A grammatical view of logic programming

Pierre Deransart, Jan Maluszynski

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 883

A GRAMMATICAL VIEW OF LOGIC PROGRAMMING

Pierre DERANSART
Jan MALUSZYNSKI

AOUT 1988

A GRAMMATICAL VIEW OF LOGIC PROGRAMMING

Pierre Deransart
INRIA
Domaine de Voluceau - B.P. 105
78153 LE CHESNAY Cedex
FRANCE
uucp : deransar@minos.inria.fr
and
Jan Maluszynski
Linköping University
DCIS
S-58183 LINKÖPING
SWEDEN
uucp : jmz@ida.liu.se

Abstract

A pure logic program has a declarative reading and a procedural reading. This report discusses the idea that these can be complemented by a grammatical reading, where the clauses are considered to be rewrite rules of a grammar. The objective is to show that this point of view facilitates transfer of expertise from logic programming to other research on programming languages and vice versa. Some examples of such transfer are discussed. On the other hand the grammatical view presented justifies some ad hoc extensions to pure logic programming and facilitates development of theoretical foundations for such extensions.

This report summarizes briefly connectionships between logic programs, W-Grammars, Attribute Grammars and Definite Clause Grammars and relates proof methods for logic programs with proof methods of attribute grammars ; new proof methods for run-time properties and completness of logics programs are presented.

Key Words : Logic Programming, Proof-Methods, W-Grammars, Attribute-Grammars, Definite Clause Grammars.

UNE APPROCHE GRAMMATICALE DE LA PROGRAMMATION EN LOGIQUE

Pierre Deransart
INRIA
Domaine de Voluceau - B.P. 105
78153 LE CHESNAY Cedex
FRANCE
uucp : deransar@minos.inria.fr
and
Jan Maluszynski
Linköping University
DCIS
S-58183 LINKÖPING
SWEDEN
uucp : jmz@ida.liu.se

Résumé

Un programme logique pur a deux lectures possibles : une déclarative et une procédurale. Ce rapport introduit et discute l'idée qu'il est possible d'en avoir une troisième : grammaticale. Les clauses sont alors considérées comme les règles de réécriture d'une grammaire. Notre objectif est de montrer que ce point de vue facilite les transferts technologiques du domaine de la programmation en logique vers d'autres champs de recherches sur les langages de programmation et réciproquement. On en présente des exemples. Il apparaît de plus que cette approche grammaticale éclaire le bien fondé de certaines extensions en ouvrant quelques voies pour leurs fondations théoriques.

Ce rapport rappelle les liens existants entre la programmation en logique, les grammaires à double niveau, les grammaires attribuées et les grammaires de clauses définies. Il présente également des méthodes de preuve de programmes logiques établies à partir d'une méthode de preuve pour les grammaires attribuées. Ceci donne lieu à la description de nouvelles méthodes de preuve de complétude et de propriétés dynamiques des programmes logiques.

Mots clés : Programmation en Logique, Méthodes de Preuves, W-Grammaires, Grammaires Attribuées, Grammaires de Clauses Définies.

A GRAMMATICAL VIEW OF LOGIC PROGRAMMING

Pierre Deransart
INRIA
Domaine de Voluceau - B.P. 105
78153 LE CHESNAY Cedex
FRANCE
and
Jan Maluszynski
Linköping University
DCIS
S-58183 LINKÖPING
SWEDEN

Introduction

A pure logic program has a declarative reading and a procedural reading. This report discusses the idea that these can be complemented by a grammatical reading, where the clauses are considered to be rewrite rules of a grammar. The objective is to show that this point of view facilitates transfer of expertise from logic programming to other research on programming languages and vice versa. Some examples of such transfer are discussed. On the other hand the grammatical view presented justifies some ad hoc extensions to pure logic programming and facilitates development of theoretical foundations for such extensions.

The logical notions of the declarative reading are to be related to their direct counterparts in the grammatical reading. As discussed in our early papers the grammar turns out to be a very special case of W-grammar (van Wijngaarden's two-level grammar). This opens for immediate extensions of the notion of logic program by introducing more general classes of W-grammars than these consisting of logic programs. One of them is the class of definite clause grammars. In our presentation the notion of DCG has a direct grammatical reading, in contrast to its usual understanding as a "syntactic sugar" for a special type of logic program. The construction of this program is now seen as a compilation of DCG.

It is well known that van Wijngaarden grammars are closely related to Knuth attribute grammars (AG's) which have been extensively studied and found many applications. Our previous work relating attribute grammars and logic programs uses as the unifying concept a common notion of decorated tree. This paper summarizes briefly this point of view and relates proof methods for logic programs with proof methods of attribute grammars ; new proof methods for run-time properties and completness of logics programs are presented.

1. Proof trees of definite clause programs.

This section relates the grammatical notion of derivation tree with the notion of proof tree of DCP. It shows that the semantics of logic programs can be expressed in the grammatical terms of proof trees.

One of the important concepts related to grammars is the notion of derivation tree. Let G be a context-free grammar and x a symbol of its alphabet. Then the notion of a derivation tree of G and z can be defined as follows.

Definition 1.1.

A *derivation tree* of G and z is any labeled ordered tree satisfying the following conditions :

1. Its root is labeled by z,
2. If a node labeled x has n immediate descendants labeled $x_1, \ldots, x_n$

   (where the indices correspond to the ordering)

then each x_i is either in the alphabet of G or $n = 1$ and x_1 is the empty string, and $x \rightarrow x_1 \ldots x_n$ is a production rule of G.

A *complete derivation tree (a parse tree)* is a derivation tree such that none of its leaf nodes is labeled by a nonterminal of the grammar.

Example 1.1

Let G be the following collection of the production rules written in the BNF notation :

<triple> → <aseq><bseq><cseq>
<aseq> → a | a <aseq>
<bseq> → b | b <bseq>
<cseq> → c | c <cseq>

A parse tree of G and <triple> is shown in Fig. 1.

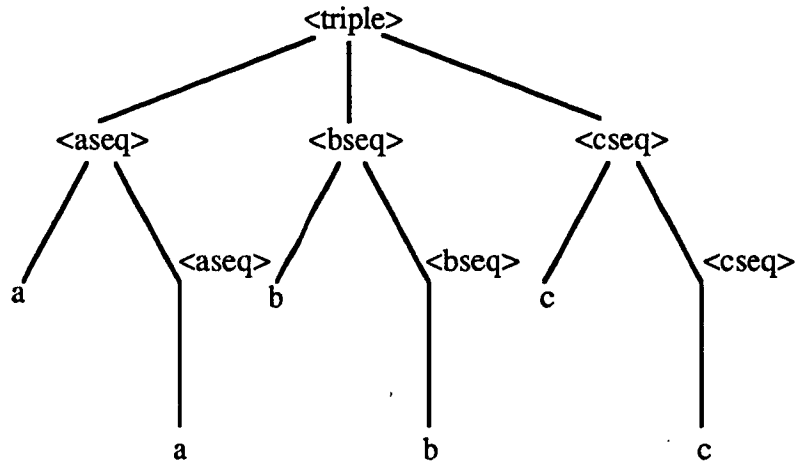

Figure 1

A derivation tree can be seen as a result of pasting together "copies" of the production rules of the grammar. The parse trees of a grammar describe its language and give (possibly ambiguous) structure to every string in the language. Thus a grammar can be considered a specification of a class of parse trees rather than a specification of the language, which is a secondary concept defined in terms of parse trees.

Consider now the notion of definite clause program. Let P be a program and G a goal clause. For simplicity assume that the body of G is one atomic formula *a*. Following Clark [Cla 79] one can introduce a 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 1.2.

A *proof tree* of P and $\leftarrow a$ is any ordered labeled tree satisfying the following conditions :

1. Its root node is labeled by an instance of a ,
2. If a node of the proof tree is labeled x and the nodes of its direct descendants are labeled $x_1, \ldots, x_n$ (where the indices reflect the ordering) then $x \leftarrow x_1 \ldots x_n$ is an instance of a clause of P.

A complete proof tree is a tree whose leaf nodes are instances of the unit clauses (facts) of P.

Example 1.2

Consider the following DCP :

triple(X,Y,Z) ← aseq(X,I), bseq(Y,J), cseq(Z,K).
aseq(a,s(0)).
aseq(a.X, s(I)) ← aseq(X, I).
bseq(b, s(0)).
bseq(b.X, s(J) ← bseq(X, J).
cseq(c, s(0)).
cseq(c.X, s(K)) ← cseq(X, K).

An example of a complete proof tree for P and <- triple(X,Y,Z) is given in Fig. 2.

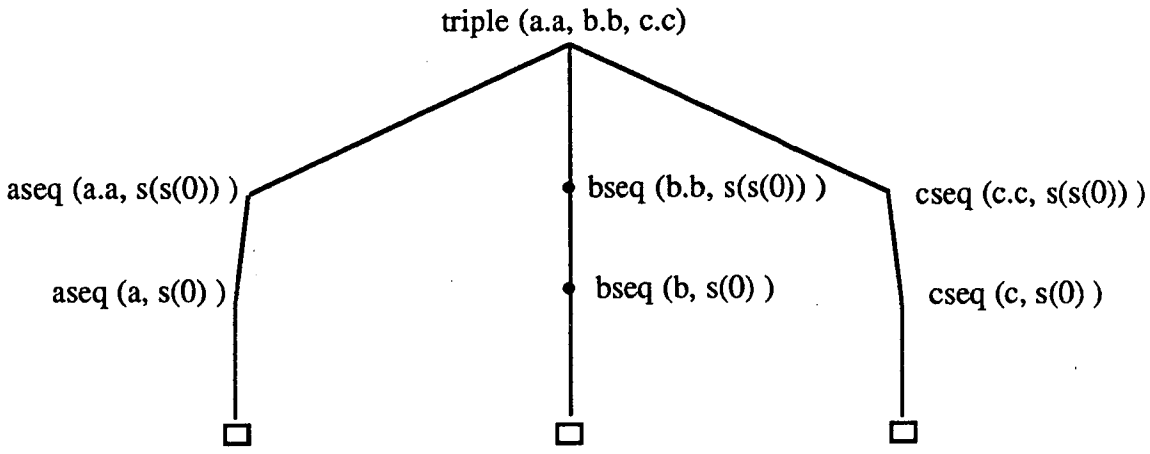

Figure 2

In the sequel we will not distinguish between the trees which are identical up to 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 a complete 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).

On the other hand, the definition of proof tree resembles very much the definition of the derivation tree of a context-free grammar. The difference is that the clauses of P usually include variables and because of that may have infinitely many distinct instances, while every instance of a production rule of a CFG is simply its copy. On the other hand the alphabet of a Definite Clause Program does not include terminal symbols in the sense of CFG.

The concept of proof tree is purely declarative. The question arises how to construct proof trees. In particular, for given program P and goal $\leftarrow a$ it may be interesting to find the set T of all complete proof trees t such that the root of t is an instance of a.. The problem can be solved by using the SLD-resolution. Each refutation for P and $\leftarrow a$ 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 87] shows that the set of complete proof trees corresponding to all refutations of a given goal is the same regardless of the computation rule. This result allows one to define (nondeterministic) operational semantics of a definite clause program. The basis for that is a nondeterministic algorithm that extends proof trees.

Algorithm (Nondeterministic Derivation Algorithm-NDA).

The algorithm transforms an incomplete proof tree into a proof tree of a given definite program P :

For a given incomplete proof tree t

do

- select one of its leaves labeled by an atom g;
- take a renamed version of a clause $h \leftarrow b_1,...,b_n$ of P, such that h unifies with g ; let μ be the mgu ;
- extend t by adding the sons $b_1,...,b_n$ to g ; denote this new tree t' ;
- the result is the tree obtained from t' by applying μ to every label.

od

For a given program P consider all goals of the form $\leftarrow n(X_1, ..., X_q)$, where X_i s are different variables.

Let t(P) be the set of all complete proof trees that can be constructed from that goals by the algorithm NDA. By the independence result t(P) is unique up to renaming of the variables in the labels. Let NDOS(P) be the set of all root labels of the trees in t(P). This set is also unique up to renaming of variables. It characterizes all computed answer substitutions [Llo 87] of P for the most general atomic goals and since it is defined in terms of the nondeterministic algorithm it can be seen as an operational semantics of P. We illustrate this concept by an example program.

Example 1.3

Consider the following program P defining addition on natural integers :

```
plus(zero, X, X).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).
```

Its denotation DEN(P) contains all atoms of the form plus($s^n(0)$,t ,$s^n(t)$) for any non negative n and any term t : t = zero, X, s(zero), s(X), ...where X is any variable, since the atoms identical up to renaming of the variables are not distinguished.

On the other hand, NDOS(P) contains the atoms :

plus(s^n (zero),X ,s^n (X)) $n \geq 0$

For a given atomic goal $\leftarrow g$ the set of all computed answer substitutions can be obtained by unifying g with the elements of NDOS(P).

For example the goal

```
plus(N1, N2, s(zero))
```

unifies only with the following elements of NDOS(P)

```
plus(s(zero),X,s(X))
```

and

```
plus(zero, X, X)
```

giving the two expected answer substitutions

N1 ← s(zero), N2 <- zero

and

N1 ← zero, N2 <- s(zero)

This shows also that NDOS(P) is a representation of DEN(P) (i.e. each element of DEN(P) is an instance of an element of NDOS(P)). This result is known as completness of the SLD resolution [Cla 79, Llo 87,see DF 87 for an approach based on the proof trees]). Thus NDOS(P) is empty if and only if DEN(P) is empty.

2. Definite clause programs as a special class of Van Wijngaardens grammars.

This section shows a class of grammars whose derivation trees resemble very much the proof trees of DCP's. These are Van Wijngaarden grammars (W-grammars) introduced in [Wij 65]. The concept is presented informally in the spirit of the formal definition given in [Mal 84].

The idea behind the concept of W-grammar is to enrich nonterminals of a CFG with parameters ranging over some context-free languages. The notion of a production rule extends thus to a parameterized production rule scheme. Every scheme describes possibly infinite number of production rules each of which can be obtained by instantiating parameters of the scheme by arbitrary chosen elements of their domains.

Example 2.1

For the grammar of Example 1.1 the nonterminal <triple> derives the (regular) language $L_G(\text{<triple>}) = a^+b^+c^+$.

We now introduce parameters in the production rules so that the language specified by the resulting W-grammar is the subset of $L_G(\text{<triple>})$ consisting of the strings with equal numbers of the symbols a, b and c.

The parameters to be introduced are X, Y, Z and N.

Their domains are defined by the following CFG :

$X \rightarrow a \mid a.X$

$Y \rightarrow b \mid b.Y$

$Z \rightarrow c \mid c.Z$

$N \rightarrow s(0) \mid s(N)$

and the parameterized production rules are the following :

<triple(X, Y, Z)> → <aseq(X, N)><bseq(Y, N)><cseq(Z, N)>

<aseq(a,s(0))> → a

<aseq(a.X,s(N))> → a <aseq(X, N)>

<bseq(b,s(0))> → b

<bseq(b.Y,s(N))> → b <bseq(Y, N)>

<cseq(c,s(0))> → c

<cseq(c.Z,s(N))> → c <cseq(Z, N)>

The actual production rules are those which can be obtained from the parameterized production rule schemata by instantiation of the parameters. For example, usingthe substitution {X/a.a,Y/b.b,Z/c.c,N/s(s(0)} one can construct the following production rules :

<triple(a.a,b.b,c.c)> → <aseq(a.a,s(s(0)))><bseq(b.b,s(s(0)))><cseq(c.c,s(s(0)))>

Similarly, by {X/a,N/s(0)} one gets :

<aseq(a.a,s(s(0)))> → a <aseq(a, s(0))>

and by {Y/b,N/s(0)}

<bseq (b.b, s(s(0)))> → b <bseq (b, s(0))>, etc.

There are infinitely many actual production rules which can be constructed in that way since the domains of X,Y,Z and N are infinite.

The language specified by a W-grammar G and a parameterized nonterminal *h* consists of all and only those strings which can be derived from any parameter-free instance of *h* using the rules of W (i.e. the parameter-free instances of rule schemata).

A W-grammar can be seen as a CFG with infinite number of production rules and infinite number of nonterminal symbols. This means that the notion of derivation tree is the same as for CFG's. A derivation tree for the W-grammar of Example 2.1 and for the nonterminal <triple(a.a, b.b, c.c>) is shown in Fig 3. It resembles very much the proof tree of Fig. 2. Both have been obtained by pasting together instances of the rules, in the first case of the DCP, in the other - of the W-grammar. There are, however, important differences between the concept of DCP and the concept of W-grammar which are discussed below.

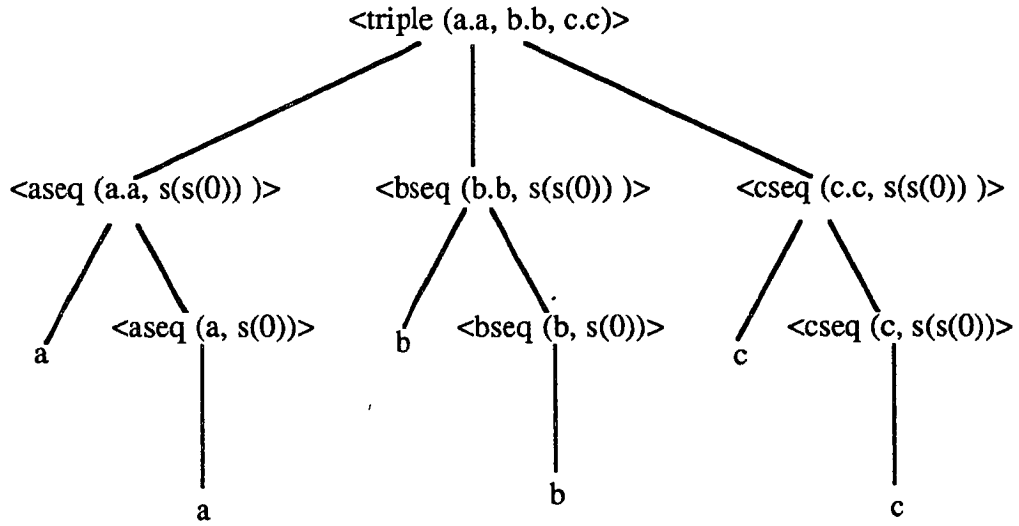

Figure 3

Terminal symbols and the language.

The rule schemata of a W-grammar, and consequently the production rules, may include terminal symbols. The notion of complete derivation tree gives thus a concept of terminal string derivable from a given nonterminal and consequently the notion of the language specified by a given W-grammar and given nonterminal symbol. The language specified by a parameterized nonterminal is the union of all languages specified by its parameter-free instances.

The clauses of a DCP include atoms and no concept of a terminal alphabet in the sense discussed above is introduced.

Domain specifications.

W-grammar includes as its component a finite set of context-free production rules specifying the domains of the parameters, called the *metagrammar*. The parameters are instantiated by the elements of the domains. Thus, the parameters play the role similar to that of variables in the clauses. Similarly the domain elements may be compared with terms. It is worth noticing that for a given finite alphabet of functors the Herbrand universe is a context-free language and can be easily defined by an (unambiguous) context-free grammar. Therefore, a DCP can be viewed as a W-grammar with the empty terminal alphabet, and with the domain of every parameter defined by default to be the Herbrand universe.

Thus the notion of W-grammar is an extension of the notion of DCP which allows many-sorted terms (sorts are the nonterminals of the metagrammar). It opens also for user-defined syntax of terms, where the structure of a term is defined by the parse tree of the string in the metagrammar. This approach brings also certain difficulties since, if the metagrammar is ambiguous, the structure of the string is not unique.

Resolution

The derivation trees of a W-grammar have no parameters in node labels. Thus, considering a DCP program to be a W-grammar one restricts the class of proof trees to ground ones. For a given program P the ground proof trees may be constructed as ground instances of not necessarily ground proof trees obtained by SLD-resolution. This raises the problem whether resolution can be extended for W-grammars. In the case of unambiguous metagrammar the extension amounts to replacing of unification of one-sorted terms by unification of many-sorted terms , as discussed in [MN 82a]. In the case of ambiguous metagrammar the problem is more complicated and may be solved by introduction of unification under associativity in place of term unification. Though this type of unification is decidable [Si 84] this generalization seems to be of little practical interest.

2.3. Proliferation of concepts

We now show some application and practical consequences of the differences between DCP's and W-grammars.

The first question is the impact of W-grammars on DCP's.

There are two possible extensions of logic programs which naturally fit in that framework. The first one is the concept of **Definite Clause Grammar.**

It is obtained by introducing terminal symbols and allowing the clauses to include them. A DCG is then a special type of W-grammar rather than a syntactic sugar for a special class of DCP's [PW 80].

Example 2.2

A simplified version of the W-grammar of Example 2.1 is the following DCG, where the predicates carry the information about the length of the underlying strings, expressed as Prolog numerals.

triple(N) → aseq(N),bseq(N),cseq(N).

aseq(1) → [a].

aseq(N) → [a], aseq(M), {N is M+1}.

bseq(1) → [b].

bseq(N) → [b], bseq(M), {N is M+1}.

cseq(1) → [c].

cseq(N) → [c], cseq(M), {N is M+1}.

This point of view opens for development of alternative implementations of DCG's, like that mentioned in [Nil 86] and based on the LR-parsing techniques.

The second extension is the introduction of many-sorted type discipline and domain declarations in logic programs. This was first suggested in [MN 82a, MN 82b]. The idea of using domain declarations for improving efficiency of the compiled logic programs was explored in [Nil 83].

It appears also in Turbo Prolog [Bor 86]. To illustrate this statement we quote an example Turbo Prolog program from the manual [p. 51].

```
domains
   title, author  =  symbol
   pages          =  integer
   publication    =  book(title, pages)
predicates
      written_by(author, publication)
      long_novel(title)
clauses
      written_by(fleming, book("DR NO", 210).
      written_by(melville, book("MOBY DICK", 600)).
      long_novel(Title) :-written_by(_,book(Title, Length)), Length> 300.
```

The domain declarations of this program are (non-recursive) context-free production rules, where

title, author = symbol

is a shorthand for two rules :

title → symbol

author → symbol

The rules include :

nonterminals : title, author, symbol, integer, publication,

terminals : book, , , (,)

and refer to the standard domains of integers and symbols for which grammatical rules are not included.

The predicate declarations specify types of the arguments by using nonterminals of the metagrammar. The terms of the clauses have to observe these declarations, i.e. they are sentential forms derived in the full metagrammar (including its standard implicit part). The variables are implicitly typed by parsing of the terms according to the predicate declarations : the variable Title is of type *title*, hence *symbol*, the variable Length is of type *pages*, hence *integer*.

Let us discuss now the impact of DCP's on W-grammars. As pointed out in Section 2.2 the original definition of W-grammar gives no practical possibility for constructing derivations. In [Mal 84] a notion of transparent W-grammar was defined which opens for application of resolution techniques in construction of proof trees. The idea is that the metagrammar should be unambiguous and every parameterized nonterminal should be its sentential form. In this case it has a unique parse tree which can be seen as a term and can be subject of unification. Now the formalism of transparent W-grammars can be used as a logic programming language with many-sorted type discipline and with the operational semantics based on resolution. An experimental implementation of such a language is described in [Näs 87].

3. Definite Clause Programs and Relational Attribute Grammars.

As discussed in Section 1 proof trees of a DCP resemble derivation trees of CFG's but have much more complicated labels. The question arises whether the concept of proof tree can be decomposed into two simpler concepts: a context-free skeleton tree and a labeling mechanism that decorates the skeleton. This idea is not new. It was formalized by Knuth as the notion of Attribute Grammar, which found many applications. Our previous work [DM 85] attempted to relate formally these two notions.

In the rest of this paper we briefly outline some ideas based on that work.

3.1. Relational Attribute Grammars

This section presents a general notion of Relational Attribute Grammar [CD 87] which can be seen as a generalized version of DCG where labeling mechanism is defined separately from the context-free skeleton grammar.

The labeling mechanism is then restricted and refined to the usual more specific notion of AG. Special attention is devoted to comparison of the formalisms to indicate possible transfer of ideas between them.

To facilitate presentation we assume that the set of terminal symbols of the CFG is empty so that the definition presented is an extension of DCP rather than DCG.

Definition 3.1

Relational Attribute Grammar (RAG) is a 5-tuple

<N, R, Attr, Phi, Int>

where

N is a finite set of non terminal symbols,

R is a finite set of context free rules, built with N. R can be viewed as a N-sorted signature of a R-Algebra. Elements of R will be denoted as usual :

$$n_0 \rightarrow n1... , n_k..., n_p \quad p \geq 0$$

Thus N and R give rise to a set of context-free parse trees and the other elements of RAG 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 presentation it is assumed that these sets are disjoint and linearly ordered. Thus the attributes associated with a non terminal n will be denoted $n_1,..., n_q$, where n_i denotes the ith attribute of n in the ordering. 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 non terminal. For example consider the rule

$$n \rightarrow n \ n$$

where n has two attributes n_1 and n_2. The different occurrences of the attribute n_1 will be denoted : n_1 (0), n_1 (1) and n_1 (2), or briefly n_{10}, n_{11}, n_{12}, and those of n_2 : : n_{20}, n_{21}, n_{22}.

Phi is an assignment of a logic formula Phi_r to each production rule r in R. The formula includes attribute occurrences as the variables. Thus for every interpretation it defines some relation between the values of the attribute occurrences in the rule r.

Int is the interpretation of the language used in Phi : it defines the domains, associates functors of the language with functions on the domains and predicate letters with relations.

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 *n* is to be additionally labeled by a *k*-tuple of values in the domain of Int, where *k* is the number of attributes of *n* .

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 Phi_r. It is required that the formula is true in Int 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 RAG. Thus, the semantics of a RAG 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 : Relational Attribute Grammar for 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 87].

N is { fact },

R is { fact $\rightarrow e$, fact $\rightarrow$ fact }, (e denotes the empty string)

Attr is { fact1, fact2 }, where fact1 denotes the argument and fact2 the results.

Phi is { fact1=0 and fact2 = 1, fact1(1)=fact1(0)-1 and fact2(0)=fact2(1)*fact1(0) }

Int is Nat, the domain of the natural (or non negative) integers together with the usual operations and relations (equality ...).

A valid decorated tree for the given RAG is shown below (Figure 4).

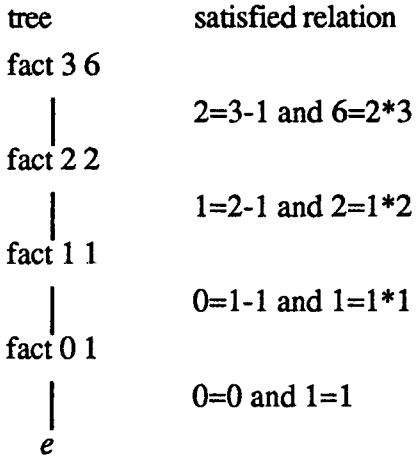

Figure 4

End of Example

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 Int with every nonterminal n of the grammar. The relation is q-ary, where q is the number of attributes of n. A q -tuple of attribute values is in the relation iff it is a root label of some valid proof tree for n. This relation will be called the *basic relation* and denoted BR_n. The family of the relations $\{BR_n \mid n \text{ in } N\}$ will be denoted BR.

It is easy to see that the relation associated with the nonterminal fac of Example 3.1 is the set of pairs (i ,i !) for $i \geq 0$.

3.2. DCP's as a special class of RAG's.

We now show that a DCP can be viewed as a special kind of RAG. In Section 1 two types of "grammatical" semantics of DCP's have been defined. The first one associates with a DCP P the set of all proof trees, the other deals with the computation trees of the NDA algorithm. The idea is then to construct a RAG whose valid trees are isomorphic to the considered class of proof trees of P.

A general framework for this is as follows.

Let P be a DCP with :

- the alphabet of functors FUNC,
- the alphabet of predicate letters PRED
- and with the set of clauses CLAUSE.

We associate with it a RAG

G = < N, C, Attr, Phi, Int >

where

N is PRED,

C is CLAUS in which all the arguments have been removed,

Attr is a set of attribute names denoting the arguments of the predicates (the *i* th argument of the predicate p will be denoted p_i),

Phi will be defined below; for each of the two types of semantics of P a separate definition will be given

Int is the interpretation of the logical language used for defining Phi ; the domain of interpretation are (not necessarily ground) terms constructed from the symbols of FUNCT ; the symbols of FUNCT are interpreted as term constructors. It is called the canonical term interpretation denoted T (V, FUNC) or in short T.

By specifying two different sets Phi we define now two different RAG's :

- RAGD whose valid trees model the proof trees of P, and
- RAGO whose valid trees model the computation trees of the NDA algorithm.

Let

$c : n_0(T_0) :- n_1(T_1) ...n_k(T_k) ...n_p(T_p)$

be a clause of P, where T_i for i=1,...p denote the tuples of terms which are the arguments of the predicates n_i.

The corresponding context-free production rule is :

$c : n_0 \rightarrow n_1, ...,n_k, ..., n_p,$

The associated formula Phi_c of RAGD is

$\exists X$ AND(for all $k > 0$ and $1 \leq j \leq q_k$) $n_{kj} = t_{kj}(X)$

where = is interpreted in Int as the identity on terms, t_{kj} is the j-th term in the tuple of terms T_k and X are all variables appearing in the clause c.

Notice that n_{kj} are (the only) free variables of Phi_c. Thus, for a given valuation of n_{kj}'s Phi_c is true iff there exists a substitution ß, such that n_{kj} is the instance of t_{kj} under ß.

It is shown in [DM 85, CD 87] that BR for RAGD is exactly DEN(P).

The associated formula Phi_c of RAGO is

AND (for $1 \leq j \leq q_0$) $n_{0j} = \partial(t_{0j})$ (or in short $N_0 = \partial(T_0)$)

where

$$\partial = mgu(<T_1,...T_p>,<N_1,...N_p>) ;$$

N_k is the tuple of attributes n_{kj} for j from 1 to q_k, T_k is the tuple of arguments of the k-th atom in the body of the clause, and the variables of the clause have been renamed away of the variables of the N_k's values.

It is shown in [Der 88c] taht BR for RAGO is exactlly NDOS(P), which corresponds also to the least N-models in [FLM 88].

Example 3.2

Consider the program for addition of Section 1

```
plus(zero, X, X).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).
```

Its underlying set of production rules is :

$plus \rightarrow$

$plus \rightarrow plus$

The corresponding formulae of RAGD are

$\exists\ X\ plus_1 = zero \wedge plus_2 = X \wedge plus_3 = X$

$\exists\ X,Y,Z\ plus_{01} = s(X) \wedge plus_{02} = Y \wedge plus_{03} = s(Z) \wedge plus_{11} = X \wedge plus_{12} = Y \wedge plus_{13} = Z.$

The corresponding formulae of RAGO are

$plus_1 = zero \wedge plus_2 = X \wedge plus_3 = X$

$plus_{01} = \partial(s(X)) \wedge plus_{02} = \partial(Y) \wedge plus_{03} = \partial(s(Z))$

where

$$\partial = mgu(\langle X, Y, Z\rangle, \langle plus_{11}, plus_{12}, plus_{13}\rangle)$$

assuming that X, Y, Z do not appear in $plus_{11}$, $plus_{12}$ and $plus_{13}$.

Fig. 5 shows one example of valid tree for both RAG's.

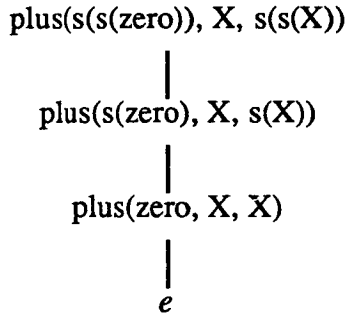

Figure 5

3.3 Proliferation of ideas

The concept of RAG was first introduced in [DM 85] as a "common denominator" of DCP's and Attribute Grammars. There are two important differences between RAG's and DCP's :

Interpretations

RAG's have explicit concept of interpretation, which is a basis for defining the semantic relations BR. There is no restriction on the type of semantic domains or on the functions associated with the functors of the alphabet. DCP programs refer to the term domain and the functors are always interpreted as term constructors.

Operational Semantics

The relations specified by a DCP can be computed by the resolution techniques. No operational semantics of RAG's has been suggested in [DM 85].

The differences suggest a possibility of a natural extension of the concept of DCP to logic programs with interpreted functors. In practical applications of attribute grammars the interpretation of the functors is provided as the "semantic functions" [Kn 68], which are some external procedures. The formalism of RAG's incorporates such low-level procedures into the high-level and declarative structure of semantic rules. An important feature of the formalism is that it imposes no restrictions on the language of the procedures or their actual form. Now the idea is to use the same approach to incorporate external functional procedures into logic programs. This should give a general method of amalgamation of logic programming with functional programming without being specific about the functional language to be used. Although the Concept of RAG gives the declarative semantics of such amalgamation it provides no direct suggestion for an operational semantics on which a practical implementation could be based. To solve this problem two ways can be suggested.

The first possibility is to modify resolution so that it can handle interpreted terms. If some appropriate axiomatization of the interpretation is available this would amount to logic programming with equality [JLM 84]. But this would also bring all difficulties related to E-unification [Si 85]. However, in the tradition of AG's it is more natural to expect that an implementation of semantic functions is given rather than their axiomatization. If the problem could be solved in this framework it would open for re-usability of the existing functional procedures in logic programs without violation of the declarative semantics. An incomplete solution along this lines has been suggested in [LBM 88] and [BM 88]. The idea is to combine Robinson's unification with term reduction done by calling the external procedures defining the interpretation of the functors. This generalized unification, called S-unification, is an incomplete E-unification. If it produces a unifier, the unifier is most general and unique. If it fails, no E-unifier for the arguments exist. In some cases it may report that it is unable to solve the problem. This resembles run-time errors reported by the arithmetic procedures of Prolog.

Example

For the RAG of Example 3.1 specifying the factorial the corresponding amalgamated program may look as follows :

fac(0,1).

fac(X,Y*X) ← fac(X-1,Y).

where 0,1,* and - are functors interpreted on the domain of nonnegative integers.

Consider the goal

← fac(3, V).

The S-unifier of fac(3,V) and fac(X,Y*X) is {X/3, V/Y*3} and results in the next goal ← fac(2,Y). Eventually the computation succeeds with the answer V = 6.

The second way of giving an operational semantics for amalgamated programs would be transformation of the corresponding RAG into the usual type of AG where some standard attribute evaluation techniques could be used. The construction and its limitations are discussed in [DM 85]. Some comments will be also given in the next section.

3.4. Functional Attribute Grammars and Logic Programs.

The RAG's provide no way of computing the values which can be assigned to the positions of a given tree. Their semantics is purely declarative and can be easily related to the semantics of logic programs as discussed above. However, the formalism of attribute grammars was originally introduced with some additional restrictions which make it possible to compute attribute values of a derivation tree in a deterministic way. These can be expressed as restrictions on the form of formulæ associated with the production rules. The basic concept is splitting of attributes into

synthesised and inherited. The splitting is called *direction assignment* since it describes a dependency relation between attributes of any parse tree . It is then required that the formula associated with the production rule is conjunction of the atomic formulae of the form X = t where X is an inherited attribute of a body nonterminal or a synthesised attribute of the head nonterminal of the rule and the only variables of t are inherited attributes of the head or synthesised attributes of the body nonterminals. Furthermore some global well-formedness condition is formulated which excludes circular dependencies between the attributes of parse trees thus making possible unique decoration of any given proof tree. Following [DM 85] such grammars will be called Functional Attribute Grammars (FAG's).

As discussed above RAG's include DCG's (thus also DCP's) as a special subclass. Another subclass are FAG's with the operational semantics defined by the attribute evaluation techniques. The intersection of the subclasses is not empty. The elements of the intersection have a similar declarative semantics based on the concept of decorated tree but different operational semantics: the decorated tree may be constructed either by resolution, where parsing is interleaved with parse tree decoration or by attribute evaluation technique applied to the CF parse tree constructed before hand by some CF parsing algorithm. This opens for transfer of techniqes between AG's and Logic Programs.There have been many attempts in both directions. Let us mention briefly a few areas.

Writing Parsers in Prolog

In implementation of AG's as well as in other applications it is necessary to generate parsers for given CF Grammars. It has been argued that it is rather easy to compile a CFG into a Logic Program that can be used for parsing. The approaches, like [Col 78, PW 83, CH 87] may be useful to write a top down nondeterministic parser, but the resulting program is rather inefficient on big grammars in case of deterministic languages. On the other hand the top down approach can be made deterministic in some cases if the CFG is LL(1) and some more conditions are satisfied [Der 88]. Thus it has been proposed to encode known algorithms for bottom up deterministic parsing [CH 87, Nil 87, MTH 83, Hen 88] are variants of such approach. It seems that the resulting deterministic parsers can be compared favorably with those obtained by traditional methods (assuming the parsers are built automatically). But the only real advantage of using Prolog seems to be the ease of rapid implementation of parser constructors rather than efficiency of the parsers. This point requires further investigation.

Using parsing techniques in logic programming

It seems that the real strength of the logic programming is its ability to handle nondeterministic parsing. Thus suggestions have been made to combine deterministic parsing methods with resolution for restriction of the nondeterminism of execution. For example [UOK 84] is an attempt to restrict the clause selection using bottom up parsing techniques and look ahead. Similarly [Nil 86] extends LR-parsing techniques to nondeterministic parsing and combines this with resolution to give an alternative implementation of DCG's. For some examples this look-ahead technique allows a considerable reduction of the search space.

Implementation of Attribute Grammars in Prolog

Since Functional Attribute Grammars resemble Logic Programs many authors suggested to use Prolog to write them and to execute them. Different implementations of this idea can be found in [Abr 84, Arb 86, Hen 88]. The analogy is particularly clear for FAG schemes, where the interpretation is not given, or for FAG's with the free interpretation, where the functors are interpreted as term constructors. A formal study of this relation was presented in [DM 85]. It has been shown that such a FAG can be transformed into an equivalent logic program where the nonterminals correspond to the predicates, the attributes correspond to the argument positions of the predicates and the production rules to the clauses. In particular, the inherited attributes of the head of a grammatical rule and the synthesized attributes of the body nonterminals are represented as distinct variables in the corresponding clause.

As shown in [DM 85] a pure FAG's can be turned into a logic program in which all input arguments (i.e inherited of the head and synthesized of the body) are simple distinct variables.

A classical example of FAG is that of [Knu 68] describing computation of the decimal value of binary numbers specified by a CFG.

The corresponding logic program is the following DCG :

binnum(V1 plus V2) → num(0, L1, V1), [.], num(minus(L2), L2, V2).

num(R0, s(L1), V1 plus V2) → num(s(R0), L1,V1), digit(R0,V2).

num(R0, 0, 0) → [ ].

digit(R0, 0) → [0].

digit(R0, exp(2,R0)) → [1]

In the original FAG the nonterminal *binum* has one synthesized attribute (the resulting decimal value), *num* one inherited (R, the range of the right most digit of the derived string of digits) and two synthesized (L, the length of the derived string of digits and V the corresponding decimal value), and *digit* has two attributes (R the range of the digit and V its decimal value). These are represented by the variables of the DCG rules.

It is shown in [DM 85] that in such DCG the circularity problem is equivalent to the occur-check problem, i.e. that such program can be (symbolically) evaluated by a Prolog interpreter without occur-check if and only if no cyclic definition of attribute instance may appear in some tree.

The circularity test is decidable (see [DJL 88] for more details and references). The given example satisfies this property and thus it is possible to obtain the symbolic value of the synthesized attribute of the root by executing the corresponding Prolog program. For example, with the string 1.1 we will obtain

(0 plus exp(2,0)) plus (0 plus exp(2, minus(1)))

This apparently easy correspondence between FAG's and Logic Programs raises the question whether the encoding of a FAG in Prolog will lead to an efficient attribute evaluator. Unfortunatly it is not the case.

First of all the experiments with attribute grammar evaluation show that the decorated trees are so large ([DJL 88]) that available Prolog systems cannot handle efficiently corresponding proof trees. However, the main problem comes from the interpreted operations. The constructed logic program produces only symbolic expressions for attribute values. They quickly become very large and their efficient evaluation is a difficult problem. Thus, for efficiency reasons one should interleave construction of the tree with computation of the attribute values.

Since inherited (resp. synthesized) attributes play the role of input (output) arguments, for some AG's implementation of the interleaving may be easy. It is the case if the input arguments of every predicate in the constructed program are ground at the moment of its call and its output arguments are ground at the success. (For more discussion see [DM 85]. According to [Dr 87] this type of logic programs is quite common.)

For example the second rule of the example DCG may be transformed as follows :

num(R0, L0, V0) → {R1 is R0+1}, num(R1, L1, V1), {L0 is L1+1}, digit(R0, V2), {V0 is V1+V2}.

Such translation is not possible in general. For example in the first rule of the DCG the first argument of the second body literal is *minus(L2)*. It should be translated into *R2 is -L2* . But *L2* is not known; it corresponds to a synthesized attribute and cannot be computed until the subtree is constructed. Thus the method fails.

This problem is well known in the field of attribute grammar where different cathegories of algorithms have been introduced to evaluate attributes. For a complete review of this aspect see [DJL 88]. So the logic programming implementations of attribute grammar are essentially encodings of known algorithms in Prolog [Hen 88]. No strong indication exists until now that the resulting logic programs could be substantially more efficient. However, known AG implementations in Prolog show that this encoding is possible and can lead to rapid prototyping of compilers or metacompilers.

Use of Functional Attribute Grammars to compute proof trees.

It is an interesting problem whether attribute evaluation techniques can be used for giving alternative semantics for some classes of logic programs. For example a well-known class of AG's are L-AG's where all attributes can be computed during top-down parsing. There is a corresponding class of DCG's for which the process of proof tree construction would "simulate" the process of parsing and attribute evaluation. This generalizes the example above explaining incorporation of the attribute evaluation in the second rule of the DCG. The question is whether this approach may improve efficiency of computation of a logic program.

In [DF 85] a category of "data driven" logic programs is defined in which L-AG are used to analyse the data flow in the proof trees and deduce the best strategy depending on the instantiation patterns of the arguments of the goal.

In [AF 88] a special class of logic programs is introduced for which the proof tree construction can be performed by an attribute evaluator.

4. Fixpoint induction in Logic Programming

As discussed in Section 3 DCP's can be seen a special class of RAG's.

This section applies the validation method for RAG presented in [CD 87] for proving properties of Logic Programs. The kind of proved properties depends on the type of semantics considered. For example for the RAG modeling the nondeterministic operational semantic, properties of the computed answer substitutions can be proved. On the other hand, for the RAG modeling the declarative semantics the properties to be proved concern all correct answer substitutions.

The proof method discussed was presented in [CD 87]. It resembles fixpoint induction for algorithmic languages. A similar approach was suggested in [Cla 79] and [Hog 84] but without giving technical details.

4.1 Partial correctness of RAG's.

This section outlines basic concepts of the proof method for RAG's presented in [CD 87].

The notion of partial correctness of a RAG is defined with respect to some specification. Let G be a RAG <N, R, Attr, Phi, Int>. Let L be a logical language with an interpretation that extends Int.

A *specification* S for G is a family $\{S_n\}$ of formulas indexed by n in N and such that the only free variables of the formula S_n are the attributes of n. If the number of attributes of n is q then the variables will be denoted $n_1, n_2, \ldots n_q$.

For example, the following formulas are specifications for the RAG *fact :*

$S1_{fact}(fact_1,fact_2) : fact_1 \geq 0$

$S2_{fact}(fact_1,fact_2) : fact_2 = fact_1!$

Another specification for the same RAG could be.

$S3_{fact}(fact_1,fact_2): \exists\ m\ \ fact_2 = m!$

A specification is said to be *valid* iff for every q-tuple $<v_1,\ldots,v_q>$ of attribute values which is the root label of a valid tree for n, the formula associated with n is true in Int under the valuation $\{n_1/v_1,\ldots n_q/v_q\}$.

In other words, if S is valid then every element of the basic relation BR_n will also satisfy S_n. Thus, since the interpretation is given, every S_n specifies a relation which is a superset of BR_n. A specification which defines exactly BR is then the strongest valid specification.

It is shown in [CD 87] that BR is not always first order expressible. As the basic relation BR can be unknown or very difficult to express, we need a method to prove the validity of a specification. In [CD 87] the notion of inductive specification is introduced as a basis of a proof method similar to the fixpoint induction.

A specification S is *inductive* iff for every r in R : $n_0 \rightarrow n_1, ..., n_k, ..., n_p$ the formula

AND (for k from 1 to p) $S_{nk} \wedge Phi_r \Rightarrow S_{n0}$

is valid in the interpretation Int.

The formulas S_{nk} are those obtained from S_n by replacing the free variables denoting the attributes by corresponding attribute occurrences, i.e. $n_i(k)$, or in short n_{ik}.

As an example, consider the specification $S1_{fact}$. It is inductive, since the formulas

$fact_1 = 0 \wedge fact_2 = 1 \Rightarrow fact_1 \geq 0$

and

$fact_1(1) \geq 0 \wedge fact_1(1) = fact_1(0) - 1 \wedge fact_2(0) = fact_2(1) * fact_1(0) \Rightarrow fact_1(0) \geq 0$

are valid under the considered interpretation (as $fact_1(0)=fact_1(1)+1$).

The method of the inductive specification is general as it is shown in [CD 87] that a specification is valid if and only if it is the logical consequence of an inductive specification. It is a consequence of the facts that every inductive specification is valid and that there exists the strongest inductive specification that defines BR.

Notice that a specification can be valid without being inductive.

For example $S3_{fact}$ is not inductive, but it is a logical consequence of $S2_{fact}$.

4.2 Declarative semantics and related properties.

The declarative semantics of a Logic Program P is the set of its proof trees or the set of its proof tree roots denoted DEN(P) (see section 1). As discussed in Section 3.2 it can be modeled by the grammar RAGD whose set of valid trees is isomorphic with the set of proof trees of P.

Now we can transfer all the notions introduced for the RAG's to Logic Programming (it has been done in [CD 87] but we use here slightly different attribute definition).

A specification for a logic program P is a family of logic formulas indexed by the predicate letters. (If a formula associated with some predicate p is the logical constant true then it is omitted). The language of formulae has some fixed interpretation. It is assumed that it defines meaning of all the functional symbols of FUNC. Notice that no restriction is put on the kind of interpretation; it needs not be the term interpretation.

A specification S is valid if for every atom of DEN(P) of the form $n(t_1, ..., t_q)$ the formula $S_n(t_1, ..., t_q)$ is valid in the interpretation considered.

A specification is valid if it is a logical consequence of an inductive specification.

A specification is inductive iff for every rule c

$n_0 \rightarrow n_1, ..., n_k, ..., n_p$ corresponding to the clause $n_0(T_0)$:- $n_1(T_1), ..., n_k(T_k), ..., n_p(T_p)$

where T_k is $t_{k1}, ..., t_{kj}, ..., t_{kqk}$, the formula

AND (for k from 1 to p) $S_{nk} \wedge Phi_c \Rightarrow S_{n0}$

where

Phi_c is $\exists$ ..X.. AND(for k from 1 to p and $1 \leq j \leq q_k$) $n_{kj} = t_{kj}(..X..)$

is valid in the interpretation considered.

The formula can be simplified by replacing existential quantifiers in premisses by universal quantifiers applied to the whole formula.

Thus :

A specification is inductive if the universal closure of the formula

AND (for k from 1 to p) $S_{nk}(...t_{kj}(...X...)...) \Rightarrow S_{n0}(...t_{0i}(...X...)...)$

is valid in the interpretation.

This gives a simple method for proving partial correctness of a logic program w.r.t. a specification. (In practice its applicability is restricted to small programs. Other proof methods, which can be used for larger programs have been considered in [CD 87]. More developments and justifications are given in [Der 88b]) One important aspect of the method presented is that it allows arbitrary interpretations. Thus it applies also for extensions of logic programs , like amalgamated programs mentioned in Section 3.2. In fact if a specification is proved inductive using a non-term interpretation, then the values of the labels of all proof tree roots will satisfy this specification. The converse is not true (some specification may be inductive even if there is no proof tree at all or few proof trees -see FACT2 below-), but this is of no use here.

We now illustrate the method by some examples.

Example 4.1 program FACT1

```
fact(zero,s(zero)) .
fact(s(X),mult(s(X),R)) :- fact(X,R) .
```

interpreted on Nat (natural numbers) with *zero* as 0, *p* as -1 and *mult* as multiplication.

It is easy to show that the specification S2

$S2_{fact}(fact_1,fact_2) : fact_2 = fact_1!$

is valid, provided that ! is interpreted as the function factorial.

Indeed :

s(zero) = zero!

and

$R = X! \Rightarrow mult(s(X),R) = s(X)!$

what is easy to prove in the intended interpretation :

1 = 0! ok

$R = X! \rightarrow (X+1)*R = (X+1)!$ ok

Note that we would have exactly the same proof with the following program FACT2 which corresponds to the RAG given in the section 3.2 :

```
fact(zero,s(zero)).
fact(X, mult(X,R)) :- fact(p(X),R).
```

Thus the specification $S2_{fact}$ is inductive, but the denotation of this program is reduced to the singleton fact (zero, s(zero)) ; it has few proof trees ! This shows that the proofs of partial correctness say nothing about the existence of proof trees satisfying the specification. For more complete discussion of this problem see [DF 88, Der 88b].

Example 4.2 program linear reverse REV

(naive reverse with difference lists)

```
rev(nil, L-L).
rev(A.L, R ) :- rev(L, Q), concd(Q, A.M-M, R).

concd(L1-L2, L2-L3, L1-L3).
```

Valid specification : (repr is defined on difference lists only and reverse on lists -i.e all variables appearing in the formulas are universally quantified on difference lists or lists depending on the type of the arguments of repr, reverse or concat)

S_{rev} : repr(rev_2) = reverse(rev_1)

S_{concd} : repr($concd_3$) = concat(repr($concd_1$),repr($concd_2$))

The following formulas hold for lists and difference lists :

repr(L-L) = reverse(nil) (as repr(L-L) = nil=reverse(nil))

and

repr(Q) = reverse(L) $\wedge$ repr(R) = concat(repr(Q), repr(A.M-M)) $\Rightarrow$ repr(R) = reverse(A.L)

as reverse(A.L) = concat(reverse(L), A.nil) and repr(A.M-M) = A.nil .

It is also easy to show the correctness of the following specification

S_{rev} : list(rev_1) $\wedge$ difflist(rev_2)

S_{concd} : difflist($concd_1$) $\wedge$ difflist($concd_2$) $\rightarrow$ difflist($concd_3$)

where list and difflist are term typing functors ranging respectively on lists and difference lists.

Proofs of this kind are is easy to perform ; many useful (declarative) properties may be inferred very quickly, improving the ability of the programmer to write a logic program.

Another kind of properties which are generally easy to deduce are "modes", i.e. properties of the form : if some arguments are ground then some others are too. For example in the program FACT1 or FACT2 it is easy to deduce (with the term interpretation) that both arguments are always ground.

A more interesting example of specification of this kind concerns the following program PERM, specifying all the permutations of a given list :

```
perm(nil,nil).
perm(A.L,B.M) :- extract(A.L,B,N), perm(M, N).

extract(A.L, A, L).
extract(A.L, B, A.M) :- extract(L, B, M).
```

Note the specific construction of the second clause of perm (second atom of the body).

One can show by induction that the following specification is valid :

S_{perm} : ground($perm_1$) $\Leftrightarrow$ ground($perm_2$)

$S_{extract}$: ground($extract_1$) $\Leftrightarrow$ (ground($extract_2$) $\wedge$ ground($extract_3$))

This type of properties can be also proved in an automatic way, e.g. by abstract interpretation.

4.3 Non Deterministic Operational Semantics and related properties.

The non deterministic operational semantics of a logic program P is the set of the effectively computed proof trees by the non deterministic algorithm NDA (see section 1) starting with the most general goals.

In Section 3 this semantics was modeled by the relational attribute grammar RAGO. RAGO can be viewed as a functional AG (see section 3.2) in which all attributes are synthesized.

For example consider RAGO for the program plus :

Phi_1 : $plus_1$ = zero $\wedge$ $plus_2$ = X $\wedge$ $plus_3$ = X

Phi_2 : ($plus_{01}$, $plus_{02}$, $plus_{03}$) = ∂((s(X), Y, s(Z)))

where ∂ = mgu((X, Y, Z), ($plus_{11}$, $plus_{12}$, $plus_{13}$))

The tree in Fig. 5 (section 3.2) is valid.

It is also a proof tree of the program. However valid trees of RAGO(P) need not be proof trees. For example, the valid tree of Fig. 6 for the RAGO of the program REV (4.2 Example 2) is not a proof tree.

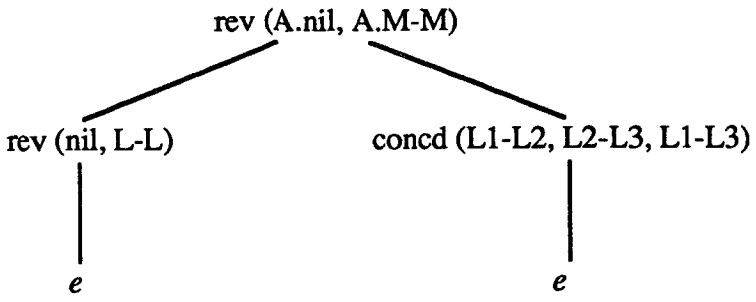

Figure 6

To get the corresponding proof tree it is sufficient to apply recursively in a descendent manner (as the instantiation does not affect roots) all the mgu used in the attribute instances definitions [Der 88c].

For the example above the substitutions produced are

L ← A.M, N ← nil, Q ← A.M-A.M, L1 ← A.M, L2 ← A.M, L3 ← M, R ← A.M-M

and the corresponding proof tree is shown on Fig. 7.

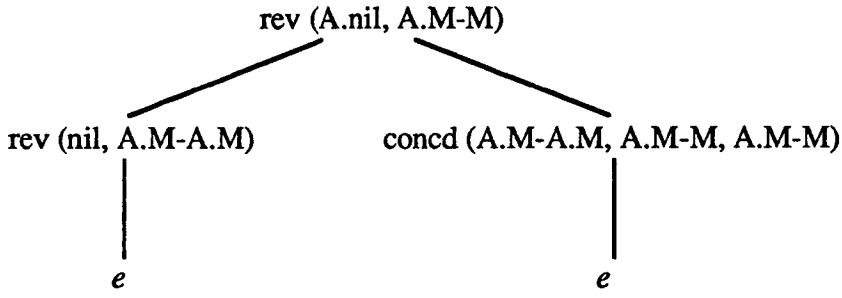

Figure 7

Now a specification is a logical formula interpreted on the canonical term interpretations defined with elements of FUNC, T (section 3.2).

A specification is valid if it is satisfied by all elements of NDOS(P).

A specification is inductive if it satisfies the following formulas on T, the canonical term interpretation :

for every clause c : $n_0(T_0) \leftarrow ..., n_k(T_k), ...$.

AND(for k from 1 to p) $S_k(A_k) \wedge A_0 = mgu((...T_j...), (...A_j...))(T_0) \Rightarrow S_0(A_0)$

Note that only the variables representing attribute occurrences are universally quantified.

Although the RAG constructed for a given program is different from that used in Section 4.2 the same proof method can be still applied: to prove that a specification is valid it suffices to show that it is implied by an inductive one.

We will now consider three applications of this proof method :

- proofs of partial correctness (non deterministic operational properties).
- proofs of completness.

- proofs of run time properties.

4.3.1. Proving properties of the computed answers.

Since RAGO describes the computed answers it is now possible to specify and prove properties of the answers computed for most general atomic goals. In that case the canonical term interpretation is to be used.

Consider for example the program REV (Example 4.2)

rev(nil, L-L).

rev(A.L, R) :- rev(L, Q), concd(Q, A.M-M, R).

concd(L1-L2, L2-L3, L1-L3).

and the specification :

$S2_{rev}$: $\exists\ n \geq 0$ (rev1 = $A_1.A_2. \ldots A_n.nil \wedge$ rev2 = $A_n.A_{n-1}. \ldots A_1.L\text{-}L$)

$S2_{concd}$: concd1= L1-L2 $\wedge$ $concd_2$ = L2-L3 $\wedge$ $concd_3$ = L1-L3

The specification characterizes expected forms of the computed answers for most general atomic goals (here S2 specifies exactly BR_{rev} of RAGO up to a variable renaming). To show that S2 is valid it suffices to prove that it is inductive in RAGO :

The condition holds trivially for the unit clauses.

It suffices then to examine the second clause. We get (using a partially renamed instance of the second clause) :

∂ = mgu((L4,Q,Q, A.M-M, R), ($A_1....A_n.nil$,$A_n....A_1.L\text{-}L$, L1-L2, L2-L3, L1-L3))

hence (with $n \geq 0$)

∂ = L4 $\leftarrow$ $A_1....A_n.nil$, Q $\leftarrow$ $A_n....A_1.A.M\text{-}A.M$, L1 $\leftarrow$ $A_n....A_1.A.M$, L2 $\leftarrow$ A.M,

L $\leftarrow$ A.M, L3 $\leftarrow$ M, R $\leftarrow$ $A_n....A_1.A.M\text{-}M$

hence

$A_0 = \partial((A.L4,R)) = (A.A_1....A_n.nil,\ A_n....A_1.A.M\text{-}M)$ with $n \geq 0$

and $S2_{rev0}$ is satisfied with n'= n+1.

This property does not seem easy to prove manually. However, it appears in practice that simpler properties can be quickly infered using simpler but useful specification. The following property will be used in the sequel (4.3.2) :

$S3_{rev}$: the queue of rev_2 is an uninstantiated variable and does not depend of rev1

$S3_{concd} = S2_{concd}$

The specification says that in every element of NDOS(REV) the queue of the second argument of rev is always a variable (this result will not depend on the nature of the first one - which by the way is always a list).

We show that S3 is inductive hence valid :

This is obvious for the unit clauses.

For the second clause we have :

∂ = mgu((L4,Q,Q, A.M-M, R),(rev_{11}, rev_{12}, L1-L2, L2-L3, L1-L3))

provided that rev_{12} = ...-L, thus Q = ...-L thus L2 = ...-L and L= A.M,

and thus L3 = M and the queue of R is M, a new variable, which does not depend on A.L4.

This property will be used in the next section to prove completness of REV .

The possibility to prove non deterministic operational properties by this type of inductive method seems to be an attractive feature of logic programming. In some sense the method is simple : only one set of variables (i.e. attributes denoting the argument of a non terminal) is needed. Another approach presented in [DrM 87] requires two sets of variables for proving the same kind of properties.

This nice feature has been also noticed in [FLM 88], where NDOS(P) is called the least N-model. However, it should be noted that a N-model is not a model (in the usual logical sense) and that a N-model has only an operational (i.e. algorithmic) definition equivalent to the definition of RAGO.

4.3.2. Proving completeness

A program will be called *complete* with respect to a specification if there exists a proof tree whose labels satisfy the specification. Recall that the specification is a family of formulae in some logical language indexed by the predicate letters of the program. We assume now that the formulae are interpreted on the canonical term interpretation.

Now we demonstrate on an example that the notion of RAGO may be used to prove completeness.

The method presented here consists in finding a subset of NDOS(P) which covers the family of atoms defined by the famulæ.

For example consider the programs plus (section 3.2)

plus (zero, X, X)

plus (s(X), Y, s(Z)) :- plus (X, Y, Z).

We want to prove its completness with regard to the set of atoms : n, m $\geq$ 0 plus (s^n(zero), s^m(zero), s^{n+m}(zero)).(subset of DEN). Tus we prove by induction on the set of atoms that all atoms n $\geq$ 0 plus (s^n(zero), X, s^n(X)) are in NDOS (up to a renaming).

Case : n = 0 obvious by the first clause.

Case : n $\geq$ 0 suppose plus(s^n(zero), X, s^n(X)) is in NDOS, thus plus (s^{n+1}(zero), X, s^{n+1}(X)) is in NDOS.

Obvious by the second clause, using RAGO's definition.

As one way observe by this example the proof method consists in showing that it is possible to build valid trees in RAGO provided the existence of valid subtrees.

We illustrate the power of the method on a more sophisticated example in which we want to prove the existential completness. By *existential completness* we mean that not all arguments are specified by the formulæ, the unspecified being left as different independent variables. Thus we want to prove only that it exists a substitution for these variables such that the resulting atom is in NDOS (thus in DEN).

For example the reverse program of example 4.2 is existentially complete w.r.t. the (sub) set of atoms :

$$n \geq 0 \qquad rev1 = A_1.A_2 \ldots nil$$

Thus we are looking for a substitution σ of R in the atoms rev(A_1 ... A_n.nil, R) such that rev(A_1. .... A_n.nil, σR) is in NDOS (thus in DEN).

To perform the proof, we will prove an other property (combining the atom caracterization and S_3 of section 4.3.1), i.e. :

$n \geq 0$ rev1 = A_1. .... A_n.nil and rev2 = ? - V, V a free variable are in NDOS.

Case : n = 0 obvious by the first clause.

Case : n > 0 we need to unify : (definition of RAGO) :

L	Q	Q	A.M - M	R
A_1. ... A_n.nil	? - V	L_1 - L_2	L_2 - L_3	L_1 - L_3

The unifier exists (as V is a free variable) and is :

$L \leftarrow A_1 \ldots A_n$.nil, $Q \leftarrow$? - V, $L_1 \leftarrow$?, $L_2 \leftarrow$ A.M, $V \leftarrow$ A.M, $L_3 \leftarrow$ M, $R \leftarrow$? - M, M a new free variable.

Thus the following atom is in NDOS :

rev_1 = A.A_1 ... An.nil and rev_2 = ? - M, M a new free variable.

Hence the results of (existential) completness (with n' = n + 1).

This shows an important aspect of proof of completness : all arguments do not need to be specified to prove completness. But combining partial correctness properties and (existential) completness leads to completness. Note that in this method, we do not need to find a decreasing criterion as in the method which we introduce in the next section. This method is a "bottom-up", when the next one is "top down".

Notice that RAGO may be also used for proving incompleteness : if there is no valid tree in NDOS(P) then there is no corresponding proof tree in DEN(P). For example from the fact that the atoms fact(p(X), R) and fact(zero, s(zero)) are not unifiable one can conclude that there is no valid tree (hence no proof tree) for the predicate fact of the program FACT2 (see 4.2) using the second clause, hence NDOS (fact) is a singleton.

4.3.3. Proving run-time properties and using it to prove completness

By run-time properties [DrM 87] we mean properties of selected subgoals during the SLD resolution process. (Notice that the first selected subgoal originates from the initial goal so that the run-time properties concern not only the program but also the initial goal). This notion is similar to the concept of Invariant Search Tree Property presented in [DF 88]. It refers to a specific strategy, i.e. a particular computation rule. We will consider here the standard computation rule, that is the left to right depth first construction of the proof trees.

A specification of a run-time property is a family of formulae indexed by predicate letters. The formulae are interpreted on the canonical term interpretation. The free variables of a formula indexed by p represent arguments of the predicate p at call time. A specification is correct iff at every call the corresponding formula is valid under the valuation of the variables by actual arguments of the call.

If we consider now the non deterministic operational semantics RAGO, the roots of the valid trees are decorated by the instances of most general atomic goals under the computed answer substitutions. Thus, for a given atomic goal, its instance under a computed answer substitution can be obtained by unifying it with the root label of a valid tree of RAGO [Der 88c].

Consider an incomplete proof tree obtained at some step of the computational process, as illustrated below (Fig. 9) :

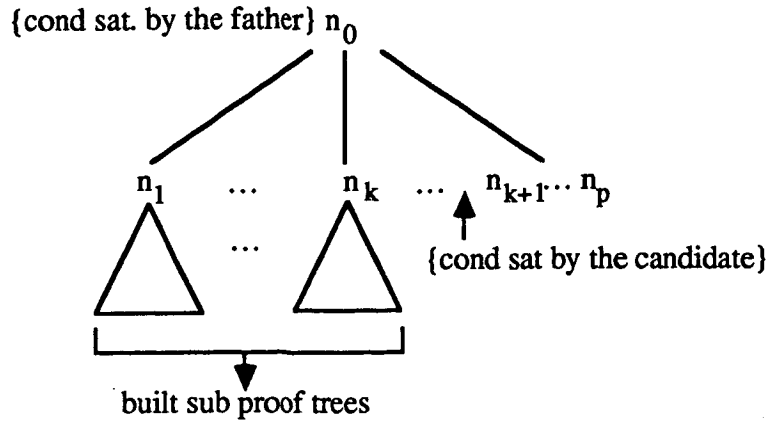

Figure 9

The actual subgoal n_{k+1} originates from a variant of a clause c ; the actual instance of this clause is :

$$n_0 :- n_1 ... n_k ... n_p.$$

The actual form of n_{k+1} (thus any of its properties) is determined by the original form of n_0 at the call time and by the nondeterministic operational semantics of the predicates of $n_1,...,n_k$. Thus to know the form of the subgoal n_{k+1} at call time, it is sufficient to compare n_0 with the current goal (after renaming of the clause c) and to compare the instances of $n_1, ..., n_k$ with their possible non deterministic semantics in NDOS (or some properties). This observation can be used for proving correctness of run-time specifications. It suffices to consider separately each clause of a program, say

$$n_0 :- n_1,...,n_k,...n_p.$$

For every k=1,..., n_p one has to show referring to the nondeterministic operational semantics of the predicates of $n_1,...,n_{k-1}$ that whenever an instance of n_0 satisfies S then the corresponding instance of n_k after solving $n_1,...n_{k-1}$ will also satisfy S. Formal presentation of the method is outside of the scope of this paper. The method resembles that of [DrM 87]. The difference is that it refers to the concept of nondeterministic operational semantics, while the other approach requires another type of specification combining description of run-time properties with the description of properties related to nondeterministic operational semantics.

As an example of use of run-time properties, we prove completeness of the program REV w.r.t goals of the form rev($A_1.A_2....A_n$.nil, Y) using the specification $S3_{rev}$. The idea of the proof consists in showing that it exists an instance of the goal, i.e. of the variable Y, which is a proof tree root by building a valid tree in RAGD. We use run-time property to build partial proof trees. Thus using the standard computation rule we show that the computation process will always terminate successfully.

Unification of the head rev(A.L,R) with such a goal gives the new goal rev($A_2....A_n$.nil,Q) of the same form. Hence the computation will continue preserving the run-time property and it will terminate as the first argument has a strictly decreasing size. Note that failure is not possible since goals of the form rev($A_1.A_2....A_n$.nil,Y) with $n \geq 0$ are always unifiable with one of the clause head of rev.

To complete the proof one may use a well-known property of concd.

It is known that concd is complete (i.e. that there exists a proof tree for some given goal) if a "compatibility condition" is satisfied which says that the queue of concd1 should be unifiable with the first element of concd2.

It remains to notice that after having obtained the proof tree for rev the second argument satisfies $S3_{rev}$, hence the compatibility condition holds for concd. This completes the proof.

Remark : one could be surprised to use an operational property to prove a declarative property like existence of proof trees. This is quite usual in the programming (even logic) activity :

axiomatic view of the clauses guarantees the partial correctness, when procedural view comes as an help (but not as a complete proof in general) to be convinced of the completness of the clause. This semantical duality serves two different aspects of the logic program correctness. But this appears as a trick and one should not forget that completness as stated here is a declarative property, i.e. a property of the denotation.

Conclusion

We presented a grammatical view of logic programming where logic programs are considered grammars. This gives a natural framework for defining extensions to the concept of logic program. The report shows that many useful extensions of Horn clauses incorporated in Prolog without theoretical justification correspond to well-established grammatical concepts. In particular the notion of DCG is a special case of W-grammar, modes are related to dependency relation of AG's, domain declarations of Turbo Prolog can be seen as a metagrammar of W-grammar and Prolog arithmetics fits naturally in the framework of RAG's with non-term interpretations. The grammatical point of view shows also a possibility of further extensions, not incorporated in Prolog, like a natural use of external procedures in logic programs. It also opens for the use of "grammatical techniques" like parsing or attribute evaluation in implementation of logic programs. On the other hand, the comparison of the formalisms shows that resolution techniques can be used for some grammars which were considered practically intractable, like RAG's or W-grammars. Last but not least, the grammatical point of view makes it possible to apply in logic programming some proof techniques developed originally for proving correctness of attribute grammars.

BIBLIOGRAPHY

[Abr 84] Abramson H.: Definite Clause Translation Grammar, International Symposium on Logic Programming, Atlantic City, IEEE, pp 233-240, 1984.

[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.

[Arb 86] Arbab B.: Compiling Circular Attribute Grammars into Prolog, Journal of Research Development, 30 3, pp 294-309, May 1986.

[BM 88] Bonnier S., Maluszynski J.: Toward Clean Amalgamation of Logic Programs with External Procedures, PLILP'88, Orléans, France, May 16-18, 1988.

[Bo 86] Turbo Prolog Owner's Handbook, Borland Int., Scotts Valley, 1986.

[CD 87] Courcelle B., Deransart P.: Proofs of Partial Correctness for Attribute Grammars with Application to Recursive Procedures and Logic Programming, RR I-8702, University of Bordeaux (to appear in Information and Computation 1988).

[CH 87] Cohen J., Hickey T.J.: Parsing and Compiling Using Prolog, ACM Trans. on Progr. Lang. and Systems, 9 2, pp 125-163, April 1987.

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

[Col 78] Colmerauer A.: Metamorphosis Grammar, LNCS 63, pp 133-189, Springer Verlag, (Bolc L. ed.), 1978.

[Der 88] Deransart P.: PROLOG : Basic Concepts and Methodology : Commented Exercises (french), University of Orléans, to appear, 1988.

[Der 88b] Deransart P. : Proof of declarative properties of Logic Programs, INRIA report (to appear).

[Der 88c] Deransart P. : On the Multiplicity of Operational Semantics for Logic Programming and their Modelization by Attribute Grammars. INRIA Report 1988 (to appear).

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

[DF 87b] Deransart P., Ferrand G.: An Operational Formal Definition of PROLOG, comprehensive version. RR 763, INRIA Rocquencourt, December 1987.

[DF 88] Deransart P., Ferrand G.: Logic Programming: Methodology and Teaching, Snd French-Japan Symposium, Cannes, Nov 1987, North-Holland 1988.

[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 84] Deransart P., Maluszynski J.: Modelling Data Dependencies in Logic Programs by Attribute Schemata, INRIA RR 323, July 1983.

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

[Dra 87] Drabent W. : Do Logic Programs Resemble Programs in Conventional Languages, Proc. of IEEE SLP'87, San Francisco, September 1987.

[DrM 87] Drabent W., Maluszynski J.: Inductive Assertion Method for Logic Programs. CFLP 87, Pisa, Italy, March 22-27 1987.

[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.

[Hen 88] Henriques P.R.: A Semantic Evaluator Generating System in Prolog, PLILP'88, Orléans, May 16-18, 1988.

[Hog 84] Hogger C.J.: Introduction to Logic Programming, APIC Studies in Data Processing 21, Academic Press, 1984.

[HS 85] Hsiang J., Srivas M. : Prolog based Inductive Theorem Proving, LNCS 206, Springer Verlag, pp 129-149, Dec. 85.

[JLM 84] Jaffar J., Lassez J.L., Maher M.J. : A theory of complete logic programs with equality, Journal of Logic Programming, 1984/3, pp 211-223.

[Kan 86] Kanamori T. : Soundness and Completness of Extended Execution for Proving Properties of Prolog Programs. TR 175 ICOT, 1986.

[Knu 68] Knuth, D.E. : Semantics of Context-Free Languages, Math. Systems Theory 2 (1968), pp 127-145.

[LBM 88] Leszczylowski J., Bonnier S. and Maluszynski J.: Logic Programming with External Procedures : Introducing S-Unification, IPL 27, 1988, pp 159-165.

[Llo 87] Lloyd J.W.: Foundations of Logic Programming, Springer, 2nd edition, December 1987 (first 1984).

[Mal 84] Maluszynski J.: Towards a Programming Language based on the Notion of Two-level Grammar, TCS 28 (1984) pp 13-43.

[MN 82a] Maluszynski J., Nilsson J.F.: Grammatical Unification, IPL 15 (1982), pp 150-158.

[MN 82b] Maluszynski J., Nilsson J.F.: A Comparison of the Logic Programming Language PROLOG with Two-level Grammars, in : Van Caneghem, ed., First Int. Logic Programming Conf., pp 193-199, Marseille 1982.

[MN 82b] Maluszynski J., Nilsson J.F.: A Version of PROLOG based on the Notion of Two-Level Grammar, in`: J. Komorowski (ed) PROLOG Programming Environments, Proc. of the Workshop, Linköping University 1982

[MTH 83] Matsumoto Y., Tanaka H., Hirakawa and al. : BUP : a Bottom Up Parser embedded in Prolog, New generation Computing, 1, 1983.

[Näs 87] Näslund T.: An experimental implementation of a compiler for two-level grammars, in : Z.W.Ras and M. Zemankova (eds.), Proc. of the 2nd Int. Symposium, Methodologies for Intelligent Systems, pp 424-431, North-Holland 1987.

[Ni 83] Nilsson J.F.: On the Compilation of a Domain-based Prolog, in : R.E.A. Mason, ed., Information Processing 83, pp 293-298, North-Holland 1983.

[Nil 86] Nilsson U., AID : an Alternative Implementation of DCG's, New Generation Computing 4, pp 383-399, 1986.

[PW 80] Pereira F.N.C., Warren D.H.D.: Definite Clause Grammars for Language Analysis : a Survey of the Formalism and Comparison with Augmented Transition Networks. Artificial Intelligence, 13-3, pp 231-278, 1980.

[PW 83] Pereira F.N.C., Warren D.H.D.: Parsing as Deduction, 21st Annual Meeting of the Ass. for Computational Linguistics (Cambridge, Mass.), pp 137-144, June 1983.

[Sie 85] Siekmann J.H.: Universal Unification, in Proc. of the 7th Int. Conf. on Automated Deduction, R.E.Shostak, ed., LNCS 169, Springer-Verlag, 1984.

[UOK 84] Uehara K., Ochitani R., Kakusho O., Toyoda J.: A Bottom Up Parser Based on Predicate Logic : a Survey of the Formalism and its Implementation Technique, ISLP, Atlantic City, pp 220-227, 1984.

[Wij 65] Van Wijngaarden A.: Orthogonal Design and Description of a Formal Language, MR 76, Mathematisch Centrum, Amsterdam, 1965.

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