Unification-free execution of TYPOL programs by semantic attribute evaluation

Isabelle Attali, Paul Franchi-Zannettacci

INRIA

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° 864

UNIFICATION-FREE EXECUTION OF TYPOL PROGRAMS BY SEMANTIC ATTRIBUTE EVALUATION

Isabelle ATTALI
Paul FRANCHI-ZANNETTACCI

JUILLET 1988

UNIFICATION-FREE EXECUTION OF TYPOL PROGRAMS BY SEMANTIC ATTRIBUTE EVALUATION [1]

EXECUTION DE PROGRAMMES TYPOL SANS UNIFICATION PAR EVALUATION D'ATTRIBUTS SEMANTIQUES

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

Paul FRANCHI-ZANNETTACCI
LISAN – University of Nice
Avenue A. Einstein
06561 Valbonne Cedex, France
e-mail: pfz@cerisi.cerisi.fr

ABSTRACT

This paper deals with relations between Logic Programming and Attribute Grammars. We use techniques designed for Attribute Grammars to eliminate run-time unification in a subclass of logic programs, namely TYPOL specifications. TYPOL is a formalism for expressing semantic specifications inside the CENTAUR Programming Environment. In the current implementation of TYPOL, specifications are compiled into PROLOG code. This code does not take into account the distinctive features of TYPOL.

We transform a static TYPOL definition into an equivalent Attribute Grammar, replacing unification by attribute evaluation. Thus, we propose an optimized implementation of TYPOL based on run-time efficient languages like LISP, C, or ADA. From this experiment, we expect to find sufficient conditions to remove unification from execution of PROLOG programs.

RESUME

Ce papier traite des rapports entre la Programmation Logique et les Grammaires Attribuées. Nous utilisons des techniques conçues pour les Grammaires Attribuées pour éliminer l'unification lors de l'exécution d'une sous-classe de programmes logiques, les spécifications TYPOL. TYPOL est un formalisme de spécifications sémantiques au sein du générateur d'environnements de programmation CENTAUR. L'implémentation actuelle du langage TYPOL produit du code PROLOG à partir de spécifications TYPOL mais ce code ne reflète pas les caractéristiques spécifiques du formalisme.

Nous transformons une spécification statique TYPOL en une Grammaire Attribuée équivalente, remplaçant ainsi le mécanisme d'unification par une évaluation d'attributs sémantiques. Nous proposons une implémentation optimisée du formalisme TYPOL, dans des langages efficaces comme LISP, C, ou ADA. Notre travail nous autorise à espérer des résultats dans le domaine de PROLOG: trouver des conditions suffisantes permettant d'éliminer l'unification lors de l'exécution de programmes PROLOG.

[1] This work is supported by CNRS-GRECO in Programming

1 Introduction

This paper discusses the relationship between TYPOL [11,12], a specialized logic programming language, and Attribute Grammars (AGs) [25], having in mind to find classes of TYPOL programs based on the general classification of AGs. Thus, we deduce for TYPOL programs a specific, optimized, and unification-free resolution strategy.

TYPOL is a computer formalism for expressing *Natural Semantics* [21]. Such a semantic definition is identified with a logic made of axioms and inference rules: reasoning with the language is proving theorems within that logic. TYPOL is developed as a semantic specification formalism for an interactive language-based editor, CENTAUR [5], at INRIA.

On the one hand, TYPOL programs, under certain conditions, can be viewed as a generalization of *Primitive Recursive Schemes* introduced by Courcelle and Franchi-Zannettacci [8]. On the other hand, TYPOL programs are merely logic programs manipulating typed terms such as syntactic patterns. Following both approaches, a TYPOL domain can be viewed as a many-sorted algebra (see definitions in section 3.1).

We use results expressed in two complementary related works [8,9] to:

- propose characterizations of some subclasses of TYPOL programs in terms of formal properties of Primitive Recursive Schemes [16];
- give a refinement and an extension of the construction proposed in [9] that transforms logic programs into AGs;
- give an execution scheme in the spirit of *abstract interpretation* which induces a specific resolution strategy based on tree-walk AG evaluators instead of the general unification.

Deransart and Maluszynski [9] have already suggested a transfer of expertise between two *a priori* independent formalisms, Logic Programming and Attribute Grammars. We extend and apply this idea: TYPOL provides user-friendliness and expressive power; AG techniques provide efficiency.

We wish to address three problems that occur during execution of TYPOL specifications.

1. Gaining the benefits of memory optimization from results in AGs should eliminate the *space explosion* experienced in practice (some variables can be implemented as global variables, or stacks as described in [23,20]).
2. Some AG evaluators provide *incremental* computation (e.g. the Cornell Program Synthesizer [31]): changing a subtree in the parse tree leads to a minimal re-computation of the attribute values. The massive use of unification is an obstacle to any incremental facility.
3. Currently, semantics of TYPOL and standard PROLOG are strongly related (depth-first and left-to-right execution strategy). Our aim is to still express unification at the TYPOL level while using pre-compiled unification-less strategies during execution (see [4,22,8]).

This paper is related to other recent investigations: AGs and Logic Programming [17], AGs as a mean of expressing algorithms traversing data structures [18] and replacement of run-time unification in Prolog programs by term matching [26,14]. An adaptation of TYPOL, named UI-TYPOL, has been described in [32] to eliminate problems encountered with PROLOG and provide an efficient implementation via automated AG evaluators. However, unification is not provided in UI-TYPOL and UI-TYPOL defines a severely restricted subclass of TYPOL programs.

Section 2 outlines the TYPOL formalism, gives necessary notions and results concerning AGs, and points out the major differences and similarities between these two paradigms. Section 3 gives some characterizations of different subclasses of TYPOL programs and our construction mapping TYPOL specifications to AGs.

2 TYPOL and Attribute Grammars

2.1 Inference rules in TYPOL

TYPOL is a computer formalism for expressing *Natural Semantics* [21] for Programming Languages. By Natural Semantics, we mean a short, readable, and elegant mathematical style based on a structural operational semantics approach [28]. This approach insists on a natural deduction style in the sense of [29] and provides a proof-theoretic tool, with possible non-deterministic computations.

TYPOL is well-suited to specify type-checkers, interpreters, and translators. Natural semantic specifications are expressed in a relational and declarative style, and are straightforwardly executable. A user-friendly environment is now available for TYPOL programmers: a pretty-printer produces inputs for TEX (TYPOL examples shown here are produced using this pretty-printer); a compiler, including a type-checker, produces PROLOG code (executable under MU-Prolog [27]); a convivial debugger allows control of execution.

So, as a rule-based system executable with an inference engine, TYPOL may satisfy the requirements of both language designers (readability, expressive power of the specifications) and end-users (efficiency of the generated code).

For clarity, we give in this section an uncomplete presentation of the TYPOL formalism (see section 3.1. for formal definitions and [11,21,12] for a detailed description).

In TYPOL, a semantic specification is an *unordered* collection of inference rules, driven by the *abstract syntax* of the language(s) under description. Abstract syntax is a $\mathcal{O}$-*many-sorted algebra*, where $\mathcal{O}$, a set of *operators*, is a signature on $\mathcal{P}$, a set of *phyla*.

Syntactically, TYPOL inference rules are similar to PROLOG clauses. Intuitively, if all predicates of the clause body (named *premises*) hold, then the clause head (named *conclusion*) holds, and the inference rule applies.
A rule with no premises is called an *axiom*.
Premises are an *unordered* collection of atomic formulae: *sequents* and *conditions*. Sequents express the fact that some hypotheses are needed to prove a particular proposition; conditions express a restriction on the applicability of the rule. Both map to PROLOG predicates.

To run a TYPOL program means prove an *equation* using inference rules and axioms belonging to this program. Such an equation is a sequent.
The major specific features of TYPOL, compared with PROLOG, are:

1. Sequents have two parts: an *antecedent* (or *inherited positions*) and a *consequent* (denoting *synthesized positions*) separated by the turnstile symbol ($\vdash$);

2. The first argument of a consequent, the *subject*, is distinguished from the others. By extension, the subject of a rule is the subject of its conclusion. Subjects are abstract syntax terms (valid *tree patterns* w.r.t. their abstract syntax). Tree patterns may contain variables, denoting subtrees; this makes it possible to simultaneously describe the structure of the abstract syntax tree and refer to (select) its subtrees;

3. Sequents may have several forms (types) depending on the syntactic nature of their subjects. In other words, sequents are typed;

4. Structural rules are distinguished from rules concerned with auxiliary computations (e.g. the management of scope); these computations may be axiomatized with auxiliary rules, grouped into sets, referenced by a name (*named sequents*);

More formally, given $\mathcal{O}$ an abstract syntax, given S a finite set of sequent symbols with assigned arities and types denoting their subject (a term of $\mathcal{O}$), inherited positions, and synthesized positions, a *TYPOL rule* is a pair consisting of a *conclusion* and a finite set of *premises*.

During the execution process, *pattern-matching* on the subjects leads the construction of a *proof tree* for a TYPOL equation. Clearly, every TYPOL program can be straightforwardly simulated by a PROLOG program (and TYPOL equations turned into PROLOG goals). However, executing such programs by the inference engine of PROLOG, we loose a large part of TYPOL semantics, due to both the particular role of the subjects and the relationship between antecedent and consequent parts in a TYPOL sequent (positions dependencies are computed on the fly, at run-time, through the PROLOG unification).

We focus on TYPOL equations having the following property (well-oriented in the sense of [32]):

A sequent is an acceptable goal-equation if:

- *its subject is a ground term,*
- *its inherited positions are ground terms,*
- *its synthesized positions are variables.*

This additional hypothesis makes it possible to show that TYPOL equations are proved by *structural induction* [6] on a many-sorted algebra using full unification.

TYPOL specifications are said to be *data-driven* [9]: provided an acceptable goal, all sequents used during the proof have their inherited positions instanciated to ground terms.

Let's now illustrate our presentation with an example taken from semantic specifications expressed within the CENTAUR system [5].

Example 1: *The ASPLE type-checker within the TYPOL formalism*
ASPLE [13] is an Algol-like toy language with declarations of variables and simple statements such as the assignment, the if-statement, and the while-statement.

$$\frac{\rho_\emptyset \vdash \text{DECLS} : \rho \qquad \rho \vdash \text{STMS} : c}{\vdash \text{begin DECLS STMS end} : c} \tag{1}$$

$$\rho \vdash \text{decls}[\,] : \rho \tag{2}$$

$$\frac{\rho \vdash \text{DECL} : \rho_1 \qquad \rho_1 \vdash \text{DECLS} : \rho_2}{\rho \vdash \text{DECL; DECLS} : \rho_2} \tag{3}$$

The rule (1) expresses when an ASPLE program is well-typed: building an environment during the declarative part and verifying that the statement part is well-typed, one can conclude that the whole program is well-typed. The rule (2) is an axiom: an empty list of declarations doesn't modify the environment. At last, the rule (3) explains that the elaboration of declarations proceeds left-to-right.

2.2 Useful notations on Attribute Grammars

Since Knuth's initial paper [25], Attribute Grammars have been widely used in translation, compiler-compiler techniques and definitions for programming languages [22,33].

An *Attribute Grammar* is an abstract syntax $\mathcal{O}$ (a $\mathcal{P}$-signature) augmented with *semantic definitions* dealing with two disjoint finite sets of symbols: INH and SYN.
For each phylum $X \in \mathcal{P}$, we associate two disjoint finite sets of symbols: *inherited* (INH(X)) and *synthesized attributes* (SYN(X)).
For each operator $p \in \mathcal{O}, p : X_0 \rightarrow X_1 \cdots X_n$, semantic definitions describe *local dependencies* between the values of attributes:

form 1 INH(X_i) depend on INH(X_0) and SYN(X_i) for $i = 1, \ldots, n$
form 2 SYN(X_0) depend on INH(X_0) and SYN(X_i), for $i = 1, \ldots, n$

Let's apply these notions to the example 1.

Example 2.1: *The ASPLE type-checker within the AG formalism*

	ρ_{in}.DECLS := $env_\emptyset$;
program → DECLS STMS;	ρ_{in}.STMS := ρ_{out}.DECLS;
	c.program := *c*.STMS;
DECLS → ;	ρ_{out}.DECLS := ρ_{in}.DECLS;
	ρ_{in}.DECL := ρ_{in}.DECLS;
DECLS → DECL DECLS;	ρ_{in}.DECLS$_1$:= ρ_{out}.DECL;
	ρ_{out}.DECLS := ρ_{out}.DECLS$_1$;

Evaluate an AG with respect to a parse tree can be viewed as decorating the nodes in the parse tree with the values of attributes. The major area of active research in AGs is the design of automatically-generated efficient attribute evaluators (see [10] for an annotated bibliography). For this purpose, different subclasses (based on partial orders between attributes) have been introduced, and associated membership tests have been developed (e.g.

OAG [22], *l-ordered* [4], *FNC* [8]). With these (mathematically well-founded) subclasses, efficient (optimized, incremental) evaluators can be automatically generated by computing at generation time an evaluation order on attributes.

Let's focus now on the FNC subclass (Strongly Non-Circular), introduced by [24],[8] and implemented by [19]. As shown in [8], any Attribute Grammar in the FNC subclass can be transformed into a recursive tree-transducer [15] (namely a set of *Primitive Recursive Schemes*, p.r.s.) by a transitive closure algorithm on local dependencies to compute *potential global dependencies*. Intuitively, each synthesized attribute s is computed by a function taking as arguments any subtree t and inherited attributes that s may depend on.

More formally, we consider a function $\gamma : \mathcal{P} \times \text{SYN} \rightarrow \Pi(\text{INH})$
(named *argument selector*), defined as follows:

$\gamma(X, a) = \emptyset$ if $a \notin \text{SYN}(X)$
$\gamma(X, a) \subseteq \text{INH}(X)$ if $a \in \text{SYN}(X)$

The transitive closure algorithm computes the *minimal closed* argument selector γ_0 for all operators and synthesized attributes.
The characterization of the FNC subclass is given by the next theorem from [16]: *An AG is Strongly Non Circular iff γ_0 is non circular.*

Primitive Recursive Schemes stand for a functional subclass of logic programs. They are evaluated with a noetherian rewriting system based on a tree pattern-matching instead of the general unification mechanism [16].
Notice this transformation partly eliminates the low-level notation style of AGs definitions.

Let's apply this transformation on our example.
Example 2.2: *The ASPLE type-checker within the p.r.s. formalism*

$$c(program(\text{DECLS}, \text{STMS})) = c(\text{STMS}, \rho_{out}(\text{DECLS}, env_{\emptyset}))$$
$$\rho_{out}(decls[\,], \rho_{in}) = \rho_{in}$$
$$\rho_{out}(decls[\text{DECL}, \text{DECLS}], \rho_{in}) = \rho_{out}(\text{DECLS}, \rho_{out}(\text{DECL}, \rho_{in}))$$

2.3 Relationship between TYPOL and AGs

We show here in an informal way that TYPOL and AGs (namely Primitive Recursive Schemes) are closely related.
A TYPOL rule maps to a set of Primitive Recursive Schemes (as shown later). Tree patterns are described in the same way, with variables denoting subtrees. Synthesized attributes are expressed in the TYPOL consequent part, whereas inherited attributes compose the antecedent part.

Primitive Recursive Schemes are specified in a functional style, instead of the relational style of the TYPOL formalism. This difference is obviously minimal: a relational implementation of Primitive Recursive Schemes in PROLOG is described in [2,17].

However, there is a major difference between TYPOL and Primitive Recursive Schemes. The TYPOL formalism is based on the *tupling* of multiple Primitive Recursive Schemes to build a single TYPOL rule (see [17,18] for related works).

More formally, given a minimal closed argument selector $\gamma_0(X_0, a)$ (for $p \in \mathcal{O}, p : X_0 \rightarrow X_1 \cdots X_n$ and $a \in \mathrm{SYN}(X_0)$), we define the so-called *global argument selector* Γ as follows:

$$\Gamma(X_0) = \bigcup_{a \in \mathrm{SYN}(X_0)} \gamma_0(X_0, a)$$

This formulation suggests several remarks:

1. TYPOL specifications are more compact than the corresponding AG and more user-friendly than the corresponding Primitive Recursive Schemes;
2. Primitive Recursive Schemes are based on the notion of *minimal argument selector*. When tupling synthesized attributes, TYPOL also tuples inherited attributes and the corresponding argument selectors are no longer minimal; moreover they can introduce a false circularity inducing a (non necessary) least fix point process resolution;
3. the evaluation process is different: in the current implementation of TYPOL, PROLOG provides one single left-to-right pass over the parse tree instead of the standard multi-pass attribute evaluation: unification propagates upward the remaining computations, after actually evaluating non-instantiated variables.

3 Eliminating run-time unification

We focus our attention on a subclass of TYPOL programs that specify static semantics. By extension, we call such TYPOL programs *static* programs. We need first to introduce a formalism that describes TYPOL programs and we give characterizations of several subclasses of TYPOL programs. Next we present the major result of this paper: a two-step construction that transforms, under certain conditions, a static TYPOL program into an equivalent AG. Then, according to properties of this underlying AG, we can directly use AG techniques to implement TYPOL specifications without any unification mechanism. Within the scope of this paper, we focus on the practical aspects of our transformation and on implementation issues for the TYPOL language. For instance, an ASPLE type-checker implemented in LISP with a global variable denoting the environment ρ can be automatically generated from TYPOL specifications.

3.1 Definitions

We borrow from [9] the notion of *safe direction assignment* (related to the concept of *input-output directionality* in [30]) to give a formal definition of TYPOL programs.

Definition 1: Given S a finite set of sequent symbols with assigned arities and types, we define a *direction assignment*: $S \rightarrow \{\uparrow, \downarrow\}$, as a mapping of the arguments of each sequent symbol into *inherited* and *synthesized* positions.

Definition 2: Given a TYPOL rule and a direction assignment for, we call *input positions* (resp. *output positions*) the inherited positions of its conclusion and the synthesized positions of its premisses (resp. the synthesized positions of its conclusion and the inherited

positions of its premisses).

Definition 3: A direction assignment is said to be *safe* if, within each TYPOL rule, each variable in an output position occurs in some input position.

Definition 4:
A TYPOL program is a 7-uple $T =< S, \mathcal{O}, P, F, R, \mathcal{D}, E >$ where:
S is a finite set of sequent symbols with assigned arities and types;
$\mathcal{O}$ is an imported abstract syntax;
P is a finite set of predicate symbols with assigned arities and types;
F is a finite set of functor symbols with assigned arities and types;
R is a set of TYPOL rules constructed with S, $\mathcal{O}$, P and F;
$\mathcal{D}$ is a *safe direction assignment*;
E is an acceptable goal.

We can give an algebraic view of a TYPOL program as a many-sorted algebra: $M(S \cup \mathcal{O} \cup P \cup F)$. Notice a PROLOG program defines a one-sorted algebra $M(P \cup F)$. This major distinction between these two working domains leads our construction in next section.

Definitions 5:
A TYPOL program is *decreasing* if, for each of its inference rules r, subjects of the premises of r are proper subterms of the subject of r.
A TYPOL program is *linear* if, for each of its inference rules r, synthesized positions of premises of r have no common variables.
A TYPOL program is *deterministic* if, at each step of a proof, only one inference rule can apply.
A TYPOL program is *pseudo-deterministic* if all its inference rules have different subjects (this does not exclude non determinism within auxiliary sets).

Definition 6: We call *global argument selector* and we note $\Gamma(X_0)$ the set of all inherited positions of the conclusion of the rule whose subject is p and $p : X_0 \rightarrow X_1 \cdots X_n$.

Definition 7: A TYPOL program is *non circular* if its argument selector is non circular. Otherwise, this program is *pseudo-circular* and requires an unification process during execution.

As an immediate consequence of the previous definitions and formal properties studied in [16], we formulate useful propositions and remarks.

Proposition 1: All decreasing non circular TYPOL programs are equivalent to Primitive Recursive Schemes, assuming a splitting of the global argument selector into minimal ones.

Proposition 2: For all decreasing non circular TYPOL programs, the associated resolution relation is noetherian. Moreover, if the TYPOL program is deterministic, this relation

is also confluent (as in example 1).

Remark 1: For a decreasing pseudo-circular TYPOL program, there may exist a splitting of its global argument selector into non circular minimal ones. Such a TYPOL program is thus equivalent to a set of Primitive Recursive Schemes (as in example 3).

Remark 2: If such a splitting does not exist, the TYPOL program expresses a real least fix point and is equivalent to schemes with fix point introduced by [6].

We give now the definition of *Conditional AGs* (CAGs). This extension of AGs is named in [9] *Functional AGs* as opposed to *Relational AGs*, two extensions introduced to supply the relational power of logic programming to AGs. Intuitively, CAGs are standard AGs augmented with *semantic rules* that make it possible to express some *conditions* concerning values of the input attributes.

Definition 8: A *CAG* is a 4-uple $< \mathcal{O}, \text{ATTR}, R, \mathcal{I} >$ where:

$\mathcal{O}$ is an abstract syntax;

ATTR is the union of a family of finite sets of symbols ATTR(X).
For each phylum $X \in \mathcal{P}$, ATTR(X) is the union of two disjoint sets INH(X) and SYN(X);
For each operator $p \in \mathcal{O}, p : X_0 \rightarrow X_1 \cdots X_n$
$INPUT(p) = \text{INH}(X_0) \cup \text{SYN}(X_i), i = 1, \ldots, n;$
$OUTPUT(p) = \text{SYN}(X_0) \cup \text{INH}(X_i), i = 1, \ldots, n.$

R is the union of a family of *semantic definitions* and *semantic rules.*
For each operator $p : X_0 \rightarrow X_1 \cdots X_n$:

- semantic definitions express the computation of OUTPUT(p) according to both forms presented in section 2.2;
- semantic rules express conditions between values of INPUT(p);

$\mathcal{I}$ is an *interpretation*: the resulting domains reached during the computation of the attributes.

Notice that in a PROLOG implementation of CAGs, this extension is straightforwardly supported by the unification step; otherwise, CAGs are not really suited for computational applications.

We also need the auxiliary notion of selectors defined in [9]. A *selector* is a partial operation on terms: for a given term t of the form $f(t_1, \cdots, t_n)$ the selector $s_i f(t)$ is defined to be t_i.

3.2 Transformation Algorithm

We describe here our construction: in a first step, we design a CAG from a TYPOL program and in a second step, and with some additional hypotheses, we implement a CAG with an equivalent standard AG without conditions.

We focus on the practical aspects of our transformation such as characterization of several subclasses of TYPOL programs and their optimized implementation using AGs, rather than theoretical aspects such as semantical equivalence between TYPOL programs and AGs.

Since TYPOL programs define a particular subclass of logic programs, our construction relating TYPOL programs and AGs differs from the construction given in [9] for two aspects:

1. the syntactic nature of the subjects of TYPOL programs allows us to keep such patterns to design the underlying abstract syntax of the equivalent CAG. Thus, our approach avoids tedious steps in the contruction proposed by Deransart and Maluszynski;

2. as Deransart and Maluszynski use a one-to-one correspondence between clauses and productions rules of the context-free grammar, their construction does not exclude non determinism. We must consider only pseudo-deterministic TYPOL programs to make sure that any attribute is defined by only one semantic definition. Non determinism in auxiliary sets is simply mapped into non determinism in computation functions. However, we can transform any TYPOL program into an equivalent pseudo-deterministic one: if two inference rules have the same subject, we merge them together, transfering non determinism of inference rules to a (possible) non determinism of the functors used in the inference rules.

Now, we define the construction:
Construction 1: *from TYPOL programs to CAGs*
Let $T =< S, \mathcal{O}, P, F, R, \mathcal{D}, E >$ be a decreasing pseudo-deterministic TYPOL program, we construct a CAG $G =< \mathcal{O}, \text{ATTR}, R, \mathcal{I} >$ defined as follows:

1. $\mathcal{O}$ is the abstract syntax imported in T.

2. The set ATTR is the union of inherited and synthesized positions of all phyla in $\mathcal{P}$; the attributes are named from the positions of the arguments in the sequents: $\text{ATTR}(X) = \{X.i / i = 1, \cdots, n\}$.

3. For each rule r, semantic definitions are constructed as follows:

   - For each output position a of r, let t_a be the term at this position. For a variable x in t_a let b be an input position including x. Denote by $S_x b$ the set of all composed selectors s such that $s(t_b) = x$. The semantic definition for a is of the form:

     $$a = \alpha(t_a)$$

     where α is a substitution assigning to each variable x in t_a the term $s(b)$ for some $s \in S_x b$.

   - For each pair of different occurrences of a variable x at input positions b_1 and b_2 of the rule r, we construct the condition:

     $$s_1(b_1) = s_2(b_2)$$

     where s_1 and s_2 are the selectors corresponding to the considered occurrences of x in the terms at the positions b_1 and b_2.

4. $\mathcal{I}$ is the underlying interpretation of predicates of P and functors of F, augmented with the interpretation of used selectors.

Proposition 3: Let T be a decreasing pseudo-deterministic TYPOL program and let G be the CAG obtained by Construction 1. The proof trees of T are isomorphic to attributed syntax trees of G.

The proof is based on the two required properties: if the TYPOL program is non-decreasing, semantic definitions may be neither in the form 1 nor in the form 2; moreover, if it is not pseudo-deterministic, several definitions of the same attribute may be generated.

Remark 3: If the TYPOL program is also linear, the resulting CAG does not include any condition: it is a standard AG and AG techniques are directly usable to implement TYPOL programs. On the other hand, if the TYPOL program is moreover non circular, the resulting AG belongs to a rather trivial subclass of AGs.

Now, we can formulate the major result of our work:

Proposition 4: Let T be a decreasing linear pseudo-deterministic TYPOL program and G be the equivalent standard AG. If G belongs to the FNC subclass, there exists a splitting of the global argument selector into minimal non-circular argument selectors. Thus, TYPOL specifications can be executed without any unification (as described in example 3).

Example 3: *From regular expressions to deterministic automata*
This example describes how to build a finite automaton from a regular marked expression (see [1,3] for the detailed algorithm).

The syntax of regular expressions over a set of input symbols is, as usual:

$axiom ::= E\ end$

$E ::= 0 \mid \epsilon \mid I \mid E + E \mid E \cdot E \mid E^*$ where *end* is an endmarker symbol;

Definitions

$L(E)$ denotes the language generated by a regular expression E.

$\text{first}(E) = \{a \mid av \in L(E)\}$

$\text{follow}_E(a) = \{b \mid uabv \in L(E)\}$

$\delta(E)$ stands for *true* if the empty string belongs to L(E);
otherwise, $\delta(E)$ stands for *false*.

The TYPOL inference rules compute inductively the value δ and two sets simultaneously, the set ρ of pairs of the form $<a, \text{follow}_E(a)>$ and the set h, the *first* set of E. These three components define the automaton that recognizes the given regular expression.

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

$$\text{S} \vdash \text{I} : \ <\text{I} \mapsto \text{S}, \text{I}, false> \quad (2)$$

$$\text{S} \vdash \epsilon : <\emptyset, \emptyset, true> \quad (3)$$

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

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

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

Construction 1 applied to the previous TYPOL rules results in a standard FNC AG. This AG is expressed within the p.r.s. formalism, assuming a renaming of attributes, as follows:

$\rho(axiom(E)) = \rho(E, end)$
$h(axiom(E)) = h(E)$
$\delta(axiom(E)) = \delta(E)$

$\rho(+(E_1, E_2), s) = \rho(E_1, s) \cdot \rho(E_2, s)$
$h(+(E_1, E_2)) = h(E_1) \cdot h(E_2)$
$\delta(+(E_1, E_2)) = \delta(E_1) \vee \delta(E_2)$

$\rho(sym(), s) = <sym, s>$
$h(sym()) = sym$
$\delta(sym()) = false$

$\rho(\cdot(E_1, E_2), s) = \rho(E_1, h(E_2) \cup \delta(E_2) \cdot s) \cdot \rho(E_2, s)$
$h(\cdot(E_1, E_2)) = h(E_1) \cup \delta(E_1) \cdot h(E_2)$
$\delta(\cdot(E_1, E_2)) = \delta(E_1) \wedge \delta(E_2)$

$\rho(\epsilon(), s) = \emptyset$
$h(\epsilon()) = \emptyset$
$\delta(\epsilon()) = false$

$\rho(*(E), s) = \rho(E, h(E) \cdot s)$
$h(*(E)) = h(E)$
$\delta(*(E)) = true$

This standard AG is now evaluated as follows: during a first bottom-up pass, δ and h are both computed and loaded in some particular nodes of the parse tree; then, in a left-to-right pass, ρ is computed. Thus, the least fix point expressed in the rule (6) is no longer solved with the underlying unification step but implemented with attributes evaluation. Of course, this is exactly the evaluation mechanism described in [1].

Now let us study the case where the CAG resulting from our construction 1 contains conditions. We agree with Deransart and Maluszynski: from a practical point of view, CAGs are not directly usable in computational applications. Our construction 2 is then a practical contribution on CAG evaluation. We propose a simulation of the evaluation of any CAG having certain sufficient conditions in terms of standard AGs. We don't focus here on the formal transformation between CAGs and standard AGs.

Intuitively, our transformation algorithm consists in *orienting* conditions to induce an evaluation order between two *a priori* independent attributes. This evaluation order must be compatible with the evaluation order provided by the underlying standard AG. We break the condition rule with two intermediate attributes: the first one is an output attribute that propagates its value to the second one, an input attribute. Therefore, the computation of the second attribute must take into account this new information concerning its *linked* attribute.

Construction 2: *from CAGs to standard AGs*
Given a non-circular CAG $G =< \mathcal{O}, \text{ATTR}, R, \mathcal{I} >$, we construct a standard AG $G' =< \mathcal{O}, \text{ATTR'}, R', \mathcal{I}' >$ defined as follows:

1. ATTR' is ATTR augmented with the attributes introduced during the construction of R' as follows:

2. For each operator $p : X_0 \rightarrow X_1 \cdots X_n$ for which a condition is expressed between b_i and b_j (input attributes of X_i and X_j). Since the underlying standard AG is non-circular, it exists a partial order between b_1 and b_2. Let b_1 be computed before b_2. We introduce two new attributes x and y for X_j and we replace the condition by the following semantic definitions:
   $$x := s_1(b_1)$$
   $$y := \text{fcond}(s_1(b_1) = s_2(b_2), b_2, \perp)$$
   where $\perp$ is the overloaded undefined value on attribute domains.

3. the interpretation $\mathcal{I}'$ is $\mathcal{I}$ augmented with interpretation of the *fcond* function.

This construction suggests several remarks:

- our notion of intermediate attributes generalize the notion of *conditional attributes* of the GAG system [23];

- following terminology of Maluszynski and Komorowski [26], we can apply straightforwardly our construction in the TYPOL domain to transform any truly non circular TYPOL program into an equivalent linear one, assuming that positions inducing the non-linearity are *pre-saturated* (input positions are evaluated to ground terms); in that case TYPOL specifications can be executed with a single mechanism of pattern-matching, instead of a general unification. Such a transformation is illustrated in example 4;

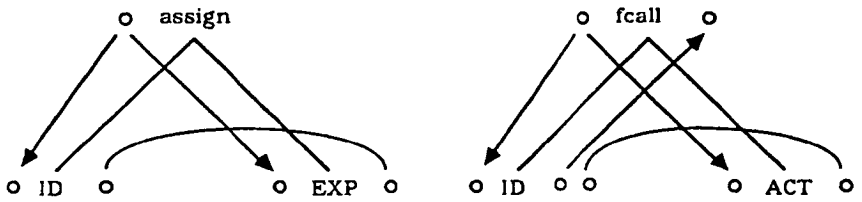
- if the condition concerning pre-saturated positions in not verified, our transformation leads to a standard AG augmented with an explicit unification function. Disconnecting the unification mechanism from other attributes evaluation, we propose a mixed strategy : optimized evaluation techniques for attributes not involved in unifiability conditions and a specific unification handler for other attributes. Such a strategy is needed to execute the Mini-ML type checker expressed in the TYPOL formalism [7].

Example 4: *unification-free overloading resolution in ASPLE*
This example comes from the ASPLE type-checker [13] augmented with subprogram declarations (possibly overloaded) specifications and overloading resolution specifications for subprogram calls. To specify this extension, we introduce TYPOL rules that update the environment ρ during elaboration of declarations. We also add some non-linear rules to verify the statement part (namely subprogram calls) is well-typed.

$$\frac{\rho \overset{\text{type\_of}}{\vdash} \text{ID} : t \qquad \rho \vdash \text{EXP} : t}{\rho \vdash \text{ID} := \text{EXP}} \tag{1}$$

$$\frac{\rho \overset{\text{type\_of}}{\vdash} \text{ID} : t \cdot m \qquad \rho \vdash \text{ACTUALS} : m}{\rho \vdash \text{ID}(\text{ACTUALS}) : t} \tag{2}$$

- the first rule deals with assignment statement and the second one with the function-call statement;
- *type_of* is a named sequent that specifies how to find the type of an identifier in the environment;
- t denotes the result type of a function and m the types of the arguments of any subprogram;
- the underlying CAG given by Construction 1 on these rules can be sketched as follows:



- if identifiers are overloaded, failure cases are early detected, because of the unification mechanism.

Construction 2 transforms the two previous TYPOL rules into the next two linear rules:

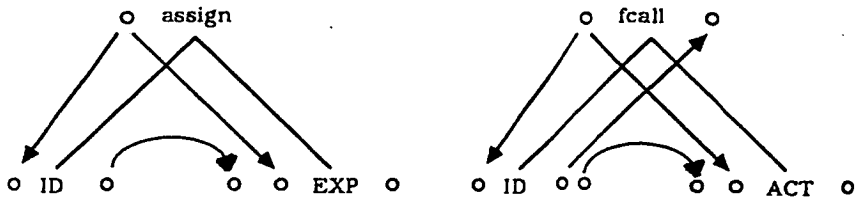
$$\frac{\rho \overset{\text{type\_of}}{\vdash} \text{ID} : t \qquad \rho, t \vdash \text{EXP} : t'}{\rho \vdash \text{ID} := \text{EXP}} \tag{1'}$$

$$\frac{\rho, t \overset{\text{type\_of}}{\vdash} \text{ID} : t' \cdot m \qquad \rho, m \vdash \text{ACTUALS} : m'}{\rho, t \vdash \text{ID}(\text{ACTUALS}) : t'} \tag{2'}$$

This transformation suggests several remarks:

- these TYPOL rules are unification-free: they can be executed with term matching;
- *type_of*, with an additional argument t, is a named sequent that finds a possible type t' of an identifier in the environment, and verifies equality of the two instantiated variables t and t';

- the underlying standard AG given by Construction 2 on these rules can be sketched as follows:



- the efficiency provided by the unification mechanism in the starting example is preserved by the type constraint propagation: failure proof subtrees are not explored during overloading resolution.

4 Conclusion

As a temporary conclusion to this work, we expect from further developments:

- an improvement of the performances of the CENTAUR system and a better programming environment based on the analysis and the classification of the TYPOL programs;
- an extension of AGs (and Primitive Recursive Schemes) to the Dynamic Semantics in the spirit of TYPOL programs;
- some results for the compilation and the translation of PROLOG programs, considering a PROLOG program (annotated with a direction assignment) as a TYPOL program concerned with an abstract syntax to be defined;

References

[1] Aho A., Sethi R., & Ullman J. "Compilers: Principles, Techniques, and Tools" Addison-Wesley, Reading, Mass., 1986

[2] Attali I. & Franchi-Zannettacci P. "Prolog-like schemes for Ada static semantics" Ada UK news, 6, 2, 1985

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

[4] Bochmann G. "Semantic evaluation from left to right" CACM 19, 2, 55-62, 1976

[5] Borras P., Clément D., Despeyroux T., Incerpi J., Kahn G., Lang B., & Pascual V. "CENTAUR: the system" INRIA research report 777, 1987

[6] Chirica L. & Martin D. "An order algebraic definition of Knuthian semantics" Math. Systems Theory 13, 1979

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

[8] Courcelle B. & Franchi-Zannettacci P. "Attribute Grammars and Recursive Program Schemes" TCS 17, 163, 1980

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

[10] Deransart P., Jourdan M., & Lorho B. "A survey on Attribute Grammars, Part III: classified bibliography" INRIA research report 417, 1985

[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] Donzeau-Gouge V., Kahn G., & Lang B. "A complete machine checked definition of a simple programming language using denotational semantics" INRIA research report 330, 1978

[14] Drabent W. "Do logic programs resemble programs in conventional languages ?" 1987 IEEE Symp. on Logic Programming, San Francisco, 1987

[15] Engelfriet J. & File G. "The formal power of one-visit attribute grammars" Proc. of the seventh ICALP, LNCS 85, 1980

[16] Franchi-Zannettacci P. "Attributs sémantiques et schémas de programmes" Thèse d'Etat, Univ. of Bordeaux I, 1982

[17] Ganzinger H. & Hanus M. "Modular Logic Programming of Compilers" Proc. Symp. on Logic Programming, Boston, Massachusetts, 242-253, 1985

[18] Johnsson T. "Attribute Grammars as a Functional Programming Paradigm" Symp. on Functional Programming Languages and Computer Architecture LNCS 274, Portland, 1987

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

[20] Julié C. "Optimisation de l'espace mémoire pour les compilateurs générés selon la méthode d'évaluation OAG: Etude des travaux de Kastens et propositions d'améliorations" Rapport de DEA, Univ. of Orléans, 1986

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

[22] Kastens U. "Ordered Attribute Grammars" Acta Informatica 13, 1980

[23] Kastens U. "The GAG-system - A Tool for Compiler Construction" in *Methods and Tools for Compiler Construction*, ed B. Lorho, 165-182, Cambridge University Press, 1984

[24] 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

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

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

[27] Naish L. "Negation and Control in Prolog" LNCS 238, 1986

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

[29] Prawitz D. "Ideas and results in proof theory" Proc. of the Second Scandinavian Logic Symposium, North Holland, 1971

[30] Reddy U. S. "On the relationship between logic and functional languages" in *Logic Programming: Functions, Relations and Equations*, DeGroot D., Lindstrom G. eds., Prentice Hall, 1986

[31] Reps T. "Generating Language based Environments" M.I.T. Press, Cambridge, Mass, 1984

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

[33] Uhl J., Drossopoulou S., Persch G., Goos G., Dausmann M., & Winterstein G. "An Attribute Grammar for the semantic analysis of Ada" LNCS 149, 1982

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