HAL
open science

On the multiplicity of operational semantics for logic programming and their modelization by attribute grammars

Pierre Deransart

To cite this version:

Pierre Deransart. On the multiplicity of operational semantics for logic programming and their modelization by attribute grammars. [Research Report] RR-0916, INRIA. 1988. inria-00075640

HAL Id: inria-00075640

https://inria.hal.science/inria-00075640

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

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° 916

ON THE MULTIPLICITY OF OPERATIONAL SEMANTICS FOR LOGIC PROGRAMMING AND THEIR MODELIZATION BY ATTRIBUTE GRAMMARS

Programme 1

Pierre DERANSART

Octobre 1988

RR_0916

Programme 1

On the Multiplicity of Operational Semantics for Logic Programming and their Modelization by Attribute Grammars

Sur la Multiplicité des Sémantiques Opérationnelles de la Programmation en Logique et leur modélisation par des Grammaires Attribuées

Pierre DERANSART
INRIA
Rocquencourt, BP 105
F 78153 Le Chesnay
uucp deransar@minos.inria.fr

October 1988

Abstract
We show in that report how the classical theory of logic programming can be considerably simplified by using an approach based on the proof trees. We show that SLD-resolution, GLD-resolution forward chaining, N-models or other kind of operational semantics of logic programs are instances of a more general and simpler algorithm whose semantics is the same and based on the unification algorithm. It results that all the non-deterministic operational semantics are equivalent. This gives general principles to build new equivalent ones. Thus we use attribute grammars to describe some of them and show that this formalism is powerful enough to cover SLD-resolution with static computation rules, the forward chaining and the N-models. This opens for many applications: new interpreters for logic programs, foundation of new proof methods of many properties like partial correctness and completness with regard to some expected non-deterministic operational semantics or run time properties for which results of soundness and completness can be easily obtained. The report does not develop complete foundations of the proof methods since some of them are direct application of known results, but gives informal presentation of some of them.

Keywords :
Logic Programming, declarative semantics, operational semantics, attribute grammars, interpreters, proof methods, partial correctness, completness.

Résumé
Nous montrons dans ce rapport comment la théorie classique de la programmation en logique peut être considérablement simplifiée par une approche fondée sur la notion d'arbre de preuve. Nous montrons que la SLD-résolution, le chaînage avant, les N-modèles ou toute autre sorte de sémantique opérationnelle des programmes logiques sont des instances d'un algorithme plus général et plus simple basé sur l'algorithme d'unification. On en déduit que toutes ces sémantiques sont équivalentes et donnons les règles qui permettent d'en construire de nouvelles. Dans un deuxième temps, nous utilisons des grammaires attribuées pour en modéliser quelques unes, en particulier la SLD-résolution avec une règle de calcul statique, le chaînage avant et les N-modèles. Ceci ouvre des perspectives pour de nombreuses applications : nouveaux interpréteurs pour les programmes logiques, fondements de nouvelles méthodes de preuve pour diverses propriétés comme la correction partielle et la complétude relativement à une sémantique opérationnelle non-déterministe attendue ou des propriétés dynamiques pour lesquelles on pourra obtenir aisément des résultats théoriques de validité et de complétude. Ce rapport ne développe pas les fondements complets des méthodes de preuve car certaines sont l'application immédiate de résultats déjà connus, mais nous en présentons quelques unes informellement sur des exemples.

Mots clés :
Programmation en Logique, sémantique déclarative, sémantique opérationnelle, grammaires attribuées, interpréteurs, méthodes de preuve, correction partielle, complétude.

On the Multiplicity of Operational Semantics for Logic Programming and their Modelization by Attribute Grammars

Pierre DERANSART
INRIA
Rocquencourt, BP 105
F 78153 Le Chesnay
uucp deransar@minos.inria.fr

Introduction :

Logic Programs, say Horn Clause Programs, have a declarative semantics based on the notion of proof trees (or equivalently the set of logical consequences) and a non deterministic operational semantics defined by the SLD resolution based on the use of the unification to compute proof trees.

Many authors have described the SLD-resolution and shown its soundness and completness with regards to the declarative semantics, that is to say that every proof tree can be computed using the SLD-resolution. Moreover any computation rule which can be used in the definition of the SLD-resolution does not influence the results. One of the most popular presentation is in [Llo 84]. In [DF 88] an other equivalent operational semantics but presented in a different framework is given by a non deterministic algorithm computing the proof trees. In [FLM88] a "new declarative semantics" for logic programming is presented and founded, showing its soundness and completness with regards to the declarative semantics. In [DM 88] this semantics is used for the purpose of proofs of completness of logic programs.

We show in that report that all these semantics are equivalent in the sense that they define the same objects. It is based on the observation that the proof trees are obtained by a process of skeletons decoration which consists in findind the most general unifier of a set of equations. The different semantics depends only on the order in which the equations are solved. Hence all semantics are equivalent and many "new" ones may be defined for different purposes. Thus we give laws indicating the conditions these "new" semantics have to satisfy to have the same semantics. We use attibute grammars to model some of them, showing that all these attribute grammars define the same objects.

Moreover proof methods for attribute grammars as developped in [CD 88] can be used to prove properties of logic programs. In fact by transferring the proof methods for attribute grammars in each operational semantics we get different proof methods of (non deterministic) operational properties of logic programs. This approach brings many potential applications in logic programming methodology and logic programming interpretation.

The paper is basically divided into two parts. The first one (sections 1, preliminaries on logic programming, and 2, General Operational Semantics) shows how the known operational semantics are equivalent and give laws to build new equivalent ones. The second (sections 3, preliminaries on attribute grammars, 4, examples of operational semantics) show the way to use attribute grammars to modelize operational semantics used for different purposes.

The basic results of the first part of this report have been already obtained in [WML 84] in which it is shown that any kind of operational semantics will lead to the same set of answer substitutions because they all make use of the basic properties of the unification algorithm. They illustrate their presentation by analysing an operational semantics called GLD-resolution.

Our approach is slightly different. It is based on the grammatical view of logic programming [DM 88] and gives a completely declarative view of the various operational semantics by looking at them as a syntax tree decoration process. Thus we obtain new results : laws to build new equivalent ones (this simplifies the proof of soundness and completness of new ones) and new applications.

1. Preliminaries on Logic Programming.

A logic programm P=<**Pred, Func, Claus**> is defined as usual [see Cla 79, Llo 84] by predicate and function symbols and Horn Clauses. The (non necessarily ground) terms are defined with **Func** and **Var**, a denumarable set of variables. We denote a clause **c** of **Claus**:

$$a_0 \Leftarrow a_1, \ldots, a_k, \ldots, a_n \qquad 1 \leq k \leq n$$

in which for $k>0$ a_k is $p_k\, T_k$ and T_k is $(t_{k1}, \ldots, t_{kj}, \ldots, t_{kq})$, $q \geq 0$.

We denote by **(P, g)** an augmented program <**Pred** U {**pt**}, **Func**, **Claus** U {**pt** ⇐ **g**}> which is **P** with a goal clause whose head is **pt** and the body is **g**. If the goal has the form of an atom whose arguments are distinct variables and predicate symbol is **p** we call it a most general goal for **p**.

To a program it is possible to associate a context free grammar **GP=<Pred, Rule>** where every rule **rc** in **Rule** corresponds to the clause

c in **Claus** and reciprocally: if **c** is $p_0T_0 \Leftarrow ..., p_kT_k, ...$ then r_c is $p_0 \rightarrow ..., p_k,...$ That is to say r_c is **c** in which all arguments have been rubbed. We call skeletons of **P** the *finite syntax trees* of **GP**. The nodes of a syntax tree, as for other kind of trees which will be considered, will be numbered a la dewey, such that if a parent node is **u**, its sons have number **u1 ... uk** to **un**. The root is node **0**, the other node numbers start with a positive number.

In what follows renaming of variables, terms clauses or trees containing variables will be necessary. It will be done by indexing the variables by tree node numbers. Thus a renamed instance of a variable, a clause, an atom, a list of arguments or an argument will be denoted by $X(u)$, $c(u)$, $a_k(u)$, $T_k(u)$, or $t_{kj}(u)$ respectively.

Following Clark [Cla 79] one can introduce the notion of proof tree. Informally speaking a proof tree is a result of pasting together instances of the clauses of **P** as stated more precisely by the following definition.

Definition 1.1.

A proof tree of **P** is any ordered labeled tree satisfying the following conditions :

1. If a node of the proof tree is labeled **a** and the nodes of its direct descendants are labeled $a_1, ..., a_k, ..., a_n$ (where the indices reflect the ordering) then $a \Leftarrow a_1, ..., a_k, ..., a_n$ is an instance of a clause of **P**.
2. All its leaf nodes are instances of a unit clause (fact) of **P**.

We will call pre proof tree a skeleton **Skel** whose nodes **u** are labeled by the renamed head $a_0(u)$ of the clause used at the root of the subtree issued from **u**. It will be denoted **Pre(Skel)**.

In the sequel we will not distinguish between the trees (terms or proof trees) which are identical up to a renaming of the variables in their labels. (Thus, as a matter of fact we deal with equivalence classes of trees and labels).

The concept of proof tree can be used as a basis for a model-theoretic semantics of a logic program. Denote by **DEN(P)** the set of the root labels of all complete proof trees of **P**. According to the convention above **DEN(P)** is determined up to the renaming. It has been proved in [Cla 79] that **DEN(P)** is the set of all (not-necessarily ground) atomic logical consequences of **P**. This is connected with the declarative reading of **P** and can be used for defining the semantics of **P** as **DEN(P)**.

The concept of proof tree is purely declarative. The question arises how to construct proof trees. In particular, for given program **P** and goal **g** it may be interesting to find the set **T** of all proof trees **t** such that the root of **t** is an instance of **g**. The problem can be solved by using the SLD-resolution. Each refutation for **P** and **g** via some computation rule corresponds to some proof tree. Now **T** consists of all instances of all such trees. The well-known result on the independence of the computation rule [Llo 84] shows that the set of proof trees corresponding to all refutations of a given goal is the same regardless of the computation rule. We will show that the same results holds for a more general non deterministic algorithm using only the unification algorithm (without any computation rule), such that any kind of computation rule appears to be a way to introduce determinism in the general algorithm without affecting his properties.

For this purpose we need some additional concept.

construction 1.2 : set of equations associated to a skeleton.

To every skeleton **Skel** of an augmented program we associate a set of equations denoted <u>**Eq(Skel)**</u>. and obtained as follows :

Every node **uk** in **Skel** is the root of a subtree (it can be reduced to its root) which is built with the rule r_c' coming from the claus $c': a_0' \Leftarrow ...$, and a parent node **u** developped with rule r_c coming from the clause $a_0 \Leftarrow a_1, ..., a_k, ..., a_n$. (the root of **Skel** only does not have parent, thus it will not produce any equation). Note that a clause is renamed using the node number of its root.

Thus **Eq(Skel)** is defined by :

Eq(Skel) = ($t_{kj}(u) = t_{0j}(uk)$) for all **u** different of **root**, **k**, **j** in **Skel** and **Claus)**

We will denote by <u>MGU(Eq)</u> the *most general unifier* of a set of equations which is a substitution which when applied to all equations makes their two terms identical .

We will use known properties of the MGU [Rob 65, Hue 77, LMM 86] : Every unifier is an instance of the MGU which is unique up to a renaming. Moreover the MGU does not depend on the way to solve the equations, i.e. it does not depend on the order the equations are considered to be unified, provided the unifier already obtained to solve a subset of equations is applied to both members of a new equation before unifying it (see law 2.6).

Finally we will make use of the canonical termal interpretation, in which values are terms (possibly with variables), assignments are substitutions and equality of terms, their identity.

2. General Operational Semantics GOS(P).

We define now a general (non deterministic) operational semantics which serves as basis for all known deterministc or non deterministic semantics that we prove sound and complete. Thus we give the conditions to be satisfied such that any new operational semantics which can be derived from the general one keeps these properties.

Algorithm 2.1 : GOSA

Given an augmented program **(P,g)**

For all skeleton **Skel** of root **pt**

Sig = MGU (Eq(Skel))

The resulting proof tree is **Sig(Pre(Skel))**

The resulting answer substitution is **Sig** restricted to the variable of g.

Note that the proof trees and the answer substitutions obtained by the GOSA are defined up to a renaming of the variables.

Definition 2.2 : GOS(P), GOSPT(P)

The General Operational Semantics of a logic program **P** is the set of all the goal instances by the answer substitutions resulting of GOSA applied to the most general goals. The set of the resulting proof trees will be called **GOSPT(P)**.

Lemma 2.3

a) Given **(P,g)** and a skeleton **Skel** of root **pt**, if **Eq(Skel)** has a solution **Sig** then **Sig(Pre(Skel))** is a proof tree of **(P,g)**.

b) If it exists a proof tree in **DEN(P)** of root **g** and of skeleton **Skel**, **Eq(Skel)** obtained with the augmented program **(P,g)** has a solution.

c) Given **(P,g)**, **g** atomic, and a skeleton **Skel** for which **Eq(Skel)** has a solution **Sig**, then **Sig(g)** is an instance of **Sig'(mg)** where **mg** is the most general goal corresponding to the predicate symbol of **g**

and **Sig'** the solution with the same skeleton (**Skel** of **(P,g)** with **g** replaced by **mg,** i.e. same **Skel** but of **(P, mg)**).

Proof of lemma 2.3

First of all note that a proof tree is built with clauses instances. It is equivalent to say that a proof tree is built with instances of renamed clauses, called **c(u)**. Thus we will make use of the renaming described in construction 1.2.

a) if **Eq(Skel)** has a solution **Sig**, then grouping the equations associated to the same node **u** we get **Sig(** a_k**(u)) = Sig(** a_0**(uk))** and thus to **Skel** it can be associated the proof tree built with instances **Sig(c(u))** of clauses. Hence **Sig(Pre(Skel))** is a proof tree of **(P,g)**.

b) If it exists a proof tree with skeleton **Skel**, thus it exists instances of clauses **c(u)** by the substitutions, say $\mathbf{Sig_u}$, such that at every node **uk** of **Skel** there is $\mathbf{Sig_u}$**(** a_k**(u)) =** $\mathbf{Sig_{uk}}$**(**a_0**(uk))**; this means both members of the equations have a common instance.

Remark that all substitutions $\mathbf{Sig_u}$ have disjoint domains since the variables are renamed in every clauses instance. Moreover no variable in the terms of the codomain of the $\mathbf{Sig_u}$'s is in the domain of any $\mathbf{Sig_u}$ (by definition of the instance this is possible for $\mathbf{Sig_u}$ itself -by an appropriate renaming- and is obvious with regards to the others since the $\mathbf{Sig_u}$'s are defined independently).

Thus call **Sig** the union of all $\mathbf{Sig_u}$'s.(which is by the way identical to the composition of the $\mathbf{Sig_u}$'s) We get obviously that all equations are equal by **Sig**.

c) Consider the three augmented programs with the same skeletons with root **pt** and the corresponding solutions :

(1) pt ⇐ p T **Sig**

(2) pt ⇐ p X , X = T **Sig** where **p X** is the most general goal of **p T** and **X** are variables different from those of **T**. Obviously the solution is the same (up to a renaming).

(3) pt ⇐ p X **Sig'**

Solving the last equation in (2) gives **(Sig'(X)) = (Sig'(T))**, let **Sag** be the MGU of this last equation. As the solution of the whole system is **Sig**, we get : **Sig = Sag Sig'** (modulo a renaming). QED.

Corollary 2.4

a) Every proof tree of skeleton **Skel** is an instance of the proof tree with the same skeleton in **GOSPT(P)**.

b) Every answer substitution for an atomic goal **g** can be obtained by unification of **g** with elements of **GOS(P)**, after appropriate renaming of the variable of **g**.

Proofs are trivial.

Theorem 2.5 (soundness and completness of GOSA) :

a) **GOS(P)** is included in **DEN(P)**.

b) Each element of **DEN(P)** is an instance of an element of **GOS(P)**.

Proof : immediate by definition 2.2 and lemma 2.3a and by corollary 2.4 a.

Consequences :

The main consequence of the last theorem, informally speaking, is the independence of the results with regards to any computation rule. in an algorithm with most general unification process. Thus the results of [Llo 84, DF 87 or FLM 88] appears to be trivial consequences of this theorem, turning every theorem on soundness and completness of the specific algorithm as a remake of the proof the unicity of results of some unification algorithm.

In fact most of the known algorithms computing proof trees define a way to introduce determinism in the GOSA. Basically one possibility is to fix an order on the equations, one other is to select one equation after the other dynamically or subsets of them, applying the composition of the previously obtained substitutions to the members of the new selected equations. This is exactly done in SLD-resolution : a skeleton is built progressively, the node to be expanded beeing choosen dynamically. If the process succeeds, then the order of visiting the nodes is a total order giving the order in which the equations have been solved. The only difference comes from the way to apply the composition of the substitutions: it is implicitely applied to the left member (indice **u**), but it not necessary to apply the composition to the right member since indice **uk** cannot appear in the variables already used (see 4.1).

Remark : also that the theorem of the independence of the computation rule in the SLD-resolution [Llo 84] is also a simple consequence of lemma 2.3. In fact to every success branch in a search tree it corresponds a unique skeleton, thus a unique (up to a renaming) answer substitution. Thus all sets of computed answer substitutions in all search trees corresponding to different computation rules are the same.

So it appears that SLD-resolution is only one (but one of the most popular) way to define operational semantics and **GOS(P)**. It is thus more interesting to define only some principles which guarantee that a new algorithm is sound and complete with regard to **DEN(P)** such that it computes the same set **GOS(P)**. We call these principles a "law of semantic conformance" as it defines properties of many algorithms or implementations which are known in the logic programming folklore or which are new ones.

Law 2.6 (law of semantics conformance) :

The following algorithms are sound and complete with regard to **DEN(P)** and compute **GOS(P)** if applied to the most general goals :

Start with the empty current substitution.

Build all skeletons of **(P,g)** of root **pt** and solve **Eq(Skel)** choosing a subset (note that the subset may be reduced to one equation) of the equations applying them the current compound substitution before unifying them; the final substitution, if any, is the final compound substitution.

Proof :

It is sufficient to show that this non deterministic algorithm computes the most general unifier (up to a renaming), and that reciprocally if it exists an unifier of the set of equations, the algorithm will succeed. The proof is easy considering the properties of the MGU at every step. QED.

These principles open for many kind of different operational semantics with different purposes. One of them could be to compute answer substitution differently, for example by saturation (forward chaining) or acting at the level of the arguments, by computing partially the atoms in the proof trees. This last approach will be illustrated with a different purpose in the next section: using a specific operational semantics to make a proof of completness of a logic program.

The last two sections will be devoted to show on examples how algorithms using a static order can be easely described by attribute

grammars. The interest of attribute grammars is to bring a conceptual framework using the rule structure of logic programs to describe the strategies and susceptible of many applications.

3 Preliminaries on Attribute Grammars.

This section recalls some results on Attribute grammars which are strictly necessary to understand section 4. We use the framework of [CD 87 and DM 85] introducing Functional Attribute Grammars (FAG), but more, but less formal, details can be found in [DJL 88].

Definition 3.1

A Functional Attribute Grammar (RAG) is a 5-tuple

<N, R, Attr, Phi, Inter>

where

N, **R** is a Context Free Grammar.

Thus **N** and **R** give rise to a set of parse trees and the other elements of FAG provide a labeling formalism.

Attr is a finite set of attribute names. Every nonterminal of **N** has associated a set of attribute names. To simplify the presentation it is assumed that these sets are disjoint and linearly ordered. Thus the attributes associated with a nonterminal **x** will be denoted **x1, ..., xq**, where **xi** denotes the ith attribute of **x** in the ordering. In some example, for sake of understandability, same name will be given to different attributes attached to different nodes. As usual, the labeling mechanism will be specified at the level of grammatical rules. Different occurences of the same non terminal in a rule will lead to different occurences of the same attribute. They will be denoted by additional indices referring to the occurrence number of the nonterminal. For example consider the rule

x -> x x

where **x** has two attributes **x1** and **x2**. The different occurrences of the attribute **x1** will be denoted : **x1(0)**, **x1(1)** and **x1(2)**. As usually attributes are splited into two disjoint subsets: the sets of inherited and synthesized attributes, i.e.:

Attr = Inh U Syn

Thus to every non terminal **x** are associated two subsets of attributes denoted **Inh(x)** and **Syn(x)**. For simplicity of the presentation of examples we will consider that inherited attributes are before synthesized ones in the order of the attributes.

Phi is an assignment of a logic formula $\mathbf{Phi_r}$ to each production rule **r** in **R.** The formula includes attribute names as the variables. Thus for every interpretation **Inter** defines some relation between the values of the attribute occurrences in the rule **r**. In FAG's the attribute definitions $\mathbf{Phi_r}$ have the following functional form :

- there is exactly one equational definition for every instance of the synthesized attributes of the root and the inherited attributes of the right hand side of the grammatical rule **r**
- each equational definition has the form :

xk(u) = somefunction(..., xl(v), ...)

where the **xl(v)** are instances of inherited attributes of the root and synthesized attributes of the right hand side.

- $\mathbf{Phi_r}$ is the logical conjunction of all equalities in **r**.

Inter is the interpretation of the language used in **Phi** : it defines the domain, associates functors of the language with functions on the domain and predicate letters with relations. In the sequel we will consider that **Inter** is the canonical termal interpretation as defined in section 1.

The formalism defines a set of labeled trees. Each of them has the context free skeleton defined by the production rules of **R**. A node of the skeleton with nonterminal **x** is to be additionally labeled by a **q**-tuple of values in the domain of **Inter**, where **q** is the number of attributes of **x** .

The tree consists of instances of the production rules. Let **r** be one of them. Then the nodes of **r** have to be decorated by the tuples of values. Each of the values corresponds to some attribute appearing in the formula $\mathbf{Phi_r}$. It is required that the formula is true in **Inter** with this valuation.

A decorated parse tree such that this condition is satisfied for all its component instances of the production rules is called valid decorated tree of the FAG. Thus, the semantics of a FAG is the set of the valid decorated trees or in short valid trees. For a more complete treatment of this definition see [CD 87] or [DM 85].

Example 3.1 : Functional Attribute Grammar for symbolic factorial

The following RAG describes the computation of i ! the factorial of i (i.e: 0! = 1 and for i > 0, i !=(i -1)!* i). A systematic presentation of such constructions is given in [CD 88].

N is	{ fact },
R is	{ 1 : fact -> e , 2 : fact -> fact }, (e denotes the empty string)
Attr is	{ fact1, fact2 }, where fact1 denotes the argument and fact2 the results.
Phi$_1$ is	fact1(0) = zero and fact2(0) = one ,
Phi$_2$ is	fact1(1)=(if fact1(0) = succ(X) then X) and (fact2(0) = fact2(1)*fact1(0))
Inter is	The standard model of Nat, the natural (or non negative) integers using the usual funtors zero and succ(essor).

A valid decorated tree for the given RAG is shown below.

```
fact succ(succ(zero)), (one*(succ(zero)))*succ(succ(zero))
 |
fact succ(zero), one*succ(zero)
 |
fact zero, one
```

It is easy to see that every subtree of a valid tree is valid. Thus, the concept of valid tree makes it possible to associate a relation on the domain of **Inter** with every nonterminal **x** of the grammar. The relation is **q**-ary, where **q** is the number of attributes of **x**. A **q** -tuple of attribute values is in the relation iff it is a root label of some valid proof tree for **x**.

It is easy to see that the relation associated with the nonterminal fac is the set of pairs :

succ**n(zero),*((...(one*succ(zero))*...*succ**n-1(zero)) succ**n(zero))...)

$n \geq 0$

An important notion in attribute grammars is the notion of dependency between attribute instances inside a syntax tree. This dependency relation states which attributes instances have to be known in order to compute some attribute instance. It is defined by the local dependency relation **D(r)** defined for each rule r as :

a D(r) b iff attribute occurrence **a** appears in the definition of **b**

Thus in every syntax tree **t** the instances of the local dependency relations lead to a global dependency relation denoted **D(t)**. The relationships between local and global dependency relations have been extensively studied. We recall here some useful results only.

Definition 3.2

A FAG is <u>non circular</u> iff in every syntax tree **t** the global dependency relation is a partial order.

Of special interest is the fact that a FAG is non circular iff in every syntax tree it exists a *total evaluation order* of the attribute instances. Thus if a FAG is non circular there exists a deterministic algorithm (provided the syntax tree is given) to compute elements of the semantics of the FAG.
In order to simplify the remaining part of this paper we will restrict our attention to subcategories of FAG's.

Definition 3.3

A FAG is <u>purely synthesized</u> (<u>inherited</u>) iff there is no inherited (synthesized) attribute.

A FAG is <u>l-ordered</u> iff in every rule $D(r_c)$ is a total order compatible with an evaluation order in all syntax trees. In the sequel we will consider subclass of l-ordered FAG's, called <u>L-AG</u> in which the evaluation of the attibute instances can be performed during a descendent tree construction.

Properties 3.4

The non circularity of a FAG is decidable.

Every FAG which is l-ordered, L-AG, purely synthesized or inherited is non circular. In the case of a L-AG the total local order is compatible with the order given in the definition 3.2.

In the sequel we will modelize non deterministic operational (some time deterministic also) semantics of logic programs by attribute grammars. As a consequence it will be possible to make use of the proof methods (correctness and completness with regard to a specification) developed in [CD 88] for the correctness without more explanation. In the short examples presented here inductive proof method will be sufficient. Thus we recall here its definition.

Let us consider without more formalism (see [CD 88 p. 29] for more details) a **specification** to be a family logical formulas indexed by the non terminals and whose free variables are attributes of the corresponding non terminal. Thus we say that a specification is valid iff for all **q**-uples of values in the semantics of the attribute grammar associated to the non terminal **x**, the specification S^x holds, in which all attribute variables have been replaced by their corresponding value in the **q**-uple. That is to say that every **q**-uple in the semantics of the attribute grammar satisfies the specification.

Inductive proof method will be used to prove the validity of a specification. It is stated as follows in [CD 88].

Definition 3.5

A specification $\{S^x\}$ $\mathbf{x} \in \mathbf{N}$ is inductive iff for all **r** in **R** the following formula is valid :

$$Phi_r \text{ and } S^{x1} \text{ and } \ldots \text{ and } S^{xn} \Rightarrow S^{x0}$$

Theorem 3.6 [CD 88, th 3.2.5]

A specification is valid iff it is weaker than some inductive specification.

Coming back to logic programming, we will specify GOS(P) by an attribute grammar whose semantics is exactly GOS(P). Thus the validity of a specification means that all elements of GOS(P) satisfy the specification and the proof method will make use of inductive specifications as in definition 3.5. Note that we will also extend the method of [CD 88] to prove the completness of an attribute grammar with regards to some specification. In other words, we will prove that some specified atoms are in GOS(P).

4 Examples of operational semantics.

As illustration of the law of semantics conformance we consider three examples of FAG's with partial or total local orders : a L-AG modeling the SLD resolution with static computation rule, a purely synthesized FAG modeling the semantics described in [FLM88] and a familly of FAG's, one of them beeing used for the purpose of the proof of completness of a little program.

All the FAG's are built following the same basic elements. They will differ essentially by the spliting of the attributes (the way to group the arguments in at most one inherited and one synthesized attribute) and the way to describe the compound substitutions by one or two attributes.

Let P = < **Pred, Func, Claus** > be a Logic Program.

We associate with it a FAG

G = < **Pred, Rule Attr, Phi, Inter** >

where

Rule is as in definition 1.3.

Attr is a set of attribute names denoting the arguments of the predicates (the **i** th argument of the predicate **p** will be denoted **pi,** or all arguments will be denoted **Arg** if they are taken all together), and the substitutions.

Phi will be defined below. For each different semantics of **P** a separate definition will be given.

Inter is the canonical termal interpretation.

4.1 SLD-resolution with static computation rule: AG 4.1

By static computation rule we mean that atoms in the body of a clause are always chosen in the same order. Without lost of generality the left to right order can be chosen.

Attr is defined by **Inh** = { **hSig, Arg** } and **Syn** = { **sSig** }. We use the same name for attributes associated to different non terminal in **Pred**. **hSig** denotes the compound substitution used to instantiate a new equation, **sSig** is the compound substitution after unifying all the equations in the subtrees in which equations are instanciated by **hSig**. **Arg** is inherited as the computation is top-down.

To define **Phi** we consider two cases: the goal rule and the others. No description of the renaming process is done: dewey numbering can be described explicitely by a L-AG. Hence tree node numbers are used in the attribute definitions.

Goal rule (we consider only atomic goals like **g** = **p T**): **pt -> p**

hSig(1) = emptysubst
Arg(1) = T(0)

Other rules : p_0 -> p_1, ..., p_k, ..., p_n

let Sig = MGU(**Arg(0)**, T0(u))
Arg(1) = Sig T1(u)
hSig(1) = Sig **hSig(0)**
...
k>0 **Arg(k)** = **sSig(k-1)** Tk(u)
hSig(k) = **sSig(k-1)**
...
sSig(0) = **sSig(n)** if **n>0**
= **Sig hSig(0)** if **n=0**.

Attribute occurrences are written in boldface, renaming is not described, it could be also by two attributes of type integer following the attributes h(s)Sig.
AG 4.1 is a L-AG.

Lemma 4.1.1

AG 4.1 satisfies the conditions of the law 2.6

Proof :

Consider the L-AG evaluation order. It corresponds to a depth first left to right skeleton walk. At every step the subset of equations corresponding to the labels of a node is choosen and the current substitution hSig is applied before unifying it (as MGU(**Arg(0),T0(u)**) = MGU(**hSig(0) Arg(0), hSig(0)T_0(u)**). In fact **hSig(0)** does not have common variables with **T_0(u)** since equations are taken from new subtrees)and updated. To verify that **hSig** is the current substitution We show that the following specification is inductive :

"**sSig(x)** is **hSig(x)** composed with the sequence of the local **Sig** substitutions computed in the whole subtree issued from **x**."

It remains to observe that the current substitution is initialized to the empty substitution to get the conditions of law 2.6. QED.

As a byproduct of the proof we may observe that **sSig(1)** in the decorated tree is the final compound substitution, hence **sSig(1)Pre(Skel))** is the corresponding proof tree.

Lemma 4.1.2

Every partially decorated tree of AG 4.1 until the **n**th visited node **u** corresponds to **n** steps of SLD-resolution with the corresponding static computation rule.

To perform the proof we give a simplified definition of the SLD resolution which corresponds to that of [Llo 84], but with a static computation rule and a fixed choice of the clauses, since we can assume that every path in the SLD tree corresponds to a partial skeleton constructive walk. Without loss of generality we may consider the standard conputation rule wich consists in choosing the atoms in the body of a clause from left to right. It corresponds to a skeleton walk construction in a depth first and left to right manner. Renaming is not described explicitely; it is assumed that new clauses are renamed with fresh variables. As it is not constrained, we will choose the same renaming as in the skeletons.

Definition 4.1.3 (SLD resolution with standard computation rule)

let **$teta_i$** be the *current compound substitution* , **g** a *goal* and **G_i** the *current resolvant*:

step 1 **teta$_1$** is emptysubst, **G$_1$** is **g**.

assume **G$_i$** is **g$_1$.G**

step i+1 let **teta** = MGU(**g$_1$,a**) where **a<= b$_1$, ..., b$_n$** is the renamed choosen clause.

G$_{i+1}$ = teta b1. ..., teta bn. teta G and **teta$_{i+1}$ = teta teta$_n$**

stop with success iff **G$_i$** = emptyresolvent, thus **teta** is the *final compound substitution* and its restriction to the variables of the goal the *answer substitution.*

Proof : of lemna 4.1.2

As in lemna 4.1.1 we consider the L-AG evaluation order and prove the lemna by induction on that order, i.e. the following property is invariant:

let **i** be the currently visited node **u** in the partial skeleton **Skel** corresponding to the **ith** step of the SLD resolution, then

hSig(u) = teta$_n$

G$_i$ = hSig(u) (list of the leaves labels pv Arg(v) for all **v** after **u** in **Skel** following the same order)

step 1 OK: **hSig(1)** = emtysubst = **teta$_1$**

G$_1$ = g

assume step i: **hSig(u) = teta$_i$, G$_i$ = Arg(u). hSig(u) G** since **hSig(u) Arg(u) = Arg(u)**.

step i+1: (unification) there are two cases depending on the existence of a body or not in the renamed choosen clause **a(u) <= b1(u), ..., b$_q$(u).**

case q>0 : let **Sig** = MGU(**Arg(u),T$_0$(u)**)

hSig(i+1) = hSig(u1) = Sig hSig(u) = teta$_{i+1}$ = teta teta$_i$ with **teta = Sig**.

G$_{i+1}$ should be, by definition of the SLD resolution: **Sig (T$_1$(u),..., T$_q$(u). hSig(u) G)**, which is exactly **hSig(u1) (T1(u), ..., T$_q$(u).G)**, since **hSig(u)** applied to the **T$_k$(u)** leaves them invariant.

case q=0 : In that case **Sig** is also MGU(**Arg(u),T$_0$(u)**), but the following node **i+1** is a node **u'k+1** on the slice of **Skel** leave of some upper rule of root **u'** such that **u** is in the subtree issued from **u'k**. If not, this means that **Skel** is complete and the SLD resolution successful.

If **Skel** is incomplete, by definitions we have : **G$_i$ = Arg(u). hSig(u) G** and **G$_{i+1}$ = hSig(u'k+1) G**, but by the attribute definitions and the definition of **u'k+1**, **hSig(u'k+1) = Sig hSig(u)** (case **q=0**), thus **G$_{i+1}$ = teta G$_i$** which corresponds to one step of resolution in case of deletion of a goal in the resolvant (**q=0**).

If **Skel** is complete thus **G$_{i+1}$** is empty and, by lemna 4.1.1 **teta$_{i+1}$** is **sSig(1)**. QED.

Theorem 4.1.4

The semantics of AG 4.1 is the semantics of the SLD-resolution with static computation rule corresponding to the order of the atoms in the body of **P**. Both compute the same set of answer sustitutions for a given goal. This set is **GOS(P)** with the most general goals.

Proof : immediate consequence of lemnas 4.1.1 and 4.1.2 since the semantics of AG 4.1 is **GOSPT(P)** (by lemna 4.1.1 it satisfies law 2.6) and modelizes also the SLD resolution (by lemna 4.1.2 every valid tree in AG 4.1 corresponds to a successful SLD resolution of the same skeleton).

Remark : Theorem 4.1.4 shows that SLD resolution implements algorithm 2.1 and satisfies law 2.6. Nevertheless a direct and much simpler proof showing that SLD resolution (with any computation rule) satisfies law 2.6 could be performed such that direct and simple proof of the independence of the results of the SLD resolution with regards to the computation rules can be obtained without sophisticated lemnas like the switching lemna in [Llo 84]. The purpose of AG 4.1 is to show that SLD resolution with static computation rule can easely be modelized by an attribute grammar, but it has also some applications which are not developped herein. For example it can be used to get results of soundness and completness of proof methods of different kind of logic programs properties like "run time properties".

4.2 Forward chaining : AG 4.2

We model computation of **GOS(P)** by a simple purely synthesized FAG which is equivalent to computations by saturation or forward chaining or, by definition, to the least N-model in [FLM 88].

Attr is **Inh** = { **Sig** }, **Syn** = { **Arg** }

Phi is : case **pt -> p**

Sig(1) = emptysubst if most general goal is considered or
= MGU(T(0), Arg(1)) if goal p T .

case $\mathbf{p_0}$ **-> p1, ..., pk, ..., $\mathbf{p_n}$**

Let Sig = MGU((T_1(u), ..., T_n(u)), (**Arg(1), ..., Arg(n)**))

Arg(0) = Sig T_0(u)

k>0 Sig(k) = **Sig(0)** Sig

if **n = 0** :

Arg(0) = T_0(u)

First observe that the attribute **Sig(u)** represents the composition of the **Sig's** in a bottom up manner.

Consider now the AG without **Sig** definitions, i.e. with one synthesized attribute only.

Theorem 4.2.1

The AG 4.2 (with or without **Sig's** definitions) satisfies the conditions of the law 2.6

Proof: Consider a node **u** in **Skel**. In all subtrees of **u** the equations can be solved independently since they have no common variables. A grouping can by given by a L-AG order since the AG is purely synthesized and the solution of a subset of equations issued from a subtree does not share any variable with the one of a brother subtree. Thus the "current" substitution may be applied to a subtree without changing the corresponding equations (the equations selection order is: solve the equations in the subtree from left to right, thus solve the equations of the root) and AG 4.2 computes the solution of **Eq(Skel)**, since it starts with no -i.e. empty- substitution. QED.

As a consequence all the roots of valid trees of this AG are elements of **GOS(P)**.

Consider now the **Sig**'s definitions (purely inherired).

Proposition 4.2.2

The labels **Sig(u) (pu Arg(u))** associated to every node **u** in **Skel** are proof tree labels of the proof tree of root **Sig(1) (p1 Arg(1))**.

Proof : immediate by observing that, by attribute definitions, **Sig(u) Arg(u)** $\Leftarrow$ **..., Sig(uk) Arg(uk), ...** is a clause instance by **Sig(u) Sig**..since by definition **Arg(u) = Sig T_0(u)** and **Sig(uk) = Sig(u) Sig**.

It should be noted that valid trees in AG 4.2 with Arg's definitions only (as the trees in the N-models) are not proof trees, but the corresponding proof tree can be obtained very easely by applying the composition of the substitutions along a path only, composing them in a bottom up way.

We illustrate the use of the AG 4.2 to make a proof of (total) correctness of a simple "spagetti" meta interpreter (uppercase letters denote variables).

SOLVE:

1- solve(emptybody) <-.
2- solve(A **and** B) <- solve(A), solve(B).
3- solve(A) <- atom(A), clause(A,B), solve(B).

It is assumed that **atom(A)** specifies all possible atoms, and **clause(A,B)** specifies all clauses of a program **P**.

It is important to remark that the proof of correctness with regards to the declarative semantics of such interpreter is staightforward as proof trees in **SOLVE** are built with clauses instances, like proof trees in **P**, hence the result that proof trees of SOLVE contain conjunction of proof trees of **P**. But we want to prove a stronger property, i.e. correctness with regards to the (non deterministic) operational semantics, i.e. that same holds if we consider **GOSPT** in place of proof trees. That is to say both programs (P and **SOLVE** for **P**) will compute the same set of answer substitutions for the most general goals (up to a renaming).

Thus we want to prove that **SOLVE** is correct and complete in the sense that all elements of **GOS(SOLVE)** are (eventually empty) conjunction of elements of **GOS(P)** and reciprocally. As we can observe that the proof trees of **SOLVE** are built with instances of variables of **SOLVE** and of **P**, we can restrict our attention to elements of **GOS(SOLVE)** whose renamed variables are renamed variables of **P**. Thus we show that to every (eventually empty) conjunction of elements of **GOS(P)**, it corresponds one element of **GOS(SOLVE)** using only renamed variables of **P**, and reciprocally. It will follows that every conjunction of proof tree roots of **P** will be in the proof tree roots of **SOLVE** and reciprocally (if appropriate renaming is choosen).

The demonstration uses two AG 4.2's, one for **SOLVE**, the other for **P**. Thus we will make use of the particularity of AG 4.2 that every root of a valid tree is in **GOS**.

The demonstration is immediate: The tree cases corresponds to the observation that any conjunction of goals can be empty, limited to one atom or a conjunction of two goals.

<u>clause 1</u>: trivial

<u>clause 2</u>: **Sig** = MGU(**(A(u),B(u))**, **(Arg(1), Arg(2)))** and **Arg(0)** = **Arg(1) and Arg(2)**, as **A(u)** and **B(u)** are pure (renamed) variables. If we suppose that **Arg(1)** and **Arg(2)** have variables of **P**

only, thus **Arg(0)** has. The converse (existence of the MGU **Sig**) holds for a given binary conjunction of conjunction of elements of **GOS(P)** as argument of **SOLVE**.

clause 3: **Sig** = MGU((**A(u)**, (**A(u)**, **B(u)**),**B(u)**), (**Arg(1)**,**Arg(2)**, **Arg(3)**)). Suppose **Arg(2)** = (**head**, **body**), thus **Sig** is also a MGU of **body** and **Arg(3)**, and of **head** and **Arg(1)** which by definition of **atom** can be considered as an identity or a renaming (in order not to change the name of the variables of P, we choose identity). Hence **Arg(0)** corresponds to the instance by **Sig** of **head**. Hence **Arg(0)** is also the root of a tree obtained in an AG 4.2 built for program **P** (**Sig** restricted to the variables of **P** is exactely the **Sig** used in AG 4.2 of **P**). The converse is obviously true since if **Arg(0)** is an element of **GOS(P)** then a proof tree root, instance of a (renamed) clause whose head instance is this root will always exist, hence **Sig** such that **Arg(3)** is in **GOS(P)**, hence **solve Arg(3)** in **GOS(SOLVE)**.QED.

4.3 Modeling computations in a given program.

Consider the following program using difference lists (uppercase letters denote variables) :

REV:
1-rev(nil, L-L).
2-rev(A.L, R) ⇐ rev(L, Q), concd(Q, A.M-M, R).
3-concd(L1-L2, L2-L3, L1-L3).

It defines the reverse rev2 of a list rev1 with a linear complexity.

Using AG 4.2 on that program leads to the observation that the queue of **rev2** is a variable which does not appear in the first argument **rev1**. (Property satisfied by the elements of **GOS(REV)**).

This property is inductive, thus valid.
- obvious in clause 1.
- suppose in clause 2 that **rev2(u1)** = **t-V**, by attribute definition in AG 4.2 we have to unify:

L(u)	**Q(u)**	**Q(u)**	**A(u).M(u)-M(u)**	**R(u)**	
rev1(u1)	**t-V(u1)**	**L1-L2(u2)**	**L2(u2)**	**-L3(u2)**	**L1-L3**

(u2) and **V(u1)** is a pure variable which does not depends on **rev1(u1)**. Thus the results as **L3(u2)** is **M(u)** which is a variable since the unification of **V(u1)** and **A(u) M(u)** does not affect **M(u)**.

The problem of completness of a program logic is the question of *existence of at least one proof tree* for some given atoms. It seems of practical interest to formulate this problem by giving instances of some arguments when the others are unknown (existential completness in [DM 88]). The question thus arises whether it exists an instance of the others such that the corresponding goal instance is a proof tree root. For example we may ask for existence of proof tree for goals like **renv(A1.A2.....An.nil, R)**, n>=0.

The way to solve this problem consists in defining an attribute grammar in which this question may be easely solved. Thus we build a FAG following the law 2.6, AG 4.3, and show its completness with regards to the given goals.

In order to make this proof easier we split the arguments of **rev** into two sets of attributes (singletons) but keep the arguments of **concd** synthesized.

Inh = {**rev1**, **hSig**} **Syn** = {**rev2**, **sSig**, **Arg**} , **Sig** attributes are associated to **rev** only, when **Arg** is to **concd** only.

```
Phi: rule:  pt -> rev
                rev1(1) = A1.A2.....An.nil
                hSig(1) = emptysubst

     rule:  rev -> e
                Sig1 = MGU( rev1(0), nil)
                rev2(0) = L(u)-L(u)
                sSig(0) = Sig1 hSig(0)

     rule:  rev -> rev
            Sig1 = MGU( rev1(0), A(u).L(u))
                rev1(1) = Sig1 hSig(0) L(u)          (= Sig1 L(u) )
                hSig(1) = Sig1 hSig(1)
            Sig2 = MGU((rev2(1), Arg(2)) , (Q(u), (Q(u), A(u).M(u)-M(u),
                                                          R(u) )))
                rev2(0) = Sig2 sSig(1) R(u)          (= Sig2 R(u) )
                sSig(0) = Sig2 sSig(1)

     rule:  concd -> e    (we follow here AG 4.2 definitions style,
                                                     case n = 0)
            Arg(0) = (L1(u)-L2(u), L2(u)-L3(u), L1(u)-L3(u))
```

It is not difficult to be convinced that AG 4.3 follows law 2.6, as it combines both previous AG's. Thus we prove the completness of the program by proving the completness of the AG., i.e. that starting with

rev1(1) = A1.A2.....An.nil it is possible to obtain a valid tree in AG 4.3. It is shown by following the order of the computations.

Clearly one of the rule will always apply with successful computations of **Sig1**. So it is possible to get a finite partially decorated skeleton with attribute **rev1**. It remains to show that it is possible to complete the decoration, thus that **Sig2** always exists.

In fact **Arg(2) = (L1(u2)-L2(u2), L2(u2)-L3(u2), L1(u2)-L3(u2))**, thus by the property shown previously and the same reasoning (as the second element of rev2 does not depend on the rev1's instanciation), we see that **Sig2** always exists.

Remark : The completness of **REV** could have been proved directly showing that **GOSP(REV)** contains at least all atoms of the form **rev(A1. A2. ... An. nil, An. An-1. .... A1. L-L)**.and using AG 4.2 : first fact of **REV** shows it is true for **n=0**, if it is true for **n** (i.e) the given atoms are valid tree roots in AG 4.2, then, by the second clause of **REV**, we get easely the case **n+1**.

But we wanted to illustrate some thing more powerful: by doing the latter proof with AG 4.2, we did make use of a stronger property, supposing that we had a complete specification of the atoms expected in **GOS(REV)** when by using AG 4.3 we got the same results of (existential) completness without full characterization of the second argument of **rev**. On a so simple example this may appear as a useless subtlety, but this has in fact important practical applications: this shows that by using an appropriate attibute grammar which modelizes a computation of the arguments, one can prove completness without knowing the full characterization of the expected atoms in **DEN** or **GOS**. This will be very useful in bigger programs when no full specification is to be known, completness with regards to some (input) values may still be proved.

Conclusion

The view of Logic Programming as a (proof) tree decoration process permited to make a very simple presentation of the general non deterministic operational semantics. This semantics can be made deterministic without changing its properties (soundness and completness with regards to the denotation). This point of view brings a drastic simplification of the foundations of pure logic programming, comparing the way it is usually presented (as already claimed in

[WML 84]). This comes as an explanation of the remarkable simplicity of logic programming. But this results does not have pedagogical applications only ; we may expect that new ideas and clarifications could arise from a simplified and more general view of the operational semantics of (pure) logic programming, we have already some illustration in the field of the occur check problem which is out of the scope of this report.

In a second part attribute grammars have been presented as a modelization tool of some (more) deterministic operational semantics.

Their main interest comes from the conceptual framework they offer to describe deterministic strategies allowing thus many transfers of expertise between both fields.

Considering main potential applications, we focused our attention on applications in methodology and more specifically on proofs of correctness and completness with regards to some specification. In [DM 88] it is used also to prove partial correctness and run time properties of logic programs with static computation rule. One could have the feeling that using attribute grammars are a sophisticated tool which bracks the simplicity of the proofs. One should thus consider that attribute grammars are in fact a clear and simple way to formalize completely some "evidences" which are not always as trivial as they seem to be at first glances. After understanding the principles of the proof methods, one may become able to handle informal proofs in order to get more confidence of bigger programs.

Finally remark that by describing the proof tree computations as a tree decoration process, we are able to consider other kind of applications like the realization of interpreters acting on partially decorated proof trees. Analogous idea has been considered in [AF 88] in which an attribute grammar evaluator is used to compute proof trees.

One may have noticed that only simple attribute grammars (L-AG) have been used. This class is particularly devoted to describe attributes computation mixtured with (syntax) tree construction.

The question is still open whether more sophisticated class of attribute grammar (like l-ordered or non-circular but not l-ordered) could be of some use.

Morevoer the theory has been developped herein for pure Horn Clause programs. But it has been shown in [DM 88] that this kind of modelization provide a stimulating framework and a good basis to include functional extensions and particular interpretations as in logic programming with constraints.

Bibliography

[AF 88] Attali I., Franchi-Zannettacci P.: Unification-free Execution of TYPOL Programs by Semantic Attribute Evaluation. PLILP'88, Orléans, France, May 16-18, 1988. and ICLP 88, Seattle, August15-19, 1988.

[CD 88] Courcelle B., Deransart P.: Proofs of Partial Correctness for Attribute Grammars with Application to Recursive Procedures and Logic Programming, Information and Computation, V78, N° 1, July 1988, pp 1-58.

[Cla 79] Clark K.L.: Predicate Logic as a Computational Formalism, Res. Mon. 79/59, TOC, Imperial College, December 1979.

[DF 87] Deransart P., Ferrand G.: Logic Programming with Negation: Formal Presentation (in french), RR 87-3, Laboratoire d'Informatique, University of Orléans, June 1987.

[DJL 88] Deransart P., Jourdan M., Lorho B.: Attribute Grammars: Main Results, Existing Systems and Bibliography. LNCS 323, Springer Verlag, August 1988 (first edition : A Survey on Attribute Grammars, INRIA RR 485, 510, 417).

[DM 85] Deransart P., Maluszynski J. : Relating Logic Programs and Attribute Grammars, J. of Logic Programming 1985, 2, pp 119-155.

[DM 88] Deransart P., Maluszynski J.: A Grammatical View of Logic Programming. PLILP 88, Orléans, France, 1988 (to appear in Springer Verlag).

[FLM 88] Falashi M., Levi G., Martelli M., Palamidessi C. : A new Declarative Semantics for Logic Languages. Dipartimento di informatica, University of Pisa, Italy. LP'88, Seattle, August 1988.

[Hue 76] Huet G. : Résolutions d'Equations dans les langages d'ordre 1, 2, ..., ω. Doctorat d'Etat, Univers. Paris 7, September 1976.

[Llo 84] Lloyd J.W.: Foundations of Logic Programming, Springer, December 1987 (first edition 1984).

[Rob 68] Robinson J.A. : A Machine Oriented Logic Based on the Resolution Principle. JACM, V12, N° 1, 1965.

[WML 84] Wolfram D. A., Maher M. J., Lassez J. L. : A unified Treatment of Resolution Strategies for Logic Programs. Proceedings of the 2nd ILPC, Uppsala, July 2-6, 1984, pp 263-276.

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