines **global dependency graph** $D_{P,\Gamma}$:

- the nodes of this graph are the set ATTR(P) of all attribute names of a Typol program P,
- the arrows are those given by the relations $D_{r,\Gamma}$ for all rules r of the program P.

Then, we define a partition of $D_{P,\Gamma}$ as follows:
Two attributes a and b belong to a same component if there is a path from a to b in $D_{P,\Gamma}$ and a path from b to a in $D_{P,\Gamma}$ (or more formally, $a \xrightarrow[D_{P,\Gamma}]{*} b$ and $b \xrightarrow[D_{P,\Gamma}]{*} a$).

This defines the **strongly connected components** of the $D_{P,\Gamma}$ graph.
This partition induces an evaluation order on attributes as follows:

- each component defines a pass of the computation in which all attributes of the component have to be computed;
- a component A must be computed before another component B if there exists an attribute a in A and an attribute b in B such that $a \xrightarrow[D_{P,\Gamma}]{+} b$

We define the subclass of reduced Typol programs as follows:

Definition 23 *A Typol program P is said to be reduced if, for each rule of P, all the sequents are reduced.*

The reduction technique makes it possible to transform a Typol program into separate Typol programs.This is particularly useful to transform a circular Typol program into separate Strongly Non Circular Typol programs. We call such Typol programs **pseudo-circular**. Hopefully, most of the time, the resulting Typol programs are reduced, as all our previous examples.

Example 10 *Reduction of the TYPOL program given in Appendix D*
The minimal argument selector Γ is given by the relation $s \xrightarrow[D_r]{} \rho$ induced by axiom (2). Therefore, the graph $D_{r,\Gamma}$ for rule (6) is cyclic (see Figure 4). So, the program is not Strongly Non Circular.

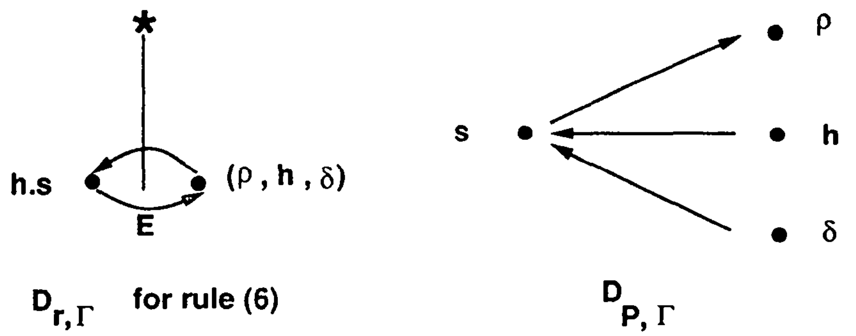

Figure 4

The reduction of this program produces the $D_{P,\Gamma}$ graph given in Figure 4. The strongly connected components are reduced to each node. So, a possible order for the distinct passes could be: compute the δ attribute (which is purely synthesized), and the h attribute (also purely synthesized); then compute the ρ attribute, using the inherited s attribute. The three resulting Typol programs are given in Appendix E.

On the contrary, one can tuple in a single pass two (or more) separate passes provided the partial order is preserved and the SNC property is also preserved. For instance, the attributes δ and h can be computed in a single pass.

4 Functional evaluation of Strongly Non Circular Typol programs

We are looking for Typol programs for which a pattern-matching mechanism in enough to prove a goal. In this case, provided the absence of unification process, we can then associate an equivalent functional evaluator. One method to avoid unification is to be sure that for every couple of terms to be unified, one is always a closed term. In this section, *closed* refers to the terms in the attribute domains.

We prove in this section that Strongly Non Circular reduced non ambiguous Typol programs can be computed without unification by a single mechanism of pattern-matching. We study two separate cases:

- purely synthesized Typol non ambiguous programs,
- SNC reduced Typol non ambiguous programs.

4.1 Unification-free execution of purely synthesized Typol programs

We begin with the case of purely synthesized Typol programs because of its own interest and also as a preliminary for the proof of the next case.

Theorem 1 *Given a purely synthesized non ambiguous Typol program P, with only safe rules, given a goal $\vdash T_0 : S_0$ then the solution S_0 is bound to a closed term and can be computed using only the pattern-matching mechanism.*

proof (by induction on the height of the proof tree of the goal):

- when the height of the proof tree is 1, the goal is directly proved by an axiom

$$\overline{\vdash T : S}$$

By definition of a proof tree, it must exist a substitution σ such that

$$\sigma T = \sigma T_0 \qquad\qquad \sigma S = \sigma S_0$$

Since T_0 is a closed term, we have $\sigma T = T_0$.
Every rule in P is safe so VAR(S) $\subset$ VAR(T) and as σ bounds each variable of T to a closed term, σS is also a closed term and therefore σS_0 too. The solution is then the closed term $\sigma' \sigma S$ where σ' is the matching substitution bounding the variables of S_0 to the closed term σS.

- when the height of the proof tree is > 1, we call r

$$\frac{\vdash T_1 : S_1 \qquad \cdots \qquad \vdash T_n : S_n}{\vdash T : S}$$

the top rule of the proof tree. By definition, it exists a substitution σ such that

$$\sigma T = \sigma T_0 \qquad\qquad \sigma S = \sigma S_0$$

And as T_0 is a closed term, we have the matching condition $\sigma T = T_0$.
The current subgoals to prove are

$$\vdash \sigma T_1 : \sigma S_1 \qquad \cdots \qquad \vdash \sigma T_n : \sigma S_n$$

Let us start with the first subgoal $\vdash \sigma T_1 : \sigma S_1$. Using the safety hypothesis for rule (r), we have $\mathrm{VAR}(T_1) \subset \mathrm{VAR}(T)$. So σT_1 is a closed term. By induction hypothesis, the solution s_1 is a closed term and is the result of the pattern-matching σ_1 which transforms the pattern σS_1 into s_1.
The next subgoal to prove is $\vdash \sigma T_2 : \sigma_1 \sigma S_2$. Similarly, we have $\mathrm{VAR}(T_2) \subset \mathrm{VAR}(T)$. So σT_2 is a closed term. By induction hypothesis, the solution s_2 is a closed term and is given by the pattern-matching σ_2.
And so on until the proof of the last subgoal $\vdash \sigma T_n : \sigma_{n-1} \cdots \sigma_1 \sigma S_n$.
So we have $\sigma S_0 = \sigma_n \cdots \sigma_1 \sigma S$. We have seen that $\sigma_n \cdots \sigma_1 \sigma$ bounds each variable of $T, S_1, \cdots, S_n$ to a closed term; on the other hand, $\mathrm{VAR}(S) \subset \mathrm{VAR}(T) \cup \mathrm{VAR}(S_1) \cdots \cup \mathrm{VAR}(S_n)$, so we deduce $\sigma_n \cdots \sigma_1 \sigma S$ is also a closed term and therefore σS_0 too. The solution is then the result of the pattern-matching σ' where σ' is the substitution bounding the variables of S_0 to the closed term $\sigma_n \cdots \sigma_1 \sigma S$.$\square$

Remark 4 *During the proof process, the attribute expressions $S_1, \cdots, S_n$ can include both Typol variables and attribute domains variables, so we consider this last kind of variables as a new sort of typed variables.*
On the other hand, if the synthesized attribute expressions $S_1, \cdots, S_n$ do not have any common variable, each substitution σ_k is given by the simpler matching condition $\sigma_k S_k = s_k$. If we assume this kind of absence of interaction, it will be easier to generate an equivalent functional evaluator (see Section 5).

4.2 Unification-free execution of Strongly Non Circular reduced Typol programs

A purely synthesized Typol program is always Strongly Non Circular. We study in this section the general form of Strongly Non Circular programs. We consider only reduced programs. In fact, most of Typol programs are naturally given in a reduced form; if it is not the case, the decomposition in passes described in section 3.4 often produces (but not always) a sequence of reduced programs.

Theorem 2 *Given a SNC reduced non ambiguous Typol program P, with only safe rules, given a goal $H_0 \vdash T_0 : S_0$ where H_0 is a given closed term then the solution S_0 is bound to a closed term and can be computed using only the pattern-matching mechanism.*

We first give a lemma that shows it is possible to arrange the premises of a Typol rule so that premises are actually computable in this order. More precisely,

Lemma 1 *Let P be a Strongly Non Circular reduced Typol program P.*
For each rule r

$$\frac{H_1 \vdash T_1 : S_1 \qquad \cdots \qquad H_n \vdash T_n : S_n}{H \vdash T : S}$$

there exists an order of the premises of r such that, for any indices $i < j$, the attribute S_i does not depend on the attribute S_j in the $D_{r,\Gamma}$ dependency relation.

proof:

The Strongly Non Circular hypothesis means the graph $D_{r,\Gamma}$ is acyclic. Let us consider an attribute without a predecessor in this graph.

- if this attribute is a synthesized one, we put its associated sequent in first position;
- if this attribute is an inherited one, the associated synthesized attribute in its sequent can only depend on it and therefore is independant from any synthesized attribute, so we can also put this sequent in first position.

Then, we remove the corresponding arrows from this sequent. We use the same process with the remaining premises to chose the sequent in second position, until all the premises are re-ordered.□

Example 11 *Re-ordering the premises*

Here a possible graph $D_{r,\Gamma}$:

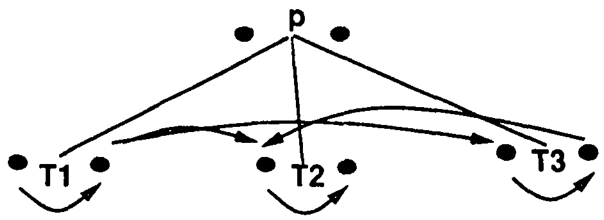

Figure 5

An order given by Lemma 1 are sequents with respective subjects T_1, T_3, T_2.

Now we give the proof of Theorem 2 assuming the rules have been re-ordered as indicated in Lemma 1.

The principle of the proof is the same as in the proof for the purely synthesized case but with addition of some technicalities due to the inherited attributes.

We prove by induction on the height of the proof tree that the solution S_0 is closed and at the same time, we show how we can compute it using only a matching process.

proof of Theorem 2:

- when the height of the proof tree is 1, the goal is directly proved by an axiom

$$\overline{H \vdash T : S}$$

By definition of a proof tree, it exists a substitution σ such that

$$\sigma T = \sigma T_0 \qquad \sigma H = \sigma H_0 \qquad \sigma S = \sigma S_0$$

Since T_0 and H_0 are closed terms, we have $\sigma T = T_0$ and $\sigma H = H_0$.

Every rule in P is safe so $\mathrm{VAR}(S) \subset \mathrm{VAR}(T) \cup \mathrm{VAR}(H)$ and as σ bounds each variable of T and H to a closed term, σS is also a closed term and therefore σS_0 too. The solution is then the closed term $\sigma' \sigma S$ where σ' is the substitution bounding the variables of S_0 to the closed term σS.

- when the height of the proof tree is > 1, we call r

$$\frac{H_1 \vdash T_1 : S_1 \qquad \cdots \qquad H_n \vdash T_n : S_n}{H \vdash T : S}$$

the top rule of the proof tree. By definition, it exists a substitution σ such that

$$\sigma T = \sigma T_0 \qquad \sigma H = \sigma H_0 \qquad \sigma S = \sigma S_0$$

And as T_0 and H_0 are closed terms, we have the matching conditions $\sigma T = T_0$ and $\sigma H = H_0$ which define σ on VAR(T) $\cup$ VAR(H).
The current subgoals to prove are

$$\sigma H_1 \vdash \sigma T_1 : \sigma S_1 \qquad \cdots \qquad \sigma H_n \vdash \sigma T_n : \sigma S_n$$

Let us start with the first subgoal $\sigma H_1 \vdash \sigma T_1 : \sigma S_1$. Using the safety hypothesis for rule (r), we have

$$\mathrm{VAR}(H_1) \subset \mathrm{VAR}(T) \cup \mathrm{VAR}(H) \cup \mathrm{VAR}(S_1) \cdots \cup \mathrm{VAR}(S_k) \qquad (i)$$

Two cases can occur in the dependency graph $D_{r,\Gamma}$:

1. there is no arrow from H_1 to S_1; in that case, the attribute S_1 is purely synthesized and Theorem 1 proves that the solution s_1 is a closed term and is the result of the pattern-matching σ_1 which transforms the pattern σS_1 into s_1.
2. there is an arrow from H_1 to S_1. From the re-ordering lemma, we know that in fact inclusion (i) can be replaced by a more restricted inclusion (ii):

$$\mathrm{VAR}(H_1) \subset \mathrm{VAR}(T) \cup \mathrm{VAR}(H) \cup \mathrm{VAR}(S_1) \qquad (ii)$$

   Moreover, since the program is Strongly Non Circular, S_1 and H_1 do not share a common variable, so (ii) is reduced to

$$\mathrm{VAR}(H_1) \subset \mathrm{VAR}(T) \cup \mathrm{VAR}(H)$$

As σ bounds each variable of T and H to a closed term, σH_1 is also a closed term. On the other hand, σT_1 is also a closed term (because the rule is safe). By induction hypothesis, the solution s_1 is a closed term and is the result of the pattern-matching σ_1 which transforms the pattern σS_1 into s_1.

The next subgoal to prove is $\sigma_1 \sigma H_2 \vdash \sigma T_2 : \sigma_1 \sigma S_2$.
With similar arguments, we prove that the solution s_2 is a closed term and is given by the pattern-matching σ_2.
And so on until the proof of the last subgoal $\sigma_{n-1} \cdots \sigma_1 \sigma H_n \vdash \sigma T_n : \sigma_{n-1} \cdots \sigma_1 \sigma S_n$.
So, the value of S_0 must be given by $s_0 = \sigma_n \cdots \sigma_1 \sigma S$. We have seen that $\sigma_n \cdots \sigma_1 \sigma$ bounds each variable of $T, S_1, H, \cdots, S_n$ to a closed term: on the other hand, VAR(S) $\subset$ VAR(T) $\cup$ VAR(S_1) $\cdots \cup$ VAR(S_n), so we deduce that s_0 is also a closed term. The solution is then the result of the pattern-matching σ' where σ' is the substitution which match S_0 with the closed term s_0.□

5 Lisp implementation

From the *method* used in the proofs of Theorems 1 and 2, we deduce how to evaluate Typol programs using only a pattern-matching mechanism. In order to automatically translate a Typol program into a functional program, we define a new language construct, called **CondMatch**. Intuitively, is a conditional expression similar to the usual Cond construct except that the selection of a branch is based on a sequence of pattern-matching. More precisely, we give the syntax and the semantics of the CondMatch construct.

Syntax of the CondMatch construct:

```
(CondMatch
    L1
    ...
    Lk
)
```

where each branch Lj is a list with the following structure:

```
(  ((p1 e1)
      ...
    (pn en))
  s1 ... sN
)
```

where $p1, \cdots, pn$ are terms and $e1, \cdots, en$, $s1, \cdots, sN$ are S-expressions.

Semantics of the CondMatch construct:
The semantics is a blend of the Lisp *Cond* construct and the ML *match*: CondMatch selects the first branch in which the whole sequence of pattern-matchings between each non evaluated pattern *pi* and expression *ei* succeeds. These matchings are done in sequence, using the local environments given by the previous matchings.
Then, CondMatch evaluates in sequence the expressions $s1, \cdots, sN$ in the resulting matching environment and returns the value of the last evaluation sN.
If one matching fails, the next branch is examined to try the pattern-matchings.

With the CondMatch construct, it is easy to give an executable version of Typol programs in a functional style: to a Typol program, we associate a Lisp function called **EvalTypol** which is merely a CondMatch expression with each branch representing a Typol rule. Note the order of the rules is preserved in this translation. The parameters of the **EvalTypol** function are either a single abstract syntax term T_0 if the Typol program is purely synthesized or an abstract syntax term T_0 together with a given closed term H_0 (the subject and the inherited attribute of the goal to be proved). The result of **EvalTypol** is the solution S_0 for the respective goal $\vdash T_0 : S_0$ and $H_0 \vdash T_0 : S_0$.
The generic EvalTypol function is as follows:

```
(defun EvalTypol (T0 H0)
    (condMatch
                ...
                ( ( ((T H) (T0 H0))
                    (S1 (EvalTypol T1 H1))
                     ...
                    (Sn (EvalTypol Tn Hn))
                  )
                 S
                )
                ...
```

```
    )
)
```

where each branch of the CondMatch construct is associated with a Typol rule of the form:

$$\frac{H_1 \vdash T_1 : S_1 \qquad \cdots \qquad H_n \vdash T_n : S_n}{H \vdash T : S}$$

To give a simple and concrete example, let us consider the typol specification of a transformation about propositional calculus formulas. We denote the usual logical operators (and, or, implies, not) by $\wedge$, $\vee$, $\rightarrow$, and $\neg$. We expect from this transformation to push inside the formula as much as possible the negation operator. For instance, the formula $\neg(a \vee (\neg b \wedge c))$ is transformed into $\neg a \wedge (b \vee \neg c)$.
The transformation is expressed by the following Typol rules:

Example 12 *From a purely synthesized Typol program to a Lisp function*

$$\frac{}{\vdash id\ X : id\ X}$$

$$\frac{}{\vdash \neg id\ X : \neg id\ X}$$

$$\frac{\vdash X : A \qquad \vdash Y : B}{\vdash X \wedge Y : A \wedge B}$$

$$\frac{\vdash X : A \qquad \vdash Y : B}{\vdash X \vee Y : A \vee B}$$

$$\frac{\vdash X : A \qquad \vdash Y : B}{\vdash X \rightarrow Y : A \rightarrow B}$$

$$\frac{\vdash \neg A : A_1 \qquad \vdash \neg B : B_1}{\vdash \neg(A \vee B) : A_1 \wedge B_1}$$

$$\frac{\vdash \neg A : A_1 \qquad \vdash \neg B : B_1}{\vdash \neg(A \wedge B) : A_1 \vee B_1}$$

$$\frac{\vdash A : A_1 \qquad \vdash \neg B : B_1}{\vdash \neg(A \rightarrow B) : A_1 \rightarrow B_1}$$

$$\frac{\vdash A : A_1}{\vdash \neg\neg A : A_1}$$

The associated Lisp evaluator implementing the proof method of our theorems is:

```
(defun EvalTypol (TO)
     (CondMatch
                          (  ( ((id x) TO))
                             (id x))
                          (  ( ((not (id x)) TO))
                             (not (id x)))
                          (  ( ((and e1 e2) TO)
                               (a1 (EvalTypol e1))
                               (a2 (EvalTypol e2)))
```

```
                  (list 'and a1 a2))
            (  ( ((or e1 e2) TO)
                  (a1 (EvalTypol e1))
                  (a2 (EvalTypol e2)))
                (list 'or a1 a2))
            (  ( ((implies e1 e2) TO)
                  (a1 (EvalTypol e1))
                  (a2 (EvalTypol e2)))
                (list 'implies a1 a2))
            (  ( ((not (or e1 e2)) TO)
                  (a1  (EvalTypol (list 'not  e1)))
                  (a2  (EvalTypol (list 'not  e2))))
                (list 'and a1 a2))
            (  ( (not (and e1 e2)) TO)
                  (a1  (EvalTypol (list 'not  e1)))
                  (a2  (EvalTypol (list 'not  e2))))
                (list 'or a1 a2)
            (  ( ((not (implies e1 e2)) TO)
                  (a1  (EvalTypol e1))
                  (a2  (EvalTypol (list 'not  e2))))
                (list 'and a1 a2))
            (  ( ((not (not e1)) TO)
                  (a1  (EvalTypol e1)))
                a1)
    )
)
```

This Typol program can not be easily described in the Attribute Grammars formalism because, in some rules, the subjects of the premises are not subterms of the subject of the conclusion; moreover, in these rules, the subject of the rule is built using two abstract syntax operators.

Now, we give another example in which a Prolog evaluator requires unification. To be short, we use an artificial example with only one rule and one axiom. This example can be used to change the left associativity with the right associativity in expressions. In the abstract syntax, there is only one unary operator "p" and one atomic operator "id":

Example 13 *A further example*

a rule

$$\frac{\vdash X : ((U \rightarrow V) \rightarrow W)}{\vdash p(X) : (U \rightarrow (V \rightarrow W))}$$

and an axiom

$$\frac{}{\vdash id\ X : (((1 \rightarrow 2) \rightarrow 3) \rightarrow 4)}$$

The associated Lisp function is:

```
(defun EvalTypol (TO)
     (CondMatch
          ((   ((p(X)) TO)
               (((U -> V) -> W) (EvalTypol X)))
             (U -> (V -> W)))
```

```
        ((    ((id X) T0))
           ((((1 -> 2) -> 3) -> 4))
       )
)
```

Proving the goal

$$\vdash p(p(id\ A)) : S_0$$

gives, without unification, the result

$$S_0 = (1 \rightarrow (2 \rightarrow (3 \rightarrow 4)))$$

Let us consider now a Typol program with links between input attributes: for instance, the rule for the application in the Mini-ML type-checker (see Appendix B):

$$\frac{\rho \vdash \mathrm{E}_1 : \tau' \rightarrow \tau \qquad \rho \vdash \mathrm{E}_2 : \tau'}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 : \tau}$$

This rule does not verify the link-less condition.
As indicated in Remark 4, it is possible to associate a functional evaluator when the given goals are acceptable, but we have to replace the CondMatch construct by a more sophisticated construct, taking in account the identity of the two occurrences of the variable τ' in the patterns $\tau' \rightarrow \tau$ and τ'. The required modification consists in the sequential updating of the matching patterns used in a branch.

Finally, it is interesting to note that the Strongly Non Circularity hypothesis is mandatory for getting a closed solution. Consider for instance the following example, where the abstract syntax is composed with only one binary operator o and one atomic operator "id".

Example 14 *a non SNC Typol program*

$$\frac{(\rho, \mu) \vdash \mathrm{E}_1 : \lambda \qquad (\rho, \lambda) \vdash \mathrm{E}_2 : \mu}{\rho \vdash \mathrm{E}_1 \circ \mathrm{E}_2 : (\lambda, \mu)}$$

$$\overline{(\rho, \lambda) \vdash \mathrm{id}\ \mathrm{I} : \lambda}$$

The associated graph $D_{r,\Gamma}$ is cyclic (see Figure 6), so the program is not Strongly Non Circular.

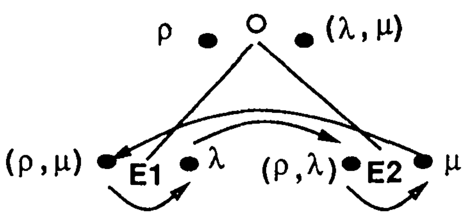

Figure 6

Proving the goal

$$\vdash id\ A \circ id\ B : S_0$$

gives the non closed result

$$S_0 = (X, X)$$

where X is a variable.

Nevertheless, we can see that the pattern-matching mechanism is still sufficient to evaluate this non SNC program, provided we use a lazy functional language. We will study this larger class of programs and their evaluation in a future paper.

6 Concluding remarks

We have studied the structure of Typol programs with the intention to provide more specific evaluators. If we compare with the Attribute Grammars framework, we can expect the existence of many other interesting classes of Typol programs.

We focused here on the Strongly Non Circular property because it makes it possible to statically translate a Typol program into a Lisp program (see [7,16] for related works in the AG framework). The absence of unification in our evaluators is a first step to a couple of applications:

- an incremental evaluation of Typol programs;
- the introduction of higher-order terms in Typol (as in Lambda-Prolog [14]) because for second order terms, only the matching is decidable and has a finite number of solutions, as opposed to the unification.

Acknowledgements: we are glad to thank Paul Franchi-Zannettacci for useful discussions during the writing of this paper.

References

[1] Alexandre F. "Une transformation de programmes logiques en systèmes de réécriture" Proc. of Journées GRECO-GROPLAN, Nice Jan. 1990, published in Bigre.

[2] Attali I. "Compilation de programmes Typol par Attributs Sémantiques" Doctoral thesis, University of Nice, April 1989.

[3] Attali I. & Franchi-Zannettacci P. "Unification-free Execution of Typol Programs by Semantic Attribute Evaluation", Proceedings Fifth International Conference Symposium on Logic Programming, Seattle, August 1988, MIT Press.

[4] Berry G. & Sethi R. "From regular expressions to deterministic automata" TCS 48, 1, 1986

[5] Borras P., Clément D., Despeyroux T., Incerpi J., Kahn G., Lang B., & Pascual V. "CENTAUR: the system" SIGSOFT'88, Third Annual Symposium on Software Development Environments, Boston, Nov 88.

[6] Clément D., Despeyroux J., Despeyroux T. & Kahn G. "A simple applicative language: Mini-ML" Symp. on Functional Programming Languages and Computer Architecture, 1986

[7] Courcelle B. & Franchi-Zannettacci P. "Attribute Grammars and recursive program schemes" (I and II) TCS 17 2 pp 163-191 and TCS 17 3 pp 235-257, 1982

[8] Damas L. & Milner R. "Principal type-schemes for functional programs" Proceedings of the ACM Conference on Principles of Programming Languages, 1982, pp 207-212

[9] Deransart P. & Maluszynski J. "Relating Logic Programs and Attribute Grammars" J. Logic Programming vol 2 n 2 pp 119-155, 1985

[10] Deransart P., Jourdan M., & Lorho B. "Attribute Grammars: Definitions, Systems and Bibliography" LNCS 323, Spinger Verlag, 1988

[11] Despeyroux T. "Executable Specification of static semantics" Semantics of Data Types, LNCS 173, 1984

[12] Despeyroux T. "Typol: a formalism to implement Natural Semantics" INRIA research report 94, 1988

[13] Gentzen G. "Investigation into Logical Deduction" Thesis 1935, reprinted in "The collected papers of Gerhard Gentzen" E. Szabo, North-Holland, Amsterdam, 1969

[14] Hannan J. & Miller D. "A meta-logic for functional programming" in Abramson H. and Rogers M. editors, Meta-programming in Logic Programming, Chapter 24, pp 453-476, MIT Press, 1989

[15] Isakowitz T. "Relating Logic Programs and Attribute Grammars" Research Report, Univ. Pennsylvania, 1988

[16] Jourdan M. "Strongly Non-Circular Attribute Grammars and their recursive evaluation" ACM Sigplan Symp. on Compiler Construction, Montreal Sigplan Notices 19, 6, 1984

[17] Kahn G. "Natural Semantics" Proc. of Symp on Theoretical Aspects of Computer Science, Passau, Germany, LNCS 247, 1987

[18] Kennedy K. & Warren S. K. "Automatic generation of efficient evaluators for Attribute Grammars" Proc. of the 3^{rd} ACM Conf on Principle of Programming Languages, Atlanta, 1976

[19] Knuth D. E. "Semantics of Context-Free Languages" Math. Syst. Theory 2, 1968

[20] Komorowski H. J. & Maluszynski J. "Unification-free execution of logic programs" 1985 IEEE Symp. on Logic Programming, Boston, 1985, pp 78-86

[21] Plotkin G. D. "A structural approach to operational semantics" Report DAIMI FN-19, Computer Science Dpt, Aarhus Univ., Aarhus, Denmark, 1981

[22] Tavernini V. E. "Translating Natural Semantic Specifications to Attribute Grammars" Master Thesis, University of Illinois, Urbana-Champaign, 1987

[23] Warren D. H. D. "Logic Programming and Compiler Writing" Software Practice and Experience, 10, 1980, pp 97-125

Appendix A: Some rules for the dynamic semantics of a small Pascal-like language

$$\frac{\emptyset_s \overset{\text{allocate}}{\vdash} \text{DECLS} : s_1 \qquad s_1 \vdash \text{STMS} : s_2}{\vdash \text{begin DECLS STMS end} : s_2} \tag{1}$$

$$s \vdash \text{stms[ ]} : s \tag{2}$$

$$\frac{s \vdash \text{STM} : s_1 \qquad s_1 \vdash \text{STMS} : s_2}{s \vdash \text{STM; STMS} : s_2} \tag{3}$$

$$\frac{s \vdash \text{ID} : x \qquad s \vdash \text{EXP} : v \qquad s, v \overset{\text{update}}{\vdash} x : s_1}{s \vdash \text{ID} := \text{EXP} : s_1,} \tag{4}$$

$$\frac{s \vdash \text{EXP} : \mathit{true} \qquad s \vdash \text{STMS}_1 : s_1}{s \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \text{ fi} : s_1} \tag{5}$$

$$\frac{s \vdash \text{EXP} : \mathit{false} \qquad s \vdash \text{STMS}_2 : s_1}{s \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \text{ fi} : s_1} \tag{6}$$

$$\frac{s \vdash \text{EXP} : \mathit{true} \qquad s \vdash \text{STMS} : s_1 \qquad s_1 \vdash \text{while EXP do STMS end} : s_2}{s \vdash \text{while EXP do STMS end} : s_2} \tag{7}$$

$$\frac{s \vdash \text{EXP} : \mathit{false}}{s \vdash \text{while EXP do STMS end} : s} \tag{8}$$

$$\frac{s \overset{\text{get}}{\vdash} \text{X} \mapsto v}{s \vdash \text{id X} : v} \tag{9}$$

$$s \vdash \text{boolean X} : \text{X} \tag{10}$$

$$s \vdash \text{number X} : \text{X} \tag{11}$$

$$\frac{s \vdash \text{EXP}_1 : x_1 \qquad s \vdash \text{EXP}_2 : x_2 \qquad \overset{\text{eval}}{\vdash} \text{OP}, x_1, x_2 : x}{s \vdash \text{EXP}_1 \text{ OP EXP}_2 : x} \tag{12}$$

Appendix B: The main set about static semantics of Mini-ML without patterns

$$\rho \vdash \text{number } \mathrm{N} : int \tag{1}$$

$$\rho \vdash \text{true} : bool \tag{2}$$

$$\rho \vdash \text{false} : bool \tag{3}$$

$$\frac{\rho \cdot [\mathrm{X} : \tau'] \vdash \mathrm{E} : \tau}{\rho \vdash \lambda \mathrm{X}.\mathrm{E} : \tau' \to \tau} \tag{4}$$

$$\frac{\rho \overset{\text{type\_of}}{\vdash} \text{ident } \mathrm{X} : \sigma \qquad \tau = inst(\sigma)}{\rho \vdash \text{ident } \mathrm{X} : \tau} \tag{5}$$

$$\frac{\rho \vdash \mathrm{E}_1 : bool \qquad \rho \vdash \mathrm{E}_2 : \tau \qquad \rho \vdash \mathrm{E}_3 : \tau}{\rho \vdash \text{if } \mathrm{E}_1 \text{ then } \mathrm{E}_2 \text{ else } \mathrm{E}_3 : \tau} \tag{6}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \tau_1 \qquad \rho \vdash \mathrm{E}_2 : \tau_2}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) : \tau_1 \times \tau_2} \tag{7}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \tau' \to \tau \qquad \rho \vdash \mathrm{E}_2 : \tau'}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 : \tau} \tag{8}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \tau_1 \qquad \sigma_1 = gen(\tau_1, \rho) \qquad \rho \cdot [\mathrm{X} : \sigma_1] \vdash \mathrm{E}_2 : \tau_2}{\rho \vdash \text{let } \mathrm{X} = \mathrm{E}_1 \text{ in } \mathrm{E}_2 : \tau_2} \tag{9}$$

$$\frac{\rho \cdot [\mathrm{X} : \tau_1] \vdash \mathrm{E}_1 : \tau_1 \qquad \rho \cdot [\mathrm{X} : \tau_1] \vdash \mathrm{E}_2 : \tau_2}{\rho \vdash \text{letrec } \mathrm{X} = \mathrm{E}_1 \text{ in } \mathrm{E}_2 : \tau_2} \tag{10}$$

Appendix C: The main set about dynamic semantics of Mini-ML

$$\rho \vdash \text{number}\,\mathrm{N} : \mathrm{N} \tag{1}$$

$$\rho \vdash \text{true} : \mathit{true} \tag{2}$$

$$\rho \vdash \text{false} : \mathit{false} \tag{3}$$

$$\rho \vdash \lambda \mathrm{X}.\mathrm{E} : [\![\text{let}\,\mathrm{X}\,\text{in}\,\mathrm{E}, \rho]\!]_{ml} \tag{4}$$

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \text{ident}\,\mathrm{I} \mapsto \alpha}{\rho \vdash \text{ident}\,\mathrm{I} : \alpha} \tag{5}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \mathit{true} \qquad \rho \vdash \mathrm{E}_2 : \alpha}{\rho \vdash \text{if}\,\mathrm{E}_1\,\text{then}\,\mathrm{E}_2\,\text{else}\,\mathrm{E}_3 : \alpha} \tag{6}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \mathit{false} \qquad \rho \vdash \mathrm{E}_3 : \alpha}{\rho \vdash \text{if}\,\mathrm{E}_1\,\text{then}\,\mathrm{E}_2\,\text{else}\,\mathrm{E}_3 : \alpha} \tag{7}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \alpha \qquad \rho \vdash \mathrm{E}_2 : \beta}{\rho \vdash (\mathrm{E}_1, \mathrm{E}_2) : (\alpha, \beta)} \tag{8}$$

$$\frac{\rho \vdash \mathrm{E}_1 : f \qquad \rho \vdash \mathrm{E}_2 : \alpha \qquad \overset{\text{apply}}{\vdash} f, \alpha : \beta}{\rho \vdash \mathrm{E}_1\ \mathrm{E}_2 : \beta} \tag{9}$$

$$\frac{\rho \vdash \mathrm{E}_1 : \alpha \qquad (\rho, \mathrm{X} \mapsto \alpha) \vdash \mathrm{E}_2 : \beta}{\rho \vdash \text{let}\,\mathrm{X} = \mathrm{E}_1\,\text{in}\,\mathrm{E}_2 : \beta} \tag{10}$$

$$\frac{(\rho, \mathrm{P} \mapsto \alpha) \vdash \mathrm{E}_1 : \alpha \qquad (\rho, \mathrm{X} \mapsto \alpha) \vdash \mathrm{E}_2 : \beta}{\rho \vdash \text{letrec}\,\mathrm{X} = \mathrm{E}_1\,\text{in}\,\mathrm{E}_2 : \beta} \tag{11}$$

Appendix D: the regular expressions translation

$$\frac{end \vdash \text{E} : t_1, t_2, t_3}{\vdash \text{axiom}(\text{E}) : (t_1, t_2, t_3)} \tag{1}$$

$$\frac{}{\text{S} \vdash \text{I} : (\text{I} \longmapsto \text{S}, \text{I}, \mathit{false})} \tag{2}$$

$$\frac{}{\text{S} \vdash \epsilon : (\emptyset, \emptyset, \mathit{true})} \tag{3}$$

$$\frac{\text{S} \vdash \text{E}_1 : (\rho_1, h_1, \delta_1) \qquad \text{S} \vdash \text{E}_2 : (\rho_2, h_2, \delta_2)}{\text{S} \vdash \text{E}_1 + \text{E}_2 : (\rho_1 \cdot \rho_2, h_1 \cdot h_2, \delta_1 \vee \delta_2)} \tag{4}$$

$$\frac{\text{S} \vdash \text{E}_2 : (\rho_2, h_2, \delta_2) \qquad h_2 \cup \delta_2 \cdot \text{S} \vdash \text{E}_1 : (\rho_1, h_1, \delta_1)}{\text{S} \vdash \text{E}_1 \cdot \text{E}_2 : (\rho_1 \cdot \rho_2, h_1 \cup \delta_1 \cdot h_2, \delta_1 \wedge \delta_2)} \tag{5}$$

$$\frac{h \cdot \text{S} \vdash \text{E} : (\rho, h, \delta)}{\text{S} \vdash \text{E}^* : (\rho, h, \mathit{true})} \tag{6}$$

Appendix E: Decomposition in passes of the regular expressions translation

set δ **is**

$$\frac{}{\vdash \mathrm{I} : \mathit{false}}$$

$$\frac{}{\vdash \epsilon : \mathit{true}}$$

$$\frac{\vdash \mathrm{E}_1 : \delta_1 \qquad \vdash \mathrm{E}_2 : \delta_2}{\vdash \mathrm{E}_1 + \mathrm{E}_2 : \delta_1 \vee \delta_2}$$

$$\frac{\vdash \mathrm{E}_1 : \delta_1 \qquad \vdash \mathrm{E}_2 : \delta_2}{\vdash \mathrm{E}_1 \cdot \mathrm{E}_2 : \delta_1 \wedge \delta_2}$$

$$\frac{\vdash \mathrm{E} : \delta}{\vdash \mathrm{E}^* : \mathit{true}}$$

end δ;
set h is

$$\frac{}{\vdash \mathrm{I} : \mathrm{I}}$$

$$\frac{}{\vdash \epsilon : \emptyset}$$

$$\frac{\vdash \mathrm{E}_1 : h_1 \qquad \vdash \mathrm{E}_2 : h_2}{\vdash \mathrm{E}_1 + \mathrm{E}_2 : h_1 \cdot h_2}$$

$$\frac{\vdash \mathrm{E}_1 : h_1 \qquad \vdash \mathrm{E}_2 : h_2 \qquad \overset{\delta}{\vdash} \mathrm{E}_1 : \delta_1}{\vdash \mathrm{E}_1 \cdot \mathrm{E}_2 : h_1 \cup \delta_1 \cdot h_2}$$

$$\frac{\vdash \mathrm{E} : h}{\vdash \mathrm{E}^* : h}$$

end h;
set s,ρ **is**

$$\frac{}{\mathrm{s} \vdash \mathrm{I} : \mathrm{I} \mapsto \mathrm{s}}$$

$$\frac{}{\mathrm{s} \vdash \epsilon : \emptyset}$$

$$\frac{\mathrm{s} \vdash \mathrm{E}_1 : \rho_1 \qquad \mathrm{s} \vdash \mathrm{E}_2 : \rho_2}{\mathrm{s} \vdash \mathrm{E}_1 + \mathrm{E}_2 : \rho_1 \cdot \rho_2}$$

$$\frac{\overset{h}{\vdash} \mathrm{E}_2 : h_2 \qquad \overset{\delta}{\vdash} \mathrm{E}_2 : \delta_2 \qquad \mathrm{s} \vdash \mathrm{E}_2 : \rho_2 \qquad h_2 \cup \delta_2 \cdot \mathrm{s} \vdash \mathrm{E}_1 : \rho_1}{\mathrm{s} \vdash \mathrm{E}_1 \cdot \mathrm{E}_2 : \rho_1 \cdot \rho_2}$$

$$\frac{\overset{h}{\vdash} \mathrm{E} : h \qquad h \cdot \mathrm{s} \vdash \mathrm{E} : \rho}{\mathrm{s} \vdash \mathrm{E}^* : \rho}$$

end s,ρ;

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

ISSN 0249-6399