Integrating Natural Semantics and Attribute Grammars: the Minotaur System

Isabelle Attali, Didier Parigot

INRIA
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Integrating Natural Semantics and Attribute Grammars: the Minotaur System

Isabelle Attali , Didier Parigot

N° 2339

Septembre 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Integrating Natural Semantics and Attribute Grammars: the Minotaur System

Isabelle Attali *, Didier Parigot **

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projets CROAP et ChLoE

Rapport de recherche n° 2339 — Septembre 1994 — 16 pages

Abstract: This paper describes the principles and the functionalities of the Minotaur system. Minotaur is a generic interactive environment based on the integration of the Centaur system and the FNC-2 system, two systems widely used to specify syntax and semantics of programming languages and generate efficient semantic tools from these specifications. We show how Attribute Grammars techniques can be adequate for evaluation of a quite large subclass of Natural Semantics specifications, including specifications of an arithmetic calculator, a tree transformation, a type-checker for an Algol-like language, ...
For this subclass of Natural Semantics specifications, the Minotaur system automatically generates an incremental and efficient (in time and memory) evaluator which gives to Natural Semantics an industrial strength implementation.

Key-words: Specifications, Natural Semantics, Attribute Grammars, Programming Environments

(Résumé : tsvp)

*Isabelle.Attali@sophia.inria.fr
**Didier.Parigot@inria.fr

Sémantique Naturelle + Grammaires Attribuées = le système Minotaur

Résumé : Ce rapport décrit les principes et les fonctionnalités du système Minotaur. Minotaur est un environnement générique interactif fondé sur l'intégration des systèmes Centaur et FNC-2, deux systèmes largement utilisés pour la spécification des langages de programmation (syntaxe et sémantique) et la génération, à partir de ces spécifications, d'outils sémantiques puissants. Nous montrons comment les techniques issues des Grammaires Attribuées sont applicables à l'évaluation d'une classe large de spécifications en Sémantique Naturelle, incluant par exemple une calculatrice, une transformation d'arbre, un vérificateur de typage pour un langage de type Algol, ...
Pour cette classe de spécifications en Sémantique Naturelle, le système Minotaur génère automatiquement un évaluateur incrémental et performant (en temps et en mémoire), ce qui confère à la Sémantique Naturelle un schéma d'exécution adapté aux applications de taille réelle.

Mots-clé : Spécifications, Sémantique Naturelle, Grammaires Attribuées, Environnements de Programmation

1 Introduction

This paper describes the principles and the functionalities of the Minotaur system. Minotaur is a generic interactive environment based on the integration of the Centaur system [9, 19] and the FNC-2 system [23, 24, 22], two systems widely used to specify syntax and semantics of programming languages and generate efficient semantic tools from these specifications. The Centaur system is dedicated to graphical interactive tools for program interpretation using Natural Semantics [27] and the FNC-2 system is devoted to compilation and transformation applications via Attribute Grammars (AGs) [30].

Natural Semantics and Attribute Grammars are well-known methods for the specification of semantics on an abstract syntax. Both formalisms are expressive, declarative, and are straightforwardly executable (the reader could refer to [13, 14, 18, 27, 30, 38, 39] for more details about these methods). However, these two frameworks are used in different contexts, depending on the nature of the application: Attribute Grammars have proved to be a useful formalism for static program analysis (type-checking, translations) while Natural Semantics, in addition, deals with program interpretation. This major difference (static vs dynamic semantics) is due to the evaluation processes on a given abstract syntax tree.

In Attribute Grammars, this tree is decorated with attribute values in a static and deterministic way, provided there is no circularity in the dependency graph between attributes, but the tree itself is constant during the computations. These evaluation aspects have been widely investigated and the main results consist in varied strategies depending on the class of the AG (see a complete bibliography in [13], and a survey in [1]). For instance, Strongly Non Circular (SNC) AGs are usually implemented in a recursive tree-walk [12, 20, 17] and Ordered AGs are evaluated in an iterative algorithm based on visit sequences [28]. All these strategies have been designed with two main goals: (1) provide a reasonable memory cost (since the number of attributes is usually tremendous) and (2) provide an incremental re-evaluation after a tree transformation. As a consequence, most modern attribute systems (see Synthesizer Generator, FNC-2, Mercury, Mjølner/Orm) include powerful memory management and incremental evaluation [17, 25, 29, 32, 33, 35, 39] and AGs have been repeatedly and successfully used in real applications such as construction of editors, translators, and compilers.

On the other hand, the framework of Natural Semantics lacks an industrial strength implementation despite a large theoretical background (Natural Deduction [18] and Structural Operational Semantics [38]). Natural Semantics rules define a logic and are used as a proof-theoretic tool to prove theorems within that logic, building proof trees in a recursive top-down strategy involving unification. This kind of execution model is close to the tree-walk strategies in AGs except that the proof tree in Natural Semantics is not always directly related to the input tree. Moreover, proof tree building is not implemented in a satisfactory manner regarding memory storage and incrementality features: to turn Natural Semantics definitions into executable code, we chose to translate Typol rules (our computer version of Natural Semantics in the Centaur system [14]) into Prolog clauses, taking advantage of the similarity of Prolog variables and variables in inference rules. Considering our experiments in the Centaur system, this implementation is not adequate for four reasons:

1. the operational semantics of Prolog (depth-first and left-to-right execution strategy) is a constraint on the Typol formalism;

2. dynamic type checking (using possibly delayed constraints) is required in Prolog to safely evaluate Typol specifications.

3. the memory storage in Prolog is not optimal for Typol evaluation (due to the management of backtracking) and we cannot handle real-size applications (or if we want to do so, we need to pollute the specification with *cuts*);

4. the proofs in Prolog are not built in an incremental manner and it is not efficient to re-build a complete proof tree for a semantic treatment during the editing process.

From this statement, our goal is to characterize a subclass of Natural Semantics powerful enough to describe an extensive collection of semantics (both in static and dynamic domains) and efficient enough to go a step further than toy examples and manage real-size applications. It has been proven that, for any specification in the addressed subclass, there exists an Attribute Grammar which evaluates it [5, 2]. We propose here a practical application of these theoretical results in order to provide an incremental and optimized implementation of Natural Semantics definitions (in this subclass) via an automatically generated attribute evaluator. A similar approach has been discussed in [41], but with a severely restricted subclass of Typol programs, called UI-TYPOL, where unification is not allowed. Prolog implementation

of Natural Semantics was also discussed and criticized in [37], and a new language (RML) and implementation strategy for Natural Semantics were suggested via a Continuation Passing Style representation to C code.

To reach our goal, we had to design (or use) tools such as:

- a membership test to determine whether a given Typol specification can be evaluated by an AG;
- a translator from Natural Semantics to Attribute Grammars;
- an efficient attribute evaluator generator;
- an interactive setting for the generated evaluators.

Our solution results in the Minotaur system: a combination of Centaur and FNC-2 which addresses real-size semantic applications. Minotaur uses the FNC-2 processors at generation-time to build attribute evaluators from semantic specifications and the Centaur components for syntactic features and user-interface at run-time. This combination has been possible because both Centaur and FNC-2 are open systems: on one hand, Centaur provides a collection of accessible tools such as a kernel for the representation and manipulation of abstract syntax trees, primitives for the man-machine interface and for communication; on the other hand, FNC-2 is flexible enough to generate an attribute evaluator acting on abstract syntax trees coming from Centaur. This kind of connection is very difficult in closed attribute systems because they have their own internal representation for trees and then duplication of data structures and data exchanges are mandatory for interaction.

We have used the Minotaur system for several applications now and positive experimental results tend to prove that we have reached our goal to get an efficient evaluation of powerful semantic specifications in term of time, memory storage and incrementality features.

The next section presents Natural Semantics and its implementation in the Centaur system. Section 3 is dedicated to Attribute Grammars and the FNC-2 system. In Section 4, we compare both frameworks, discuss the expressive power of the considered subclass for attribute evaluation, and outline the translation scheme from Natural Semantics to Attribute Grammars. Section 5 briefly describes the architecture of the Minotaur system. In Section 6, we establish the validity of our approach on a example of practical interest and we evaluate the performance of our system (in terms of memory and time). Finally, Section 7 concludes the paper, and suggests new directions for the future.

2 Natural Semantics in the Centaur System

Semantic aspects in the Centaur system are handled with *Natural Semantics* [27] and its implementation, the Typol formalism [14]. The general idea of a semantic definition in Natural Semantics is to provide axioms and inference rules that characterize semantic behaviors to be defined on language constructs. Within Natural Semantics, a semantic definition is identified with a logic and reasoning with the language is proving theorems within that logic. Axioms and rules are used as a proof-theoretic tool to generate new facts (proof trees) from existing facts in a non-deterministic manner due to the relational presentation of the formalism. Thus there can be several proof trees for the same fact.

In the Typol formalism, inference rules indicate how a *sequent* (the *conclusion* of the rule) expressing some relation between some hypothesis and some property on an abstract syntax term, the *subject*, may be deduced from other sequents or predicates (called *premises*).
Typol rules are of the form:

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

The numerator can also contain *predicates* (for auxiliary computation) of the form:

$$pred(\alpha_1, \cdots, \alpha_l \rightarrow \beta_1, \cdots, \beta_m)$$

Variables may occur anywhere in a rule and allow this rule to be instantiated during a proof. The terms H, H_i are called *inherited* positions, and the terms S, S_i *synthesized* positions. Notice that positions $H, H_1, \cdots, H_n, S, S_1, \cdots, S_n$ may be tuples (as in Figure 1). The sequents $H \vdash T : S,\ H_i \vdash T_i : S_i$ are strongly typed and must be declared with *judgements*, which make it possible to perform type inference

on variables. Valid abstract syntax patterns (the subjects) belong to some *formalism* (namely an order-sorted algebra composed with *operators* and *sorts*). Other variables can also be typed with predefined types such as string or integer.

From a Typol rule, we define the set input(r) of *input positions* composed from the positions H, $S_1, \cdots, S_n, \beta_1, \cdots, \beta_m$ and the set output(r) of *output positions* composed from the positions $S, H_1, \cdots, H_n$, $\alpha_1, \cdots, \alpha_l$. These two sets are needed to define the flow of informations in a proof tree. Roughly speaking, input positions are computed by the outer context and used in rule r to compute the output positions which are then transmitted to the outer context. This kind of flow ensures that all output positions are computable.
A typical Typol rule coming from the specification developed in section 6.1 (see Figure 5) is shown below:

$$\frac{\varphi \vdash \mathrm{EXP}_1 : \alpha_1, \sigma_1, \epsilon_1 \qquad \varphi \vdash \mathrm{EXP}_2 : \alpha_2, \sigma_2, \epsilon_2 \quad \mathrm{append\_arcs}(\alpha_1, \alpha_2 \rightarrow \alpha_3) \qquad \mathrm{append\_states}(\sigma_1, \sigma_2 \rightarrow \sigma_3) \qquad \mathrm{or}(\epsilon_1, \epsilon_2 \rightarrow \epsilon_3)}{\varphi \vdash \mathrm{plus}(\mathrm{EXP}_1, \mathrm{EXP}_2) : \alpha_3, \sigma_3, \epsilon_3;}$$

Figure 1: A Typol rule

In this rule, the subject is a *plus* expression defining a language recognized by some automaton to be defined. Automata are characterized by three components: a set of arcs (α), a set of initial states (σ), and a boolean (ϵ) expressing whether or not the empty string belongs to the generated language. The φ variable denotes the set of states following the current state. The resulting automaton for a *plus* of two sub-expressions is inductively computed from the two sub-automata: the resulting set of arcs is the union of the two sets of arcs, the resulting set of initial states is the union of the two sets of initial states, and the empty string belongs to the resulting language if it belongs to one of the two languages.

In Natural Semantics, proofs are done using *structural induction* on subjects since the initial goal to prove contains a complete abstract syntax term (i.e. *is this program well-typed ?*). This property links the given abstract syntax term and the resulting proof tree for the initial goal. Depending on some properties of the Typol rules (see Section 4), the resulting proof tree can be bigger than the given abstract syntax term (and even infinite); this makes it possible to express Dynamic Semantics in Natural Semantics.

In the setting of interactive programming environments, *incrementality* is considered as a major goal for efficiency and user-friendliness reasons (see for instance [6, 32, 33, 39, 43]). The Centaur system also enhances interactivity in syntactical aspects. Parsing is incremental: any sentence of the object language corresponding to a valid sort can be parsed and an abstract syntax subterm is constructed. Pretty-printing of abstract syntax trees is also incremental so the user is not annoyed with screen redisplays during the editing process.
However, semantical aspects in the Centaur system can not be handled in an incremental manner and a call to a type-checker or a translator on the whole program has to be done on request after this program has been edited and (even slightly) modified. As in [3, 4], we consider the current implementation in Prolog (with unification and backtracking) as an obstacle to any attempt towards incrementality. Moreover, our applications in Typol are more or less limited to toy examples due to overflows in the Prolog stack.

3 Attribute Grammars in the FNC-2 System

Since Knuth's initial paper [30], Attribute Grammars have been widely used in translation, compiler-compiler techniques and definitions for programming languages. In this section, we recall some basic notations about Attribute Grammars and we present the major features of the FNC-2 system, an AG processing system; we focus more particularly on its specification language Olga and the advantages of the generated attribute evaluators.

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 (attributes): INH and SYN.
For each phylum $X \in \mathcal{P}$, we associate two disjoint finite sets of symbols: *inherited* and *synthesized attributes* ($\mathrm{INH}(X), \mathrm{SYN}(X)$).
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 and define $\mathrm{SYN}(X_0)$ and $\mathrm{INH}(X_i)$ (*output attributes*) in terms of $\mathrm{INH}(X_0)$ and $\mathrm{SYN}(X_i)$ (*input attributes*).

Evaluating an AG with respect to an abstract syntax tree can be viewed as decorating the nodes in the tree with the values of attributes. The major area of active research in AGs is the design of automatically-generated efficient attribute evaluators (see [13] 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* [28], *l-ordered* [8], *SNC* [12, 20, 17]). With these subclasses, efficient (optimized, incremental) evaluators can be automatically generated by computing at generation time an evaluation order on attributes.

The specificity of the FNC-2 system [23, 24, 22] is to consider that an Attribute Grammar specifies, and an attribute evaluator implements, an attributed-tree to attributed-tree mapping. In this scheme, the intermediate trees are described by abstract syntaxes extended with attribute declarations. In the FNC-2 system, Attribute Grammars are written in Olga [21] and specify the computations performed by one or more passes, according to some input and output data (attributed abstract trees). The Olga formalism was designed for the description of all aspects of an AG. This, of course, involves constructs to declare attributes, access them in semantic rules, but also constructs to describe pure calculations without resorting to an external language. Moreover, the Olga formalism has been extended with the notion of *pattern-based tree attribution* [15]: each operator is expressed as a pattern with variables.

A typical Olga rule coming from the specification developed in Section 6.1 (see Figure 6) is shown below:

```
where plus -> EXP1 EXP2  use
        $epsilon := or($epsilon(EXP1), $epsilon(EXP2));
        $alpha := append_arcs($alpha(EXP1),$alpha(EXP2));
        $sigma := append_states($sigma(EXP1),$sigma(EXP2));
        $phi(EXP1) := $phi;
        $phi(EXP2) := $phi;
end where ;
```

Figure 2: An Olga rule

The major advantages of the FNC-2 system are:

- its expressive power: the accepted class of attribute grammars is the *SNC* class, which is large enough to cover usual needs for static semantics in programming languages;
- the efficiency of the generated evaluators: the generated evaluators are deterministic, since they are based on a total evaluation order on each sort of the abstract syntax, thanks to a transformation from the *SNC* subclass to the *l-ordered* one (see [24, 22, 35] for more details);
- the flexibility of the generated evaluators: from the same specification, the generated evaluators can be alternatively implemented with or without memory optimization, in an exhaustive or incremental manner, in a sequential or parallel manner, and in C or Lisp.

The FNC-2 system provides a very fine static analysis of the lifetime of each attribute instance, which in turn allows to determine the most efficient way to store it. This is possible because of the necessity to have a statically-determinable total evaluation order to produce visit-sequence-based evaluators (as described in [29]). In [25], we give exact conditions to decide whether a temporary attribute can be stored in a global variable rather than in a stack (all temporary attributes can be, at worst, stored in a stack) and whether a non-temporary attribute can be stored in a global variable.

Finally, the evaluator generator is able to produce incremental attribute evaluators (see [1, 39] for related work). Our method, presented in [24, 22, 35], is based on the subclass of SNC AGs called *Doubly Non-Circular* (DNC) [17]. This class is however larger than the l-ordered class, and our SNC to l-ordered transformation makes it possible to actually use this method for any SNC AGs. Thus, we can generate an evaluator which is able to start at any node in the tree. Moreover, a set of "semantic control" functions allow to limit the re-evaluation process to affected instances.

4 From Natural Semantics to Attribute Grammars

Relationship between Natural Semantics and Attribute Grammars has been discussed in [2, 5]: a subclass of Typol programs has been defined and a translation scheme from this subclass to Attribute Grammars has been developed in order to provide an optimized implementation of Typol programs, without unification when possible. To avoid the re-definition of this subclass, we briefly compare both approaches in terms of style, expressiveness, and execution process. This comparison will naturally raise the main restrictions required on a Typol program to get an attribute evaluation.

4.1 Comparison of the two formalisms

From the presentation point of view, Natural Semantics specifications are expressed in a relational style as opposed to a functional style for Attribute Grammars.

About the relative expressive powers, Natural Semantics can deal with dynamic semantics when Attribute Grammars are limited to static semantics (type-checking, translations). On one hand, the result of an Attribute Grammar is the input abstract syntax tree decorated with attribute values; on the other hand, the result of a proof in Natural Semantics is a proof tree which is not (in the general case) isomorphic to the input abstract syntax tree, subject of the goal to prove. Moreover, the logical framework of Natural Semantics provides unification and non-determinism which are outside the scope of Attribute Grammars: most strategies for evaluation are based on a static order for the computation of all attributes and are totally deterministic. Note that unification *per se* is not always a problem: we show on our example in Section 6.1 how a least fix-point (solved with unification) expressed in a Natural Semantics specification naturally disappears in an Attribute Grammar.

From the execution point of view, unification makes a big difference between Natural Semantics and Attribute Grammars. The proof tree is built with structural induction in a single recursive pass over the original abstract syntax tree and unification propagates the remaining computations on non-instantiated variables. We distinguish two kinds of unifications during the proof process: *bottom-up* unifications due to the *tupling* of several attributes in a single sequent (as in rule (6) of Figure 5) and *left-right* unifications due to common variables in input attributes (see rule (5) in Figure 3). In the former case, according to some property on the dependency graph between attributes, propagation of unification can be implemented with attribute evaluation and the single pass over the tree can be decomposed into separate passes (see [3] for more details).

Finally, during the execution process in attribute evaluators, there is no notion of failure: the generation of an evaluator ensures that any evaluation will be successfully achieved. This is not the case with Natural Semantics.

4.2 Conditions for translation

Now, we can enumerate the conditions necessary to check whether a given Natural Semantics specification can be evaluated with attributes:

1. *no missing definition*: all variables occurring in output positions also occur in input positions;
2. *no missing rule*: there must be an applicable rule for each abstract syntax operator;
3. *determinism*: only one applicable rule for each abstract syntax operator;
4. *no constraint*: input positions are reduced to variables;
5. *no link*: no common variables between input positions;
6. *no dynamic rule*: subjects of premises are strictly subterms of the subject of the rule;

The reader can find violations of these constraints illustrated in Figure 3.
Missing definition and missing rule (conditions (1) and (2)) may lead to a failure during the proof; therefore these two conditions are mandatory for the translation. Condition (3) involves non-determinism which is not acceptable in Attribute Grammars. However, we can express fake non-determinism, a situation where more than one rule can apply (pattern-matching on the conclusion), but only one will be successfully proved (failures in the premises for others). This fake non-determinism is based on exclusive rules (as rules (cond1) and (cond2)) with a conditional constructor. [1].

[1] A similar approach for the translation from Typol to Coq is discussed in [42].

$$\frac{e \vdash T_1 : a}{e \vdash p(T_1, T_2) : \mathbf{b}} \quad (1)$$

$$\frac{e \vdash T_1 : a}{e \vdash p(T_1, T_2) : f(a)} \quad (3)$$

$$\frac{e \vdash T_2 : a}{e \vdash p(T_1, T_2) : f(a)} \quad (3')$$

$$\frac{e \vdash T_1 : \mathbf{f(a)} \qquad e \vdash T_2 : b}{e \vdash p(T_1, T_2) : b} \quad (4)$$

$$\frac{e \vdash T_1 : \mathbf{a} \qquad e \vdash T_2 : \mathbf{a}}{e \vdash p(T_1, T_2) : f(a)} \quad (5)$$

$$\frac{e \vdash T_1 : \text{true} \qquad e \vdash T_2 : e' \qquad e' \vdash \mathbf{p(T_1, T_2)} : a}{e \vdash p(T_1, T_2) : a} \quad (6)$$

Figure 3: Hopeless Typol rules for Attribute Evaluation

$$\frac{e \vdash E : \mathit{true} \qquad e \vdash T_1 : a}{e \vdash \text{cond}(E, T_1, T_2) : a} \quad (\mathit{cond}1)$$

$$\frac{e \vdash E : \mathit{false} \qquad e \vdash T_2 : a}{e \vdash \text{cond}(E, T_1, T_2) : a} \quad (\mathit{cond}2)$$

Conditions (4) and (5) involve *left-right* unifications which may also lead to a failure during the proof. However, rules (4) and (5) can be reformulated in (4') and (5') in such a way that unification is not expressed in the rule but in an auxiliary predicate *unify*.

$$\frac{e \vdash T_1 : a' \qquad \text{unify}(a', f(a)) \qquad e \vdash T_2 : b}{e \vdash p(T_1, T_2) : b} \quad (4')$$

$$\frac{e \vdash T_1 : a \qquad e \vdash T_2 : a' \qquad \text{unify}(a, a')}{e \vdash p(T_1, T_2) : f(a)} \quad (5')$$

Even though an equivalent *unify* predicate can be provided within the attribute evaluator, we still have the problem of failure management in the generated evaluator. This problem is not solved in a satisfactory manner for the moment. That is why conditions (4) and (5) are mandatory.
Finally, condition (6) will also be absolutely necessary as long as the associated attribute evaluator works on a constant abstract syntax tree.

4.3 Translation scheme

Given a Natural Semantics specification in which all the conditions presented above are verified, we can apply our translation scheme in order to generate an AG definition, and then an attribute evaluator. We briefly outline the main principles of the translation scheme:

- from judgement declarations we generate attribute declarations:
  - each position induces a kind of attribute (inherited or synthesized);
  - each position generates a new name (according to the profile of the judgement);
  - each position gives its type to the attribute;
  - the sort of the subject defines the sort the attribute is defined on.
- from each rule we generate a production rule and its semantic rules:
  - the production rule is the subject of the rule;
  - a semantic rule is generated from each output position in the rule, according to the correspondence between attribute names and positions in the rule. Notice that the relational style of Typol sequents and predicates has to be converted into a functional style.