Relating logic programs and attribute grammars

Pierre Deransart, Jan Maluszynski

▶ **To cite this version:**

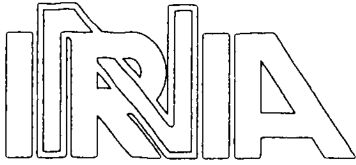
Pierre Deransart, Jan Maluszynski. Relating logic programs and attribute grammars. [Research Report] RR-0393, INRIA. 1985. inria-00076163

HAL Id: inria-00076163
https://hal.inria.fr/inria-00076163

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.



CENTRE DE 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° 393

RELATING LOGIC PROGRAMS AND ATTRIBUTE GRAMMARS

Pierre DERANSART
Jan MALUSZYNSKI

Avril 1985

PROGRAMMATION LOGIQUE ET GRAMMAIRES ATTRIBUEES

(Version révisée de [10])

Pierre Deransart
INRIA
Domaine de Voluceau, Rocquencourt
BP 105 78153 Le Chesnay
France

and

Jan Maluszynski
Department of Computer and Information Science
Linköping University
581 83 Linköping
Sweden

Résumé

Cette étude montre les rapports que la programmation logique et les grammaires attribuées peuvent entretenir. Elle présente des constructions qui transforment des programmes logiques en des grammaires attribuées équivalentes et vice-versa. Il devient alors possible d'appliquer à des programmes logiques des méthodes développées pour les grammaires attribuées. Ces résultats aboutissent à déterminer des conditions suffisantes assurant que lors de la résolution d'un but d'un programme logique aucun terme infini ne sera engendré. De plus on met en évidence une classe non triviale de programmes logiques qui peuvent être exécutés sans faire appel à l'unification dans sa forme la plus générale.

Mots clés = programmation logique, grammaires attribuées, schémas de dépendances d'attributs, analyse du flot de données.

PAPIER RECUPERE ET RECYCLE

RELATING LOGIC PROGRAMS AND ATTRIBUTE GRAMMARS

(Revised version)

Pierre Deransart
INRIA
Domaine de Voluceau, Rocquencourt
BP105 78153 Le Chesnay
France

and

Jan Maluszynski
Department of Computer and Information Science
Linköping University
581 83 Linköping
Sweden

Abstract. This paper shows that logic programs and attribute grammars are closely related. Constructions are given which transform logic programs into semantically equivalent attribute grammars, and vice versa. This opens for application in logic programming of some methods developed for attribute grammars. These results are used to find a sufficient condition under which no infinite term can be created during a computation of a logic program, and to define a nontrivial class of logic programs which can be run without employing unification in its general form.

Keywords: logic programming, attribute grammars, attribute dependency scheme, data flow analysis

1. Introduction

The aim of this paper is to study relationships between two computational formalisms which have been independently developed with different motivations: the attribute grammars introduced in [14] and the Horn clause logic considered as a programming language [15]. We show that these formalisms are closely related to each other and we discuss a possibility of transforming logic programs into attribute grammars and vice versa. This makes it possible to use methods of one of the formalisms in solving problems related to the other. We give a few examples of such applications and we hope that there are many others.

The paper is organized as follows. Section 2 gives necessary notions concerning attribute grammars and definite clause programs. Section 3 gives constructions relating attribute grammars and logic programs and shows differences between the formalisms. The constructions presented may have different applications. In Section 4 we present examples of applications of one of the constructions. The examples concern the occur-check problem, data-driven evaluation, and unification-free evaluation of definite clause programs.

This paper differs essentially from its original version [10]. In particular, in the present version, the basic concepts are introduced in more structured manner that improves the conceptual clarity of the presentation and makes it possible to simplify the original constructions. Furthermore, we discuss in greater detail the problem how to transform a logic program into a semantically equivalent attribute grammar.

2. Preliminaries

2.1. Attribute Grammars and Attribute Dependency Schemes

The formalism of attribute grammars has been introduced for assigning semantic values to the nodes of derivation trees of a context-free grammar. With every symbol X of the grammar a finite set of *attributes* is associated, denoted *Attr(X)*. The cardinality of this set will be denoted n_X . The attributes are names of the semantic values to be associated with any derivation tree node labeled X. Each of these nodes has attached n_X places to which one assigns values from some fixed semantic domains. We call these places attribute occurrences or *positions*. In this paper we assume without loss of generality that only the nonterminals of the grammar may have attributes, but not the terminal symbols. To deal with attributes under this assumption it suffices to consider only abstract syntax trees, not including the terminal symbols. Following [5] we base our definition of attribute grammar on the notion of abstract context-free grammar. It is a pair $< N, P >$ where N is a finite set of nonterminals and P is a finite set of context-free productions over

N. In this way we follow the algebraic view of a context-free grammar as a many-sorted algebra (see e.g. [6]). However, in this paper we do not use the algebraic terminology for that purpose.

Let *p* be a production in *P* of the form

$$X_0 \rightarrow X_1, \dots, X_n$$

For each $i = 0,\dots,n$ and each *a* in $Attr(X_i)$, we introduce a new symbol $a(i)$ called an *occurence* of the attribute *a* in *p*. The set of all such symbols is denoted $Pos(p)$. Its elements are also called *positions* of *p*.

We want to specify the semantic values to be assigned to the positions of a given derivation tree. Each derivation tree consists of instances of the production rules of the grammar. The idea of an attribute grammar is to associate with each production rule a restriction on the semantic values which should be observed by every occurrence of this production in any tree. To formulate such restrictions we introduce a logical language. Since it is assumed that different attributes may range over different domains we take a many-sorted approach.

Let *S* be a set of *sorts*. The *S*-sorted language consists of formulas constructed in the usual way from atomic formulas by means of logical connectives (possibly including quantifiers). The atomic formulas are constructed from variables, functors, and predicate letters.

The set *V* of variables is sorted: it is the union of the family of disjoint sets $\{V_s\}_{s \in S}$.

The set *F* of functors is typed: each functor has associated a pair (σ, s), $\sigma \in S^*$, $s \in S$, called its *type*. If for some *f* in *F* the string σ is empty *f* is called a *constant* of sort *s*.

The set *R* of predicate letters is typed: each predicate letter has associated a string σ in S^*, called its *type*.

To define the syntax of atomic formulae we refer to the notion of term. The terms are sorted.

A *term of a sort s* is defined as follows:

1. Every variable in V_s is a term of the sort *s*;
2. Every constant of sort *s* is a term of the sort *s*;
3. If $(s_1 \dots s_n, s)$ is the type of a functor *f*, ($n>0$) and t_i is a term of sort s_i for $i = 1,\dots,n$ then $f(t_1,\dots,t_n)$ is a term of the sort *s*;
4. Nothing else is a term of the sort *s*.

If *S* is a singleton this definition reduces to the usual case of one-sorted term.
The set of atomic formulae consists of all strings of the form $r(t_1,\dots,t_m)$, where *r* is a predicate letter of the type $s_1 \dots s_m$, and for $i = 1,\dots,m$, t_i is a term of the

sort s_i.

To define *an interpretation* $\Im$ of an S-sorted logical language we proceed as follows.

For each sort s in S we define a semantic domain D_s.
With each functor f of the type $(s_1...s_n, s)$ we associate an operation from $D_{s_1} \times ... \times D_{s_n}$ into D_s.
With each predicate letter r of the type $s_1...s_m$ we associate a subset of $D_{s_1} \times ... \times D_{s_n}$.

We extend $\Im$ for the formulae of the language in the usual way, using the notion of *assignment*. An assignment α is a mapping of the variables in V into the domains of the corresponding sorts. We extend it to terms and tuples of terms in the following way:
Let t be a term of the form $f(t_1,...,t_n)$, where $t_1,...,t_n$ are terms and f is a functor, and let $f_\Im$ be the function assigned to f in $\Im$. Then $\alpha(f(t_1,...,t_n))$ is defined to be $f_\Im(\alpha(t_1),...,\alpha(t_n))$.
Let x be the n-tuple of terms $<t_1,...,t_n>$. Then $\alpha(x)$ is defined to be the n-tuple of values $<\alpha(t_1),...,\alpha(t_n)>$.

We adopt the usual notion of validity of a formula for a given interpretation and a given assignment of its free variables.

In the sequel we assume that an S-sorted logical language L is given. We will use L to formulate restrictions on attribute values associated to the nodes of derivation trees. For this we will use attribute positions as variables. Given a set W of S-sorted variables we will denote by $L(W)$ the set of all formulae whose free variables are in W.

Relational Attribute Grammars

We introduce first a general conceptual framework for defining more restricted types of attribute grammars used in existing implementations. We adopt here essentially the definition given in [5].

Definition 1 : Relational Attribute Grammar (RAG)

A *RAG* is a 5-tuple $G = < N, P, Attr, R, \Im >$ where:

(1) $< N, P >$ is an abstract context free grammar ,

(2) *Attr* is the union of a family of finite sets of attributes $\{Attr(X)\}_{X \in N}$ and every attribute a in *Attr* has a sort $sort(a)$ in some set of sorts S.

(3) $R = \{ R_p \}_{p \in P}$, where R_p is a formula of the language $L(Pos(p))$ (that is, the only free variables of R_p are the attribute positions of p); The formulae are called *semantic rules* of G;

(4) $\Im$ is an interpretation of the S-sorted language L.

The semantics of such an object is formally defined in [5]. It is based on the notion of decorated derivation tree. Derivation trees are constructed by "pasting together" instances of production rules of P. For a given tree T one can enumerate its nodes. We shall assume that such an enumeration is given. By a ***position of a tree*** T we mean any pair $a(k)$ such that a is an attribute of a nonterminal X and k is the number of a node of T labeled by X. The set of all positions of T is denoted $Pos(T)$. By an ***instance*** of a production rule p in T we mean any subtree t of T consisting of a node u and all its sons $u_1, \ldots, u_m$ and originating from p. Clearly, there is a one-one correspondence between the positions of p and the positions of the subtree t. This is illustrated in Fig.1. The subtrees t_1 and t_2 are occurrences of the production rule p in the derivation tree T. The attribute position $a(n+2)$ of T is a position both of t_1 and of t_2. If considered as a position of t_1 it corresponds to the position $a(2)$ of p. Otherwise, if considered as a position of t_2 it corresponds to the position $a(0)$ of p.

A ***valuation*** of a tree T is a function from the positions of T to values in the corresponding semantic domains. A valuation is ***valid*** iff for each occurrences of any production rule p in the tree T the formula R_p is valid under this valuation. A finite complete tree T with a valid valuation will be called a ***decorated*** tree.

Now, a given RAG $G = < N, P, Attr, R, \Im >$ can be considered to be a specification of the set of all decorated trees T of the abstract context free grammar $< N, P >$. Note that R and $\Im$ may be such that the set of the decorated trees of a RAG is empty.

This definition of the semantics differs slightly from that in [5]. The latter uses the notion of decorated tree to associate relations with the nonterminals of the grammar. For the purposes of this paper it is more convenient to refer directly to the decorated trees.

Example 1
To illustrate the formalism we use it to describe the behavior of a reversible counter. The counter accepts sequences of input signals. At each step of its operation its state may be characterized by a natural number. There are two input signals: *increase,* denoted i, and *decrease,* denoted d which cause corresponding changes of the state. The behavior of the counter is characterized by associating with each sequence of input signals a binary relation on the states of the counter describing the global change of the state caused by this sequence.

The sequences can be described by the following grammar:

(p_0) $X \to \epsilon$
(p_1) $X \to IX$
(p_2) $X \to DX$
(p_3) $I \to i$
(p_4) $D \to d$

The nonterminal X is used to derive sequences of signals. Its particular instance in a derivation tree corresponds to a particular sequence of signals. Therefore we associate with X a pair of attributes whose values describe, respectively, the initial state (*initial*) and the final state (*final*) of the counter when the sequence of signals derived from X is accepted. We assume that the initial state is determined by some outside factors, and we relate the final state to the given initial one and to the sequence of signals derived from X.

Now we relate the attribute instances in the production rules of the grammar.

For p_0 the derived sequence of signals is empty. We assume that the initial state remains unchanged when such a sequence is accepted. Formally, using the notation for representing the attribute positions of the productions, we express this condition as the following semantic rule

(R_0) $final(0) = initial(0)$

For any instance of the production rule p_1 the sequence X_0 derived from its left-hand side is the sequence X_1 derived from the nonterminal X of its right-hand side preceded by the nonterminal I generating the increase signal. Thus, the initial state of X_1 is that obtained from the initial state of X_0 by accepting the increase signal, while the final states of both sequences are the same. In our notation we express this as follows:

(R_1) $initial(2) = f_i(\ initial(0)\ ) \quad \wedge \quad final(0) = final(2)$

where f_i is a function describing the effect of the signal i, and 0 and 2 are numbers of the nonterminals in the production p_1.

Similarly for p_2 we have

(R_2) $initial(2) = f_d(\ initial(0)\ ) \quad \wedge \quad final(0) = final(2)$

where f_d is a function describing the effect of the decrease signal.

The semantics of that example depends on the meaning we attach to the functions f_i and f_d . For example, f_i may be defined to be the successor function, and f_d the (partial) predecessor function. In this case the RAG defined above relates the final state to the initial state for any sequence of signals which is acceptable for the initial state. For example for initial value 10 (value of the attribute *initial* of the root) and final value 8 (value of the attribute *final* of the root), the sequence *iddd* is acceptable, as shown in Fig. 2 . On the other hand, it is not acceptable for initial value 1. Since the predecessor of 0 is undefined, thus the truth value of R_1 is also undefined. Hence for this

initial value there is no valid valuation for the considered tree.

In terms of Definition 1 this example describes the RAG
$G = < N, P, Attr, R, \Im >$ where

$N : \{ X, I, D\}$

$P : \{ X \rightarrow \epsilon , X \rightarrow IX , X \rightarrow DX \}$

$Attr : Attr(X) = \{ initial, final \}$ $Attr(I) = Attr(D) = \{ \}$

$R : \{ R_0 , R_1 , R_2 \}$

$\Im$ is defined as follows:

$D_{initial}$ and D_{final} is the domain of natural numbers,

f_i is the successor function,

f_d is the predecessor function,

$=$ is the equality on natural numbers.

The grammar specifies the set of all decorated trees which can be obtained from the derivation trees by finding decorations satisfying the formulae associated with the occurrences of the production rules. An example of a decorated tree is given in Fig. 2.

End of Example 1

Functional Attribute Grammars

The relational attribute grammars, as defined above, 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 will be discussed in Section 3.

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. In this section we formulate these restrictions in order to be able to refer to the attribute evaluation problem. Most of the results obtained in the field of attribute grammars have been motivated by this problem and we claim that some of these results may be applied in the field of logic programming. (In section 4 we show some applications reinforcing that claim). It seems also probable that some results obtained in logic programming could be applied in the field of attribute grammars.

The restrictions we are going to formulate concern the form of the formulae in R. Assume that for each production rule p the semantic rule R_p is a conjunction of formulae of the form

$$x = \mathcal{T}_x$$

where x is a position of p and $\mathcal{T}_x$ is a term of the sort *sort(x)* (such formulae will be called *semantic definitions*). Knowing values of the attribute positions in $\mathcal{T}_x$ one can easily compute the value of x. One can require that any position in any derivation tree can be computed in that way, and that each position is

defined by only one definition of this type. To formulate this restriction in a systematic way we introduce an auxiliary notion of *splitting* of the attribute set *Attr*. A splitting is defined by specifying disjoint sets: *Inh* (of *inherited* attributes) and *Syn* (of *synthesized* attributes), such that $Attr = Inh \cup Syn$. Thus for all X in N $Attr(X) = Inh(X) \cup Syn(X)$ with empty intersection.
Let p be a production rule of the form

$$X_0 \rightarrow X_1 \ldots X_n$$

A given splitting of the attributes induces a splitting of the positions of p into the *input* positions defined as follows:

$$Input(p) = \{\ a(i) \mid a \in Inh(X_0) \text{ or } a \in Syn(X_i) \text{ and } i>0\ \}$$

and the *output* positions defined as follows

$$Output(p) = \{\ a(i) \mid a \in Syn(X_0) \text{ or } a \in Inh(X_i) \text{ and } i>0\ \}$$

Clearly

$$Pos(p) = Input(p) \cup Output(p)$$

The intuition behind the splitting of the attributes concerns the intended organization of the process of computing attribute values for a given derivation tree. As mentioned above, each formula R_p will be restricted to be a conjunction of semantic definitions, and each of the definitions will be used to compute the attribute value of one attribute position of p. Consider a derivation tree of a given attribute grammar. Any inner node of the tree is shared by two different instances of some production rules within the tree. We call them, respectively, the *upper* production rule, and the *lower* production rule of the node. For example, the node $n+1$ of the tree in Fig.1 occurs in the subtrees t_1 and t_2 which are different instances of the production rule p, which is in the same time the upper production rule and the lower production rule of the node. As mentioned above, the value of a position of the shared node is to be defined either by the semantic rule of the upper production, or by the semantic rule of the lower production, but not by both of them. In our example the only attribute position of the shared node is $a(n+1)$, which in t_1 corresponds to the position $a(2)$ of p, and in t_2 to the position $a(0)$ of p. In this case the requirement means that the semantic rule of p includes either a semantic definition of $a(0)$ or a semantic definition of $a(2)$, but not both of them.

A given attribute splitting is a pattern for writing semantic rules satisfying the requirement. Let X be the nonterminal labeling an inner node of a derivation tree. Let p' and p'' be, respectively, the upper production and the lower production of this node. The attribute positions of this node corresponding to the inherited attributes of X are to be defined by the semantic rule of p', and the others (i.e. those corresponding to the synthesized positions of x) by the

semantic rule of p". Thus, the semantic rule of a production p consists of the definitions of all positions which are output positions under a given splitting. The input positions are not defined in the semantic rule since their values will be determined by the outer context. However, sometimes it may be covenient to state explicitly some conditions on input values. We allow them to be included in the semantic rule.

As discussed above the notion of splitting may be considered a tool for writing RAG's with functional dependencies between attribute positions. The splitting can be deduced from the form of the semantic rules of a RAG written in this way. Therefore it is not necessary to give it explicitly with the definition of the RAG.

The intuitions discussed above are reflected in the following definition.

Definition 2 : Functional Attribute Grammar (FAG)

A *Functional Attribute Grammar* is a RAG $G = < N, P, Attr, R, \Im >$ such that

for each p in P the formula R_p is of the form

$$\Gamma_p \wedge C_p$$

where

(1) $\Gamma_p = \wedge_{x \in Output(p)} x = \mathcal{T}_x$

and $\mathcal{T}_x$ is a term with variables in $Pos(p)$ not including x,

(2) $C_p \in L(Input(p))$

(that is the variables of C_p are input positions of p)

(3) *Output(p)* and *Input(p)* are determined by some splitting of the set *Attr*.

Notice that any splitting determines in a unique way the number of conjuncts of Γ_p. Thus, when writing an attribute grammar it may be helpful to choose first a splitting and to use it for developing the formulae in a more systematic way.

The component C_p of each R_p makes it possible to express some conditions concerning values of the input positions. If the condition is *true* it may be omitted.

Example 2

It is easy to see that the RAG given in Example 1 is a FAG with the splitting $Inh(X) = \{initial\}$, $Syn(X) = \{final\}$.

The formulae include no conditions.

End of Example 2

If all positions used in the the right-hand sides of the semantic definitions of an FAG are input positions, the FAG is said to be *normalized* . It is the case in Example 2.

We are now able to state the *attribute evaluation problem.* For simplicity we will consider, without loss of generality, that the FAG has no condition. Such attribute grammars will be called *pure* FAG's.
According to our restriction any attribute position in a given derivation tree is defined only by one formula referring to the values of some neighboring positions. In this way the attribute positions within the tree depend on each other and the question arises in which order they should be computed. This problem can be formulated as follows.

The Evaluation Problem

Given a derivation tree T find a total order on $Pos(T)$ such that for any $x \in Pos(T)$ its value is determined by definitions including only positions of T less than x in this order.

If such a total order is known it can be used for sequencing computation of the attribute values of a given tree. Fig.3 shows a total order on the positions of a derivation tree of the FAG of Example 2. The order reflects the dependencies between the positions induced by the semantic definitions associated with that tree. We give now a formal definition of the dependency relation determined by a FAG on the positions of any derivation tree.

Consider a FAG $G = < N, P, Attr, R, \Im >$. An output position r of p *depends* on a position q of p iff q occurs in the right-hand side of the definition of r . This will be denoted $q \rightarrow_p r$.
The family of the relations $\rightarrow_p$ for $p \in P$, associated with a given attribute grammar G, will be denoted D_G.

Let T be a derivation tree and let t be an occurrence of a production p in T. As it was mentioned above there exists a one-one correspondence between the positions of p and the positions of the subtree t. This makes it possible to define a dependency relation $\rightarrow_t$ on the positions of the subtree t:
$q \rightarrow_t r$ iff $q' \rightarrow_p r'$ where q' and r' are the positions of p corresponding to q and r.

Each derivation tree T can be in a unique way decomposed into a finite number of instances of the production rules. Denote by P_T the family of instances of the production rules from which T is composed. Observe that some elements of P_T have common nodes. This makes it possible to define the dependency relation $\rightarrow_T$ on the positions of T: it is defined as the union of all relations $\rightarrow_t$, for t in P_T . The total order of Fig. 3 is the transitive closure of

the dependency relation of a derivation tree of the FAG of Example 2.

A FAG is said to be ***well-formed*** iff for every derivation tree T of the underlying CF-grammar the transitive closure of the relation $\rightarrow_T$ (denoted $\rightarrow^+_T$) is a partial order.
It is not difficult to see that a functional attribute grammar is well-formed iff there exists an order of evaluation for any derivation tree [14,11] . The well-formedness property is statically decidable, but it is of exponential complexity. However, several subclasses of FAG's have been discovered whose well-formedness can be tested in polynomial time [6,12] .
Most of the literature on attribute grammars is devoted to the attribute evaluation problem. When studying this problem abstractly one usually refers only to properties of the dependency relation rather than to concrete semantic rules defining this relation. To specify the dependency relation in an abstract way one can use a notion of attribute dependency scheme. It is similar to a FAG but it includes neither formulae nor interpretation. Instead the local dependency relations on the attribute positions of the production rules are explicitly defined (the definition is essentially the same as in [5]).

Definition 3 : Attribute Dependency Scheme (ADS)

An *ADS* is a 4-tuple $S = < N, P, Attr, D >$ where

(1) $< N, P >$ is an abstract context free grammar.

(2) *Attr* is a set of attributes with a given splitting.

(3) D is a family of binary relations $\{ D_p \}_{p \in P}$ defined on $Pos(p)$, such that

$$\{x \mid yD_p x \text{ for some } y \in Pos(p) \} \subseteq Output(p)$$

Since an ADS has no semantic rules the attribute splitting cannot be deduced and must be given explicitly. The condition (3) reflects the fact that an ADS is an abstraction of a FAG; none of the input positions of a production rule depends on a position of this rule.
If

$$\{y \mid yD_p x \text{ for some } x \in Pos(p) \} \subseteq Input(p)$$

then the ADS is said to be ***normalized*** . Since an ADS determines local dependency relations on the positions of the production rules the concept of well-formedness introduced for FAG's can be also used for ADS's..

Clearly, the local dependency relations on *Pos(p)* defined by a FAG can be used to construct an ADS with the same production rules and attributes. In this way one can characterize the sequencing restrictions which should be satisfied by any attribute evaluation order of the FAG without referring to the semantic definitions.

Basic Term Interpretations of pure FAG's

We outline now a possibility of "symbolic" attribute evaluation for pure and well-formed FAG's. The idea consists in representing attribute values by terms constructed from the functors of the semantic definitions [6]. Thus the evaluator would not make use of the interpretation associated with the FAG.

Let p be a production of a pure and well-formed FAG. The value of each output position of p is determined by a term. The only variables of this term are the input positions of p on which the output position depends. If p occurs in a derivation tree T each input position of this occurrence is either an output position b of an instance of a production rule p', or it is a minimal element of the dependency relation of T. In the first case the value of b is determined by a term whose only variables are some input positions of p'. This term can be substituted for b in all terms representing the values of the output positions of p. In this way b can be eliminated from these terms. This process can be repeated for all input positions of all occurrences of the production rules in T. Since the FAG is well-formed the value of each attribute position of T will be finally represented by a term whose only variables are minimal positions of the dependency relation of T. For example for the derivation tree in Fig. 3 the value of the attribute *final* of its root is represented by the term

$$f_d(f_d(f_d(f_i(initial(0)))))$$

where *initial(0)* is the input attribute position of the root. This value can be obtained by evaluation of the term given above under the interpretation of the FAG. However, to find the term we don't use the interpretation.

As a matter of fact the terms constructed as described above are attribute values in an interpretation $\mathfrak{I}$ defined as follows:

It has only one domain consisting of the terms which can be constructed from the functors occurring in the semantic definitions of the FAG and from variables.

With each n-ary functor f it associates the n-ary operation which for given terms $t_1,...,t_n$ produces the term $f(t_1,...,t_n)$.

With the only predicate letter $=$ it associates the identity relation on terms.

The interpretation defined above for a given FAG will be called its ***basic term interpretation***.

2.2. Definite Clause Programs

The idea of logic programming concerns computing relations specified by logic formulae. This section outlines briefly this idea and stresses mainly the notions which are used in the sequel for relating logic programs and attribute grammars. For more details the reader is referred to the literature on logic

programming (e.g. [2],[4],[15]).

The syntax of Definite Clause Programs

We focus our attention on a special type of logic formulae called *definite clauses.* According to [2] a *definite clause* is a pair consisting of an atomic formula A and a finite set of atomic formulae $\{B_1, ..., B_q\}$, $q \geq 0$, written as

$$A <- B_1, ..., B_q.$$

The atomic formulae are constructed, as usual, from predicate letters and (one-sorted) terms: A is an atomic formula iff it is of the form $P(t_1, ..., t_n)$ where P is a n-ary predicate and $t_1, ..., t_n$ are terms.

A definite clause of the form described above can be represented in the standard logic notation as the following formula

$$\forall x_1 ... \forall x_k \, (B_1 \wedge ... \wedge B_q \rightarrow A)$$

where $x_1 ... x_k$ are all variables occurring in the clause.
An atom (a term) is said to be ***ground*** if it has no variables.

Definition 4 : Definite Clause Program (DCP)

A *DCP* is a triple $C = < \mathcal{N}, \mathcal{F}, \mathcal{P} >$ where

$\mathcal{N}$ is a finite set of predicate letters with assigned arities,

$\mathcal{F}$ is a set of functors with assigned arities,

$\mathcal{P}$ is a finite set of definite clauses constructed with $\mathcal{N}$ and $\mathcal{F}$.

The semantics of Definite Clause Programs

Usually a DCP is considered a specification of its least Herbrand model (see e.g. [2]). It was shown in [4] that one can deal instead with the set of all atoms (not necessarily the ground ones) which are logical consequences of the clauses of the DCP. Each element of this set can be obtained by constructing a ***proof tree***. For the purposes of this paper it is convenient to consider a DCP to be the specification of the set of all proof trees.

We introduce now some auxiliary notions and the notion of proof tree.
A ***substitution*** is an operation on expressions (terms, formulae), which replaces all occurrences of a variable in an expression by a term. The result is called an ***instance*** of the expression. A substitution θ is called a ***unifier*** of expressions e_1 and e_2 iff $\theta(e_1) = \theta(e_2)$.

Definition 5 : Proof tree

A ***proof tree*** is an ordered labeled tree whose labels are atomic formulae (possibly including variables) or are empty. The set of the proof trees of a given DCP C is defined as follows:

1. If $A <- .$ is an instance of a clause of C then the tree consisting of two vertices whose root is labeled A and whose only leaf has the empty label

is a proof tree.

2. If $T_1, \ldots, T_q$ for some $q>0$ are proof trees with roots labeled $B_1,\ldots,B_q$ and if A <- $B_1, \ldots, B_q$, is an instance of a clause of C, then the tree consisting of the root labeled with A and the subtrees $T_1, \ldots, T_q$ is a proof tree.
3. Nothing else is a proof tree.

By a *partial proof tree* we mean any finite tree constructed by "pasting together" instances of clauses. Thus a proof tree is a partial proof tree whose all leaves have empty labels.
If θ is a substitution and T is a partial proof tree we denote by $\theta(T)$ the proof tree obtained from T by replacing each of its labels L by $\theta(L)$. It is called an *instance* of T. An instance of T is ground if all its labels are ground.

Example 3
Consider the following DCP

add(0,x,x) <- .
add(s(x),y,s(z)) <- *add(x,y,z)*.

A partial proof tree and a proof tree of that DCP are shown in Fig. 4 .

End of Example 3

Computations of a DCP

Logic programming systems make it possible to construct proof trees of a given DCP. To start a computation of a DCP one has to submit a finite set of atoms, called a *goal*. A goal may be considered a clause with the empty left-hand side. To simplify our constructions we shall assume that the goal is an additional clause whose left-hand side is a special nullary predicate *goal*, which does not occur in the clauses of the program. A DCP with a goal clause will be called an *augmented* DCP and denoted $<C,g>$, where

$$g : goal <- B_1,\ldots,B_q.$$

is the additional clause. The task of the computation is to find a substitution under which all atoms of the goal become logical consequences of the clauses of C. This can be achieved by constructing a proof tree. Backtracking is used to find different substitutions.
In logic programming systems the proof trees are constructed in a descendant manner, starting with the goal clause and using resolution [15] to construct subsequent partial proof trees. The reader is assumed to be familiar with that principle. The partial proof trees constructed during that process will be called *resolution trees.*

Example 4
Consider the DCP of example 3. The trees in Fig. 4 can be obtained from

resolution trees of the goal

$$goal <- add(s(0), x, z).$$

by removing the roots of the latter (which are labelled *goal*).

End of Example 4

3. Relating Definite Clause Programs and Attribute Grammars

The proof trees of a DCP and the decorated trees of a RAG have a similar structure: the predicate symbols of a DCP play the role of nonterminals while terms are its "semantic values". Indeed, every predicate symbol has a fixed arity, hence a fixed number of positions, which within labels of a proof tree may be instantiated to different terms. Thus there is a direct correspondence between both formalisms.

In this section we show that a DCP can always be considered as a RAG, and that, if an additional information, called d-assignment, which is comparable to the attribute splitting, is provided, one can sometimes transform it into a semantically equivalent FAG. Thus for studying properties of DCP's and for constructing their proof trees one can make use of the methods developed for attribute grammars. To make this statement more precise we describe some constructions which transform DCP's into attribute grammars. We discuss also the problem whether an attribute grammar can be transformed into an equivalent DCP.

3.1 Transforming DCP's into RAG's

In this section we describe a construction which transforms any DCP into an equivalent RAG. We comment first on the logical language L used to express the semantic rules of the resulting RAG. We construct L as a one-sorted language. For each clause c of the original DCP we introduce a separate n_c-ary predicate symbol r_c, where n_c is the number of all argument positions of c. Thus, if c is of the form

(*) $\quad p_0(t_{00},\dots,t_{0m_0}) <- p_1(t_{10},\dots,t_{1m_1}),\dots, p_n(t_{n0},\dots,t_{nm_n}).$

where $p_1,\dots,p_n$ are predicate symbols and $n \geq 0$, then $n_c = m_0 + m_1 + \dots + m_n$.
The set of functors is defined to be empty. Thus, L consists of the formulas constructed in the usual way from the predicate symbols and from variables.

Now we define the construction.

Construction 1

Given a DCP $C = <\mathcal{N}, \mathcal{F}, \mathcal{P}>$ we construct a RAG $G_C = < N, P, Attr, R, \mathfrak{I} >$ defined as follows:

1. $N = \mathcal{N}$.

2. For every clause c of $\mathcal{P}$ of the form (*) we construct the production rule

$$p_c : p_0 \rightarrow p_1 \ldots p_n$$

The set P of the production rules of G_C consists of all production rules constructed in this way; we distinguish between different copies of a production rule which might have been obtained from different clauses.

3. The set *Attr* is defined as follows:

For each predicate symbol q (i.e. for each nonterminal of G_C) the number of attributes n_q equals its arity. The attributes are names of the positions of the predicate symbols. They are constructed from the predicate symbols followed by natural numbers. Thus

$$Attr(q) = \{qi \mid i = 1,\ldots,n_q\}$$

The attribute set is one-sorted. The total ordering on $Attr(q)$ induced by the numbers of the attributes will be denoted $\lhd$. We use it to define a total order $\preceq$ on $Pos(p_c)$:

$$a(i) \preceq a'(j) \text{ iff } i<j \text{ or } i=j \text{ and } a \lhd a' \text{ in } Attr(q).$$

Attr is the union of the sets $Attr(q)$ for $q \in \mathcal{N}$.

4. For each production rule p_c the following formula R_c is created:

$$r_c(x_1, \ldots x_{n_c})$$

where

r_c is the predicate symbol of L corresponding to the clause c

$\{x_1,\ldots,x_{n_c}\} = Pos(p_c)$ and $x_i \preceq x_j$ for $i<j$.

5. The interpretation $\Im_C$ is defined as follows:

Its domain is the language of all terms constructed from the functors of $\mathcal{F}$ and from variables.

The relation associated with each letter r_c is defined as follows.

Let $<t_{00},\ldots, t_{0m_0}, t_{10},\ldots,t_{1m_1}, \ldots, t_{n0},\ldots,t_{nm_n}>$ be the n_c-tuple consisting of the terms occurring in the clause c taken in their textual order. Then the relation r_c is defined to be the set of the images of this tuple under all possible assignments of its variables in the semantic domain (thus, it is a relation on terms).

End of Construction 1

Example 5

We use Construction 1 for the DCP of Example 4 . The underlying context-free production rules are

(p_1) $\quad add \rightarrow \epsilon$

(p_2) $\quad add \rightarrow add$

where *add* is the only nonterminal. The language defined by these rules consists of the empty string and the second rule makes the grammar ambiguous. However, the grammar is not used for generating strings but rather

for producing derivation trees. The nonterminal ***add*** has 3 attributes, denoted in the sequel ***add1, add2,*** and ***add3.***

The set of the semantic rules consists of the formulas

$R_1 : r_1 \; (\; add1(0) \; , \; add2(0) \; , \; add3(0) \;)$

$R_2 : r_2 \; (\; add1(0) \; , \; add2(0) \; , \; add3(0) \; , \; add1(1) \; , \; add2(1) \; , \; add3(1) \;)$

The relations associated with the symbols r_1 and r_2 by the interpretation are characterized by the following tuples of terms

$r_1 : < 0 \, , x \, , x >$

$r_2 : < s(x) \, , y \, , s(z) \, , x \, , y \, , z >$

End of Example 5

We use Construction 1 to relate DCP's and RAG's:

Theorem 1

Let C be a DCP and let G_C be the RAG obtained from C by Construction 1. The set of the proof trees of C is isomorphic with the set of the decorated trees of G_C .

Proof

We show that there is one-one correspondence between the decorated trees of G_C and the proof trees of C.

We show first how to transform the decorated trees into the proof trees. Let T be a complete derivation tree of the RAG with a valid valuation. Thus, each node x of T is labeled by a predicate symbol p_x and has $m{=}n_{p_x}$ attribute positions $\{ a_1,...,a_m \}$ to which some terms $t_1,...,t_m$ are assigned. Assume that $a_i \lhd a_j$ for $i \leq j$, *where* $\lhd$ is the ordering on $Attr(p_x)$ defined in Construction 1. Since the attribute values satisfy the semantic constraints, then the tree T' obtained by replacing the label of each node x of T by the atom $p_x(t_1,...,t_m)$ is a proof tree of G_C .

To transform a proof tree T of C into a decorated tree T'' of G_C we proceed as follows. Let x be a node of T. By the definition it is labeled by an atomic formula of the form $p(t_1,...,t_n)$, where p is an n-ary predicate letter and $t_1,...,t_n$ are terms. To obtain T'' we replace the label of x by the letter p and we associate with x the attribute positions $p1(x),...,pn(x)$, where $pi(x) \preceq pj(x)$ for $i \leq j$. To each attribute position $pi(x)$, $i{=}1,...,n$ we associate as its value the term t_i. In this way we define the valuation of T''. Since by the definition T consists of instances of clauses of C, then the valuation is valid.

End of Proof

3.2 Modelling data dependencies of a DCP by Attribute Dependency Schemata

The notion of dependency relation for FAG's makes it possible to prove properties of FAG's and to organize better attribute evaluation [11]. To define this relation one chooses an attribute splitting. This does not influence the declarative semantics of the FAG.

In this section we follow the example of FAG's to introduce a notion of dependency relation for DCP's. Its usefulness is demonstrated in Section 4 by a number of applications. Essentially, the relation is defined on the argument positions of the labels of a proof tree of a DCP. Formally, to define it we transform a given DCP into an Attribute Dependency Scheme. For this we have to define a notion of dependency between positions of a clause. Intuitively, positions depend on each other only if they share a variable. However, this relation is symmetric. To make it into an ordering relation we split positions of the predicate symbols of the DCP into "synthesised" and "inherited" ones.

Definition 6 : d-assignment

Given a DCP $C = < \mathcal{N}, \mathcal{F}, \mathcal{P} >$, a *direction assignment* or briefly *d-assignment* is a mapping of the arguments of each predicate symbol of $\mathcal{N}$ into the set $\{ \downarrow, \uparrow \}$.

By analogy with functional attribute grammars, we will call an argument assigned to $\downarrow$
(resp. $\uparrow$) "inherited" (resp. "synthesized").
In the examples, d-assignments will be defined by associating with each predicate symbol the list of the assignments to its arguments in the same order. The semantics of a RAG is not affected by introducing a splitting on the attributes, hence by Theorem 1 the semantics of DCP's is not affected by introducing a d-assignment.

Construction 2

Given a DCP $C = < \mathcal{N}, \mathcal{F}, \mathcal{P} >$ with a d-assignment d we construct an attribute dependency scheme $< N , P , Attr , D >$ defined as follows :

1. N, P and $Attr$ are defined as in construction 1.
2. $D = \{ D(p) \}_{p \in P}$, and
   $a\ D(p)\ b$ iff
   (i) $a \in Input(p)$ and $b \in Output(p)$
   (ii) the terms corresponding to these positions have a common variable.

End of Construction 2

Example 6

For the DCP of Example 3 and for the d-assignment *d* defined by *d(add)* = ↓↓↑ the relations D_p of the corresponding ADS are shown in Fig. 5.

End of Example 6

3.3 From DCP's to FAG's

Since FAG's have been extensively studied in the literature and have many applications it is interesting to know whether a given DCP can be transformed into an equivalent FAG. This will open for employing in compilation of DCP's some optimization methods developed for attribute evaluators.

To answer this question we assume that a DCP is given with a d-assignment. In this way the positions of each clause are split into input positions and output positions, as discussed above. To solve our problem we have to express the value of each output position of a clause in terms of the values of the input positions. For example, consider the clause

add(s(x),y,s(z)) <- add(x,y,z).

with the d-assignment *d*

d(add) = ↓↓↑.

The output position *add1(1)* depends on the input position *add1(0)* since they share the variable *x*. In every instance of the clause the values of the positions are terms. The value of *add1(1)* can be determined from the value of *add1(0)* by selecting the appropriate subterm of the latter. To be more precise we introduce for each *n*-ary functor *f* of the DCP *n* ***selector operators*** denoted *s1-f,...,sn-f*. These are partial operations on terms. For a given term *t* the value *si-f(t)* is defined to be t_i if *t* is of the form $f(t_1,...,t_n)$, and it is undefined otherwise. The identity function on terms is also considered to be a selector. Using this notation we can write a semantic definition for our example:

add1(1) = *s1-s(add1(0))*

where "=" denotes the equality on terms. To select a subterm of a given term it may be necessary to use a composition of selectors called a composed selector.

The value of an output position *a* of a clause is determined by the values of the input positions iff each variable of *a* occurs in some input positions. A d-assignment, which determines the splitting of the positions of the clauses will be called *safe* iff this condition is satisfied. A DCP for which there exists a safe d-assignment will be called a *simple* DCP.

Example 7

Consider the classical *append* program (written in the notation of DEC10 Prolog).

append([ ],L,L) <- .
append([E|L1],L2,[E|L3]) <- append(L1,L2,L3).

The d-assignment

$$d(append) = \downarrow \downarrow \uparrow$$

is safe. For the first clause it gives

$$append3(0) = append2(0).$$

However, this definition does not express the fact that in each instance of the clause the input position *append1(0)* is the empty list.

End of Example 7

Thus, in general case a condition concerning the form of the input positions of a clause may be necessary. To express such conditions we introduce the predicate *instance*, representing the following binary relation on terms:

$instance(t_1,t_2)$ holds iff t_1 is an image of t_2 under some substitution.

Now the semantic rule for the clause of Example 7 can be expressed as follows:
$instance(append1(0),[\ ]) \wedge instance(append2(0),L) \wedge append3(0) = append2(0).$
Clearly, the second component of this rule can be omitted since any term is an instance of a variable.

The examples discussed above can be generalized as follows.

Construction 3

Given a DCP $C < \mathcal{N}, \mathcal{F}, \mathcal{P} >$ with a safe d-assignment d we construct a FAG $G_C = < N, P, Attr, R, \mathfrak{I} >$ defined as follows:

1. N, P and $Attr$ are defined as in Construction 1.
2. For each clause c the semantic rule R_c is constructed as follows:

(i) for each output position a of c we construct the following semantic definition.

Let t_a be the term at the position a of c. For a variable x in t_a let b be an input position including x. Denote by S_{xb} the set of all composed selectors s such that $s(t_b)=x$. (Each of them corresponds to an occurrence of x in t_b). The semantic definition for a is of the form

$$a = \alpha(t_a)$$

where α is a substitution assigning to each variable x in t_a the term $s(b)$ for some s in S_{xb}.

(ii) For each pair of different occurrences of a variable x at (not necessarily different) input positions b_1 and b_2 of the clause c we construct the condition

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

where s_1 and s_2 are the selectors corresponding to the considered

occurrences of x in the terms at the positions b_1 and b_2. (Notice that the position names play the role of variables of the condition).

(iii) For each input position b if the term t_b is not a variable we construct the condition

$$instance(b,t_b).$$

R_c is the conjunction of all formulas constructed by (i), (ii) and (iii).

3. $\mathfrak{I}$ is defined as follows:

The semantic domain consists of all terms constructed with the functors of $\mathcal{F}$ and from variables (positions of the derivation trees).

The functors of $\mathcal{F}$ are associated with the term constructing operations, as in the basic term interpretation;

The selectors are associated with the selecting operations, as defined above;

The predicate letters = , and *instance*, represent respectively, the identity, and the relation defined above.

End of Construction 3

Notice that because of the free choice of the selector s in 2(i) the construction is deterministic only if in every clause each variable occurring at an output position occurs only once at one input position.

Example 8

Construction 3 applied to the DCP and to the d-assignment of Example 7 results in the FAG with the production rules:

(p_0) *append* $\rightarrow \epsilon$

(p_1) *append* $\rightarrow$ *append*

and with the corresponding semantic rules

(r_1) *instance(append1(0),[ ])* $\wedge$ *append3(0) = append2(0).*

(r_2) *instance(append1(0),[E | L1])* $\wedge$
*append3(0))=[***head***(append1(0) | append3(1)]* $\wedge$
*append1(1)=***tail***(append1(0))* $\wedge$ *append2(1)=append2(0)*

where **head** and **tail** are the selectors of the components of a list.

End of Example 8

The main result of this section follows directly from Construction 3.

Proposition 1

For every simple DCP there exists a FAG whose set of decorated trees is isomorphic with the set of proof trees of the DCP.

To obtain such a FAG for a given DCP one has to find a safe d-assignment and to apply Construction 3. As noticed in [9] the class of simple DCP's is rather large. Furthermore, Construction 3 can be extended for a generalization of this class obtained by considering "multiple d-assignments" [9]. These describe the situation when different calls of the same procedure in one logic program have to be characterized by different d-assignments.

It is worth noticing that in some cases the FAG obtained by Construction 3 is pure. This happens for example if in each clause all input positions are different variables.

3.4. Transforming pure FAG's into DCP's

The question arises whether it is possible to transform a RAG into a semantically equivalent DCP. For this the semantic values of the RAG should be represented by the elements of the semantic domain of the DCP to be constructed. Generally it is not possible since the semantic domain of any DCP is countable and the definition of RAG puts no restriction on the nature of the semantic domains.

From practical point of view, this question may be more interesting if restricted to a subclass of RAG's used in computational applications, e.g. to pure FAG's. In the latter case the attribute values can be represented by terms, as discussed in Section 2, and the problem can be solved by the following construction.

Construction 4

Given a pure FAG $G = < N, P, Attr, R , \mathfrak{I} >$ where $\mathfrak{I}$ is the basic term interpretation we construct a DCP $C_G = < \mathcal{N}, \mathcal{F}, \mathcal{P} >$ defined as follows:

1. $\mathcal{F}$ is the set of functors occurring in the semantic definitions of G.
2. $\mathcal{N}$ is the set of nonterminals of G: each nonterminal X is considered to be a n_X-ary predicate letter.
3. To define $\mathcal{P}$ we construct for each production rule p of G a corresponding clause c_p :
   Let p be of the form
   $$X_0 \rightarrow X_1 .. X_m$$
   for some m≥0 and let
   $$Pos(p)=\{a_1,...,a_k\}$$
   Clearly, $k=n_{X_0}+...+n_{X_m}$.

   Assume $a_i \preceq a_j$ for $i \leq j$. For $i=1,...,k$ denote by d_i the variable a_i if a_i is an input position, and the term on the right-hand side of the corresponding semantic definition if a_i is an output position.

Now c_p is defined to be the clause

$$X_0(t_{01},...,t_{0n_0}) <- X_1(t_{11},...,t_{1n_1})...X_m(t_{m1},...,t_{mn_m})$$

where

$t_{0j} = d_j$ for $j = 1,...,n_0$, and
$t_{ij} = d_{(n_0+...+n_{i-1}+j)}$ for $i=1,...,m$ and $j=1,...,n_i$.

Finally $\mathcal{P}$ is defined to be the set of all clauses c_p for $p \in P$.

End of Construction 4

Example 9

Construction 4 applied to the pure FAG of Example 2 gives the following DCP:

$X(initial(0),initial(0)) <-.$
$X(initial(0),final(2)) <- I(), X(f_i(initial(0)),final(2))$
$X(initial(0),final(2)) <- D(), X(f_d(initial(0)),final(2))$

End of Example 9

The main result of this section follows directly from Construction 4.

Proposition 2

For any pure FAG with the basic term interpretation there exists a DCP whose set of proof trees is isomorphic with the set of decorated trees of the FAG.

To find such a DCP it suffices to apply Construction 4 to a given pure FAG. Notice that the DCP's obtained by Construction 4 are simple and that Construction 4 is reversible under Construction 3.

4. Examples of applications of the dependency relation

Our attempt to relate DCP's and attribute grammars resulted in the notion of DCP with a d-assignment, called in the sequel *annotated* DCP. The directions assigned to the predicates of a DCP make it possible to apply the proof techniques developed for attribute grammars [5] to proving properties of logic programs. This section illustrates that statement by three examples. First we deal with the occur check problem, i.e. with the problem whether an infinite term can be produced during a computation of an augmented DCP. We show that this problem is in general undecidable, and we give a sufficient condition under which this cannot happen. Then we introduce a notion of data-driven DCP and we prove a sufficient condition for a DCP to be data-driven. Finally we consider the problem whether a DCP can be run without employing unification in its general form. Though the results of this section may be of some practical importance our primary objective is to illustrate the methodological usefulness of the dependency relation rather than its practical applications.

4.1. Sufficient conditions to make occur check unnecessary

Existing interpreters of logic programs are based on the resolution principle and employ unification. The unification process should construct a most general unifier of given arguments, or it should show that no unifier exists. In the sequel we will assume without loss of generality that the unifier is constructed as described in [22]. Unification is eventually reduced to elementary steps, where an attempt is made to unify a term t which is not a variable and a variable X. A unifier of X and t exists iff X does not occur in t. (Alternatively it may be considered that if X occurs in t then the unifier is an infinite term). Thus the unification algorithm should in principle check at every step of unification, whether a variable occurs in a term, or not. This is called ***occur check*** . Many existing interpreters do not perform occur check since it is rather time expensive, and for most programs it never happens that the term argument of a unification step includes its variable argument. However, unification without occur check destroys completeness of the SLD resolution, since in this case the results of a computation may be incompatible with the declarative semantics of a logic program (see e.g. [21]). In this section we will first show the general occur check problem to be undecidable and then use concepts related to attribute grammars to develop a sufficient condition for a DCP under which the occur check can be safely omitted.

To be more precise we give a formal definition of the occur check problem.

A pair (t_1, t_2) of terms, or atomic formulas, is said to be ***subject to occur check*** iff occur check is the only reason of non-existence of a most general unifier of t_1 and t_2. For example, the terms $f(x,g(x))$ and $f(y,y)$ are subject to occur check, but the terms $h(x,g(x),g(x))$ and $h(y,y,f(y))$ are not since the subterms $g(x)$ and $f(y)$ are not unifiable.

Let T be a resolution tree of an augmented DCP $<C,g>$. We assume without loss of generality that the variables in the labels of T are different from those in the clauses of C. T is said to be ***subject to occur check*** iff there exists a pair (r_1,r_2) of atomic formulas such that:

- r_1 is the label of a leaf of T;
- r_2 is the head of a clause in C;
- the pair (r_1,r_2) is subject to occur check.

Definition 8 : The occur check problem

Given an augmented DCP $< C, g >$, decide whether there exists a resolution tree which is subject to occur check.

Theorem 2

The occur check problem is undecidable.

Proof

We describe a construction which for an arbitrary Turing machine M gives a DCP C that "simulates" the computations of M by resolution trees of C. In this way we relate the halting problem for Turing machines with the occur check problem for a class of DCP's.

The construction goes as follows.

Any instantaneous description of M is modelled by a ground term of the form $id(l,r,q)$, where l is a list describing the tape of M to the left of the head in reverse order, r is a list describing the tape of M to the right of the head, including the scanned symbol, and q is the actual state of M. Thus, the instantaneous description $abqcd$, where a,b,c,d are tape symbols and q is a state will be represented by the term $id([b|a],[c|d],q)$. (We use here the list notation of the DEC10 Prolog). A move of M in the state q for a scanned symbol x is determined by the transition function; q is replaced by some q', x by some symbol x', and the head may be moved, for example to the left. Thus, each move can be represented by a pair of terms. For example, the move described above can be represented by the pair:

$$(id([Z|Y],[x|X],q),id(Y,[Z,x'|X],q')).$$

Clearly, all moves of M can be represented by a finite set of pairs of terms.

The machine M when started in some initial instantaneous description i continues until it reaches a final state q_f (it may also interrupt the computation if the next move is undefined). The result of a successful computation is the final instantaneous description. To simulate operation of M we introduce a ternary predicate ***machine***. Its intended interpretation is the relation on instantaneous descriptions defined as follows: $machine(i_1,i_2,i_3)$ holds iff the machine M when started in i_1 reaches i_2 in one move and the computation terminates in i_3. The computations are to be simulated by resolution trees: if the final state q_f is reached by M the resolution tree corresponding to the computation should be completed. Therefore we introduce the clause

(i) $\quad machine(id(Y,Z,q_f),id(Y,Z,q_f),id(Y,Z,q_f))$ <- .

where Y and Z are variables. For each move of M characterized by a pair (a,b) of terms, as described above we introduce the clause

(ii) $\quad machine(a,b,F)$ <- $machine(b,N,F)$.

where N and F are variables not occurring in the terms a and b.
The DCP C consists only of the clauses of type (i) and (ii).

To simulate M we call C with the goal:

$goal$ <- $machine(i,N,F)$

where i is the ground term representing the initial instantaneous description of the computation while N and F are variables.

It is easy to see that any resolution tree of such a goal is not subject to occur check and has the following properties:

- each of its nodes has at most one son;
- the first arguments of the labels of the consecutive nodes are ground terms representing consecutive instantaneous descriptions of a computation of *M*.

Observe also that every terminating computation of *M* is represented by a unique complete resolution tree of *C*.

Consider now the DCP *C'* obtained from *C* by replacing the clause (i) by the following clauses:

(iii) $\quad machine(id(Y,Z,q_f),id(Y,Z,q_f),id(Y,Z,q_f)) <- equal(Y,g(Y)).$

(iv) $\quad equal(Y,Y) <-.$

Let *C'* be called with the same goal clause

$$goal <- machine(i,N,F).$$

Let *T* be an incomplete resolution tree of this goal. Two cases are possible:

Case 1.

The only leaf of *T* is of the form *machine(r,s,t)*. In this case *T* represents a sequence of consecutive moves of *M*, as discussed above, and *T* is not subject to occur check.

Case 2.

The only leaf of *T* is of the form *equal(t,g(t)) and T* is subject to occur check. In this case the consecutive labels of *T* with except of the label of the leaf represent consecutive moves of a complete terminating computation of *M*. Observe that any terminating computation of *M* gives rise to such a tree.

Thus, a resolution tree which is subject to occur check exists iff the machine *M* halts on the input determined by the goal clause. If the occur check problem were decidable we could decide whether a given Turing machine halts on a given input what is known to be undecidable. Hence the occur check problem is undecidable.

End of proof

Looking for sufficient conditions

It is well known that if in a DCP all heads of the clauses are *linear* (i.e. in each of them every variable occurs at most once) the occur check is not necessary. We present here a more general condition, whose principle is to relate the occur check problem with well-formedness of the attribute dependency scheme associated with a given DCP.

When running a DCP the unification procedure is invoked to unify a distinguished leaf *l* of a partial proof tree *T* and the left-hand side of a clause *c*

of the program. If the arguments are unifiable a most general unifier θ is constructed, and a new partial proof tree is created: in the tree $\theta(T)$ the leaf $\theta(l)$ is replaced by the subtree $\theta(c)$. It is assumed that T and c have no common variable, otherwise the variables of c are renamed before the unification process begins.

The proof trees which can be constructed in that way are the resolution trees (cf. Section 2.2).

The leaf l and the left-hand side of c are atomic formulas of the form $p(t_1,\ldots,t_n)$ and $p(t'_1,\ldots,t'_n)$, where p is an n-ary predicate and $t_1,\ldots,t_n$ $t'_1,\ldots,t'_n$ are terms. A most general unifier of the lists of terms $(t_1,\ldots,t_n)$ and $(t'_1,\ldots,t'_n)$ can be constructed by the composition of a most general unifier η_1 of t_1 and t'_1, and of a most general unifier of the lists of terms $(\eta_1(t_2),\ldots,\eta_1(t_n))$ and $(\eta_1(t'_2),\ldots,\eta_1(t'_n))$. In that way the unification of a pair of atoms can be reduced to n unification steps for the consecutive positions of the atoms. These steps can be described as follows: denote by η_i a most general unifier of the terms $\theta_{i-1}(t_i)$ and $\theta_{i-1}(t'_i)$, where $i=1,\ldots,n$, θ_0 is the identity substitution, and θ_i is the composition of θ_{i-1} and η_i. Then a most general unifier of l and of the left-hand side of c is θ_n.

To deal with the occur-check problem we introduce a notion of ***partially unified*** resolution tree. It is a graph obtained from a resolution tree T and from a clause c (with renamed variables) by unifying a number $k \geq 0$ of positions of the distinguished leaf of T with the corresponding positions of the left-hand side of c (Fig. 6). This graph is an ordered labeled tree T' with a distinguished node, where unification takes place, called the active node of T'. The subtree of T' whose root is the active node can be considered an instance of c, while the remaining part of T', including the active node can be considered an instance of T. The label of the active node has to be described more precisely. It has, as usual n positions, where n is the arity of its predicate. However, only first k of them are terms, while the others are pairs of terms. The active node of T' is created by pasting together a distinguished leaf l of T and the root of c. Let the label of l be $p(t_1,\ldots,t_n)$ and let the label of the root of c be $p(t'_1,\ldots,t'_n)$. Then for $i = 1,\ldots,k$ the i-th position of the label of the active node of T' is $\theta_k(t_i) = \theta_k(t'_i)$, while for $j=k+1,\ldots,n$ *the* j-th position of the label is the pair of terms $\theta_k(t_j)$ and $\theta_k(t'_j)$. The other labels of T' are the instances of the corresponding labels of T, or c, under the substitution θ_k. Observe, that the active node has as a matter of fact $k + 2(n-k)$ positions; for each $k<j\leq n$ there are two positions of the active node: one originating from the tree T, and the other originating from c.

We assume now that a d-assignment d is defined for a given DCP. Thus we

consider the attribute dependency scheme obtained in the construction 2 (section 3.2). For any resolution tree T of the program as well as for any clause c the ADS determines a dependency relation on the positions of its labels as described in section 2.1 . The dependency relation is also defined for a partially unified resolution tree T' obtained by unification of some positions of the root of c with the corresponding positions of a leaf of T. The dependency relation on the positions of T' is determined by the dependency relations of T and c : its graph is constructed from the graphs of both relations by identification of the unified positions, belonging originally to disjoint domains.

To formulate our sufficient condition we extend the notion of input position and the notion of output position for the case of a partially unified resolution tree T' constructed from a resolution tree T and a clause c.

A position of a partially unified resolution tree is said to be an *input position* with respect to a given d-assignment iff it is
(1) a synthesized position of a leaf, or
(2) an inherited non-unified position of the active node originating from c, or
(3) a synthesized non-unified position of the active node originating from T.

A position of a partially unified resolution tree is said to be an *output position* with respect to a given d-assignment iff it is
(1) an inherited position of a leaf, or
(2) an inherited non-unified position of the active node originating from T, or
(3) a synthesized non-unified position of the active node, originating from c.

We will also need the following property of unification which is left without proof.

Proposition 9

Let t' be a term, let t'' be a linear term with no variable in common with t', then

(i) the pair (t',t'') is not subject to occur check, and

(ii) if there exists a most general unifier θ of t' and t'' then

(1) If X is a variable of the domain of θ occurring in t' then $\theta(X)$ is a linear term and all its variables occur in t''.

(2) If X, Y are different variables of the domain of θ occurring in t' then $\theta(X)$ and $\theta(Y)$ have no common variable.

Definition 9 : proper d-assignment

Let C be a DCP and let g be a goal clause. An d-assignment d is said to be a *proper* d-assignment for the augmented DCP consisting of C and g iff
(1) the associated ADS of the augmented DCP is well formed, and
(2) in each clause of the augmented DCP all input positions are linear and

have no common variables.

Lemma 1

Let $<C,g>$ be an augmented DCP where C is a DCP and g is a goal clause. If d is a proper d-assignment then in any partially unified resolution tree T of $<C,g>$

(i) all input positions are linear and have no common variables, and

(ii) if an input position p has a common variable with an output position q then $p \rightarrow^{+}_{T} q$.

Proof

We prove the lemma by induction on the size of the partially unified resolution tree.

For the tree consisting of the goal g the lemma holds by the assumption concerning g and the definition of $\rightarrow^{+}_{T}$.

Assume now that it holds for any completely unified resolution tree T, constructed from at most n instances of clauses. We will show that it holds also for any partially unified resolution tree that can be constructed from T and an arbitrary clause c. This will be proved by induction on the number of already unified positions of the active node of the tree.

If the number of the unified positions of the active node is zero, then (i) holds by the induction hypothesis concerning T, by the hypothesis of the lemma, and by the assumption that T and c have no common variables (renaming). The condition (ii) holds trivially since the positions of T and of the instance of c are disjoint, and the dependency relation of the partially unified resolution tree is the union of the dependency relations of T and c.

Assume now that the lemma holds for the partially unified resolution tree T' constructed by unifying $k \geq 0$ positions of a leaf l of T and of the left-hand side of c. We prove that it holds also for the tree T'' obtained from T' by unification of the terms at the $k+1$ position of the active node.

For the tree T'' we prove separately (i) and (ii).

By the induction hypothesis (i) holds for T'. The number of the input positions of T'' is that of T', decreased by 1 since one of the input positions of T' is being unified and disappears in T''. Any input position of T'' corresponds to an input position of T'. We show first that every input position of T'' is linear. Assume that an input position p'' of T'' differs from the corresponding input position p' of T'. This means that a variable X occurring at p' occurs also at one of the unified positions. By the induction hypothesis (i) X may occur only at the output unified position. Since the input unified position is linear, by Proposition 3, the term replacing X after unification must be also linear. Moreover, by the induction hypothesis, the variables occurring in this term do

not occur at any input position of T' with except of the input unified position. We show now that different input positions of T'' have no common variables. Notice that any variable occurring at an input position of T'' occurs also at an input position of T'. If a variable X appears at an input position p'' of T'' and does not appear at the corresponding input position p' of T' so it appears at the input unified position in T'. Hence by the induction hypothesis and by Proposition 3 this variable cannot appear at any input position of T'' different from p''. Note that Proposition 3 can be used since the ADS is well-formed and thus, by the induction hypothesis, both unified positions have no common variable.

We prove now (ii). Let p'' be an input position of T'', let q'' be an output position of T'' and let X be a variable occurring both at p'' and at q''. Denote by p' and respectively by q' the positions of T' corresponding to p'' and q''. We show (ii) by considering separately the following cases:

Case 1.
X occurs at p' and at q'. Thus $p' \rightarrow^{+}_{T'} q'$, hence $p'' \rightarrow^{+}_{T''} q''$.

Case 2.
X occurs at p' but not at q'. Since X is passed to q'' during unification, it must occur also at a unified position of T'. By (i) it must be the output unified position u. Moreover, there must be a variable Y occurring both at q' and at a unified position of T'. By Proposition 3 Y cannot occur at the input unified position u, so it must occur at the output unified position w. Hence, by the induction hypothesis concerning (ii): $p' \rightarrow^{+}_{T'} u$ and $w \rightarrow^{+}_{T''} q''$. Thus, after unification, $p'' \rightarrow^{+}_{T''} q''$.

Case 3.
X occurs at q' but not at p'. By the reasoning similar to that of Case 2 one can prove that there is a variable Y occurring at p' and at u, hence X occurs at w. Thus $p' \rightarrow^{+}_{T'} u$, $w \rightarrow^{+}_{T'} q'$, and consequently, $p'' \rightarrow^{+}_{T''} q''$.

Case 4.
X occurs neither at p' nor at q'. Since X is passed to p'' during unification there is a variable Y occurring both at p' and at u, which is substituted by a term with X. On the other hand, there is a variable Z different from Y occurring at q', which is also substituted by a term with X. Thus Z must appear at one of the unified positions. Since w is linear, by the assumption that Y unifies with a subterm of w including X we get from Proposition 3 that Z cannot occur at u. Thus Z appears at w, hence (ii) holds also in this case.

End of proof

Now we formulate the main result of this section.

Theorem 3

Given an augmented DCP $<C,g>$ if there exists a proper d-assignment for C and g then none of the resolution trees of the DCP is subject to occur check.

Proof

It follows by Lemma 1 that the unified positions of the active node of a partially unified resolution tree have no common variables. (Otherwise the ADS associated with the DCP is not well-formed what contradicts the assumption that the d-assignment is proper). Thus, by Lemma 1 Proposition 3 applies to any pair of terms to be unified in any partially unified resolution tree. Hence none of the resolution trees of the DCP is subject to occur check.

End of Proof

The theorem gives a sufficient condition under which no infinite term can be created during resolution of the goal g with the program C. It requires existence of a proper d-assignment for C and g. The latter problem is generally intractable, since it is known that the complexity of the well-formedness of an attribute grammar is intrinsically exponential [13]. However, some sufficient conditions of well-formedness known from the literature, which can be checked in polynomial time could be used to implement tractable tests for the occur check problem.

Example 10

Consider the augmented DCP $<C,g>$, where C is the DCP of Example 4 and g is a goal clause of the form

$$goal <- add(t_1,t_2,x)$$

where x is a variable, t_1 is a term, and t_2 a term without x.

The d-assignment of Example 6 :

$$d(add) = \downarrow \ \downarrow \uparrow$$

is proper for this DCP. Every term occurring in its clauses is linear and well-formedness of the attribute scheme can be proved by some known criteria. Thus no resolution tree is subject to occur check, which may be safely omitted.

End of Example 10

Though the syntactic conditions of Theorem 3 are conceptually simple they apply to a non-trivial class of DCP's. However, Lemma 1 and Theorem 3 should be seen rather as examples of application of the dependency relation than as a practical solution of the occur check problem. A more complete treatment of this problem is given in [21]. In contrast to our approach it is based on "simulation" of computations of a DCP. The main problem in this case is how to approximate in a finite way the potentially infinite set of labels

of resolution trees. The solution presented in [21] employs the notion of binary term schema and leads to rather complicated algorithms.

The proofs of Lemma 1 and Theorem 3 show that the notion of dependency relation makes it possible to study run-time properties of logic programs such like occur-check.

Our approach to the occur check problem makes it possible to relate formally the well-formedness problem for FAG's with the occur-check problem for a class of DCP's. We introduce first some auxiliary notions.

By a resolution tree of a non-augmented DCP C we mean any resolution tree of a clause in C.

By an *extending* FAG we mean a pure FAG such that the right-hand side of each semantic definition is a term different from a variable. Clearly, each pure FAG can be transformed into a semantically equivalent extending FAG. For this it suffices to introduce a new functor I interpreted as identity, and to replace each variable x which is the right-hand side of a semantic definition by the term $I(x)$.

Theorem 4

An extending FAG G is well-formed iff none of the resolution trees of the DCP C_G obtained from G by Construction 4 is subject to occur check.

Proof

Assume that G is well-formed. Observe that under the d-assignment of C_G corresponding to the attribute splitting of G, all input positions of each clause of C_G are different variables. Thus, the d-assignment is proper for C_G and by Theorem 2 none of the resolution trees is subject to occur-check.

Assume now that none of the resolution trees of C_G is subject to occur check. We prove that in this case G must be well-formed.

By Construction 4 there is a one-one correspondence between the production rules of G and the clauses of C_G. To show that G is well-formed it suffices to prove that for each resolution tree of C_G the graph of the dependency relation induced by the attribute splitting of G is acyclic.

Let T be a resolution tree of C_G. Observe first that each input position of T is a variable and each output position of T is a term. Moreover, an output position depends on an input one iff the variable of the latter occurs in the term at the output position. This can be shown by induction on the size of the resolution tree. This property holds also for partially unified resolution trees.

The local dependency relations in clauses have no cycles. Thus a cycle may be introduced only by unification of an output position of a partially unified

resolution tree with an input position on which the output position depends. But in this case the input position is a variable and this variable occurs in the term at the output position. This means that the terms are subject to occur check what contradicts the assumption about C_G. Hence G is well-formed.

End of Proof

The proof of Theorem 4 shows that the notion of occur check is closely related to the notion of well-formedness of an attribute grammar. This was informally pointed out also in [1], where the formalism of Definite Clause Translation Grammars was introduced, which may be thought of as a logical implementation of functional attribute grammars. It was claimed that using DCTG's one can avoid well-formedness tests since the occur check will detect possible circularities. However, unification with occur check is rather expensive, and if occur check is not performed during the unification a safe method of dealing with infinite terms may be necessary, or otherwise some static tests for checking that infinite terms cannot be created. We have shown that such tests are generally at least as expensive as well-formedness tests.

4.2 Data Driven Programs

The depth-first left-to-right strategy used commonly by Prolog interpreters, called in the sequel the ***standard*** strategy, may sometimes have some disadvantages. For example, consider the following simple program

grandfather(X,Y) <- *father(X,Z), father(Z,Y).*
father(Mary,George) <-.
father(Paul,George) <-.
father(Peter,Paul) <-.

...

If we consider the goal *grandfather(Peter,Y)* the best way to solve it is to use the standard strategy. On the other hand, if we consider the goal *grandfather(X,George)* it would be better to deal first with the second atom of the right-hand side of the corresponding clause, since both variables of the first atom are uninstantiated and that will cause backtracking. Alternatively, to avoid backtracking, one can rearrange the atoms in the right-hand side of the first clause.

In this example, to choose a good strategy (or to perform a proper rearrangement) one needs additional information, namely which positions of the goal are expected to be ground terms. Similar information may be optionally given to a Prolog compiler as mode declarations [16]. In [3] an

obligatory use of this type of control information is suggested for all predicates of the program. A more complicated control information of similar type is discussed in [24]. A method for generating control information for a given program is described in [18].

In this section we study the propagation of ground terms in resolution trees during computations of augmented DCP's. Thus, our objective is different from those of the papers mentioned above. We suggest to provide control information by a d-assignment and we define a class of augmented DCP's for which data flow during the execution can be properly modelled by the dependency relation induced by a given d-assignment. To be more precise we introduce the following definition.

Definition 10

A DCP C, is said to be ***data-driven*** under a direction assignment d iff at any step of a computation using the standard strategy the actual subgoal has all its inherited positions instantiated to ground terms, provided that the initial goal has this property.

Clearly, the example program is data-driven under the d-assignment d such that:

$d(grandfather) = \downarrow \uparrow$, $d(father) = \downarrow \uparrow$

but it is not data-driven under the d-assignment d':

$d'(grandfather) = \uparrow \downarrow$, $d'(father) = \uparrow \downarrow$.

A DCP C' is said to be a ***version*** of C if it is obtained from C by rearrangement of atoms in the right-hand sides of some clauses of C.

A version of the example program can be obtained by replacing its first clause by the clause

grandfather(X,Y) <- father(Z,Y), father(X,Z).

This version is data-driven under the direction assignment d'.

In this section we use a proof technique introduced for RAG's [5,7] to give a sufficient condition for an annotated DCP to be data-driven, or to be rearranged so as to become data-driven under the same d-assignment. We show that the restricted class of data-driven DCP's satisfying this condition is powerful enough to model any Turing machine.

Generally speaking, as noticed in [24], problems like whether a given DCP is data-driven under a given d-assignment are undecidable. The proof technique is similar to that used in the proof of Theorem 2. This section gives a conceptually simple and algorithmically tractable sufficient condition for a DCP to be data-driven under a given d-assignment. The class of DCP's for which there exist d-assignments satisfying the condition is restricted but non-trivial.

We formulate first a sufficient condition for an annotated DCP which makes it sure that in any proof tree whose inherited positions of the root are ground also the synthesized positions of the root are ground. (A version of the theorem which follows can be also found in [9]).

Theorem 5

Let C be an annotated DCP with a safe direction assignment d and such that the ADS obtained from C by Construction 2 is well formed. For every proof tree of C if all inherited positions of its root are ground then also all synthesized positions of its root are ground.

Proof

The theorem is proved by induction following the method introduced in [7] for Logical Attribute Grammars (here we deal with a particular case of the method).

We show that in any proof tree whose inherited positions of the root are ground all positions must be ground. Since the attribute scheme is well formed the dependency relation on the positions of any proof tree determines a partial order on the positions. The induction will follow the directed acyclic graph spanned on the positions of the tree and corresponding to this partial order.

Since d is safe then the only minimal elements in the ordering relation are the inherited positions of the root. These are assumed to be ground.

Assume now that for some non-minimal position x of a given proof tree all positions less than x in the partial order are ground. Since x is not minimal it is an output position of an instance of a clause c. Since d is safe all variables occurring in c on the output position corresponding to x occur also in some input positions on which the output position depends. By the induction hypothesis in the considered instance of c the corresponding input positions must be ground, since they precede x in the partial ordering. Therefore x is also ground. Hence all positions of the tree are ground.

End of Proof

Following [12] we now introduce a notion of one-sweep attribute dependency scheme, which will be used in the sequel to formulate the main result of this section.
For any nonterminal X of an attribute scheme denote by $\dashv_X$ the relation on the positions of X defined as follows:

$$p \dashv_X q \text{ iff } p \text{ is inherited and } q \text{ is synthesized}$$

With each production rule

$$r : X_0 \rightarrow X_1 ... X_n$$

of an attribute scheme we associate a relation $=>_r$ on its positions defined as follows:

$$=>_r = \dashv_{X_1} \cup ... \dashv_{X_n} \cup \rightarrow_r$$

where $\rightarrow_r$ denotes the local dependency relation as defined in Section 2.

Intuitively, the relation $=>_r$ gives a rough approximation of the dependencies between the positions of instances of r in any decorated tree of the attribute scheme. More precisely: If r' is an instance of r in a decorated tree T, and $p' \rightarrow^+_T q'$ then the corresponding positions p and q are in the relation $=>_r$ (but not vice versa). Fig. 7 shows the diagram of the relation $=>_r$ for the first rule of the attribute scheme associated with the example DCP of this section under the d-assignment d'.

Definition 11

An attributed scheme is called *one-sweep* iff for each of its productions r the graph of the relation $=>_r$ is acyclic.

(This definition is equivalent to that given in [12], as shown in [5, p.18]). It is known [12] that any one-sweep attribute scheme is well formed.

We use the relation $=>^+_r$ to introduce an ordering on the nonterminals of the production rule r. Let

$$r : X_0 \rightarrow X_1 ... X_n$$

be a production rule of a one-sweep attribute scheme. Denote by $<_r$ the relation on
$\{X_1 , ... ,X_n \}$ defined as follows: $X_i <_r X_j$ iff there exists a position p of X_i and a position q of X_j such that $p =>_r q$. For the example rule r of Fig. 7, $n = 2$, $X_0 = grandfather$, $X_1 = X_2 = father$, and $X_2 <_r X_1$. It follows from the definition that

Proposition 4

$<_r$ is a partial ordering.

As a consequence we get

Theorem 6

Given an annotated DCP C with a safe direction assignment d, if the associated ADS is one-sweep then there exists a version of C which is data driven under the direction assignment d.

Proof

Consider the DCP C' obtained from C by rearranging the right-hand side of each clause c according to the relation $<_{r_c}$, where r_c is the production rule of the associated ADS. Consider now the resolution process using the standard strategy, and starting with a goal whose inherited positions are ground. Using Theorem 5 one can show by induction on the number of resolution steps that at each step of the process the inherited positions of the actual subgoal are ground.

End of Proof

The class of logic programs which have a corresponding one-sweep scheme seems to be rather restricted. Nevertheless, many of the examples published in the literature fall in this class. Moreover, one can model an arbitrary Turing machine by an annotated logic program, whose associated attribute scheme is one-sweep. The DCP obtained for a given Turing machine by the construction described in the proof of Theorem 4 has this property for the d-assignment

$$d(machine) = \downarrow \uparrow \uparrow .$$

4.3. Running clause programs without unification.

In this section we give a sufficient condition which makes it possible to run a DCP with a very restricted form of unification. For the case of a term t_1 being unified with its instance t_2 the resulting unifier assigns to each variable occurring in t_1 a subterm of t_2 , while the variables occurring in t_2 remain unbound. An occurrence of a variable in t_1 can be localized by a number of selection operations. For instance, for $t_1 = f(g(X,Z),W)$, W is the second subterm of t_1 , while X is the first subterm of its first subterm. Thus, whichever instance of t_1 is the term t_2 the unifier assigns to X the first subterm of its first subterm. Similar properties of logic programs are used in some compilers to compile out unification (see e.g. [19]).

We now define a sufficient condition under which any unification during a computation of an augmented DCP reduces to a finite number of unification steps of the type described above. We introduce first some auxiliary notions.

An augmented DCP $<C,g>$ is said to be *d-ordered* for a given direction assignment d iff the associated ADS is one-sweep and for each clause c of the

program (including the goal clause) the partial ordering $<_{r_c}$ of the nonterminals of the production rule r_c of the ADS is consistent with the textual ordering of the corresponding atoms in c. Clearly, if $<C,g>$ is d-ordered DCP then C is data-driven under d (Theorem 6).

Definition 12

Let $x = p(t_1 ,...,t_n)$ and $x' = p(t'_1 ,...,t'_n)$ be atoms and let d be a direction assignment on a set of predicates including p. We say that x ***d-subsumes*** x' iff the following conditions are satisfied:

1. x and x' have no common variables;
2. the terms at the inherited positions of x' are linear and have no common variables;
3. the terms at the synthesized positions of x are linear and have no common variables;
4. for every inherited position i of the predicate p, t_i is an instance of t'_i ;
5. for every synthesized position i of the predicate p, t'_i is an instance of t_i .

Clearly, a most general unifier of such atoms always exists and it can be constructed as discussed above, without employing a general unification algorithm. For example, the atom

$$add(s(s(X)), s(X), s(Z))$$

d-subsumes the atom

$$add(s(W), s(V), s(s(W)))$$

under the d-assignment $d(add) = \downarrow\downarrow\uparrow$. A most general unifier θ of the atoms can be constructed by unifying separately their corresponding components, as described above. This gives:

$$\theta(W) = s(X); \quad \theta(V) = X; \quad \theta(Z) = s(W).$$

We give now a sufficient condition concerning a DCP and its goal, under which any unification during the run of the program can be performed as described above. Moreover, for any possible unification step one of the term arguments is known before the computation starts, while the other one is created in run time.

Theorem 7

Let $<C,g>$ be a d-ordered augmented DCP such that:

(1) each variable occurring in a clause (including the goal clause) occurs on exactly one input position of this clause.

(2) if x is an atom occurring in a right-hand side of a clause, and y is a left-hand side of a clause such that x and y unify, then every synthesized position of y is an instance of the corresponding position of x,

then during the computation of C with g , using the standard rule, if the actual subgoal unifies with the left-hand side of a clause, it also *d*-subsumes it.

Proof

Since the goal clause satisfies (1) and the DCP is *d*-ordered the inherited positions of the first atom of the right-hand side of the goal clause must be ground. Observe also that Lemma 1 (Section 4.1) applies to the DCP $<C,g>$. (Any one-sweep ADS is also well formed).

For an arbitrary step of the resolution process we show that if the actual subgoal x unifies with the left-hand side x' of some clause it also *d*-subsumes it.

We check the conditions of Definition 12:

1. Because of renaming x and x' have no common variables.
2. Since x' is the left-hand side of a clause, by (1) all terms at the inherited positions of x' are linear and have no common variables.
3. By the proof of lemma 1 all terms at the synthesized positions of x are linear and have no common variables.
4. Since $<C,g>$ is *d*-ordered, all terms at the inherited positions of x are ground. Thus, they are instances of the terms at the corresponding positions of x'.
5. To verify this last condition we have to show that by using the standard computation rule, no input position of the partial proof tree is changed before its unification. To be changed, we know by lemma 1 that it is necessary to have a path from that input position to some previously unified output position. It is easy to show, using the d-ordered condition and the one-sweep property that this is not possible. Thus all output positions of the partial proof tree are copies (with possibly renamed variables only) of terms of the corresponding synthesized positions associated to predicates of the body of a clause. Hence by (2) all terms at the synthesized positions of x' are instances of the terms at the corresponding synthesized positions of x.

Thus by 1-5 x *d*-subsumes x'.

End of Proof

For the examples published in the literature it is often possible to find a d-assignment satisfying the conditions of Theorem 7. For the append program

append([],L,L)<-.
append([E|L1],L2,[E|L3]) <- append(L1,L2,L3).

assigning the directions $\downarrow \downarrow \uparrow$ or $\uparrow \uparrow \downarrow$ to the predicate ***append*** we obtain by Construction 2 a one-sweep ADS. Also the conditions (1) and (2) of Theorem 7 are satisfied under this direction assignment. Provided, that the goal is of the form ***append(l1,l2,R),*** where *l1* and *l2* are ground terms and R is a variable

the program can be run without unification in its general form. This information could be used to compile it into an efficient code.

Also the DCP obtained for a given Turing machine by the construction described in the proof of Theorem 4 fulfills the conditions of Theorem 7 under the direction assignment *d(machine)* = $\downarrow \uparrow \uparrow$.

5. Conclusions

The paper explains formally the nature of the relationship between logic programs and attribute grammars. Both formalisms specify relations, which may be defined by referring to the similar notions of decorated tree and proof tree.

The similarity of the formalisms makes possible transfer of the expertise. For example, the results of Section 4 concerning logic programs were obtained by using a proof technique introduced for attribute grammars. Section 4 shows also that the notion of dependency relation originating from attribute grammars may be used as a formal framework for studying run-time properties of logic programs. This framework results in clear and simple concepts which may be used in logic programming.

Most of the literature on attribute grammars is devoted to FAG's. This means that to transfer the expertise it may be desirable to restrict the use of DCP's to the simple DCP's, which by Construction 3 can be transformed into FAG's. We have shown that simple DCP's are a non-trivial class of programs. Nevertheless the question arises whether this restriction kills the spirit of logic programming. We leave it open for statistical investigation of published logic programs. It is easy to see that "reversible" use of a predicate in a DCP violates the restriction. For example, the *append* procedure may be called within one program to concatenate a pair of lists and to also to split a given list into sublists. If such different calls appear in a DCP, formally it is not a simple DCP. However, quite often it can be transformed into an equivalent simple DCP by creating different copies of the subprogram corresponding to the different types of calls. If such a transformation is done at compile time it does not influence the external form of the program, and in the same time it may result in producing different versions of the compiled code for different uses of the predicate, to improve the efficiency of computations (see [9] for more details).

It was shown in Section 3.4 that a pure FAG with basic term interpretation can be seen as a DCP. This opens for applying new evaluation methods for FAG's, based on the procedural semantics of DCP's. This shows that the expertise in logic programming can be also transferred to the field of attribute grammars. However, this issue was not discussed in this paper and requires

further investigation.

There are several differences between DCP's and RAG's. Some of them may deserve further investigation. In particular, it may be interesting to examine whether the following concepts of functional attribute grammars may find some applications in logic programming:

Controlling computations by the dependency relation

Attribute evaluation is controlled by the dependency relation. The partial ordering induced by the dependency relation reflects restrictions on the sequencing of computational operations. The concept of attribute splitting provides a useful tool for characterizing data flow during the computation and facilitates construction of attribute grammars.

Unspecified interpretations.

The formalism of attribute grammars provides no means for defining interpretations. On the other hand, it gives a formal framework for combining semantic rules in one well structured system regardless of the interpretation to be used for performing actual computations.

Many-sorted semantic domains..

Usually it is assumed that different attributes of an attribute grammar may have different domains. This means that many-sorted algebraic structures are considered, what results in a type mechanism, in contrast to the typeless principles of logic programming.

Separate parsing.

The attribute evaluation process begins with a context-free derivation tree which is assumed to be given. Usually it is constructed by a parser from a given terminal string. A similar method can be applied for implementation of the Definite Clause Grammars [20]: the context-free production rules obtained by a modified Construction 3 can be used to separate the parsing process from the rest of the computations.

It is worth noticing that similar concepts appear in logic programming as pragmatic facilities of existing implementations, e.g. integer arithmetic of Prolog, type system in [17], or read-only variables of Concurrent Prolog [23]. By referring to attribute grammars we get a formal framework for systematic treatment of these facilities.

Acknowledgment

The authors are very grateful to Jan Komorowski for stimulating discussions, and to Wlodek Drabent and Bryan Lyles for their comments on the revised version of the paper. We also acknowledge gratefully the criticism of the reviewer, which contributed in many aspects to the revision of the original

version of this paper.

REFERENCES

[1] Abramson, H,. Definite Clause Translation Grammars, in: *Proc. of the 1984 Int. Symposium on Logic Programming, Atlantic City,* 233-241.

[2] Apt, K.R. and van Emden, M.H., Contributions to the Theory of Logic Programming, *J.ACM.* 29:841-862 (1982).

[3] Bruynooghe, M., Adding redundancy to obtain more reliable and readable Prolog programs, in: *Proc. 1st International Logic Programming Conf.*, van Caneghen,M. (ed.), Marseille 1982, 129-133.

[4] Clark, K.L., Predicate Logic as a computational Formalism, Research Monograph 79/59 TOC, Dept. of Computing, Imperial College, London (1979).

[5] Courcelle,B. and Deransart,P., Partial Correctness of Attribute Grammars. INRIA, Rapports de Recherche, No.322 (1984).

[6] Courcelle,B. and Franchi-Zannettacci, P., Attribute Grammars and Recursive Program Schemes, *TCS,* 17: 163-191 and 235-257 (1982).

[7] Deransart, P.: Logical Attribute Grammars, in: *Proc. of the IFIP Congress 1983,* Mason R.E.A. (ed.), Elsevier Science Publishers B.V. (North-Holland), 463-469.

[8] Deransart,P., Jourdain,M. and Lorho,B., Speeding up Circularity Tests for Attribute Grammars, INRIA, Rapports de Recherche, No. 211 (1983) and *Acta Informatica* 21:375-391 (1984)

[9] Deransart,P. and Maluszynski,J., Modelling Data Dependencies in Logic Programs by Attribute Schemata, INRIA, Rapports de Recherche, No.323 (1984).

[10] Deransart,P. and Maluszynski,J., Relating Logic Programs and Attribute Grammars, Linköping University, IDA Research Report 84-07 (1984).

[11] Engelfriet J., Attribute Grammars: Attribute Evaluation Methods, In: *Methods and Tools for Compiler Construction* (B. Lorho ed.) Cambridge University Press, 103-138 (1984).

[12] Engelfriet,J. and File,G., Passes, Sweeps and Visits, Twente University of Technology, Enschede, Memorandum INF 82-6 (1980).

[13] Jazayeri,M., A Simpler Construction for Showing the Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars, *J.ACM* 28:715-720 (1981).

[14] Knuth,D.E., Semantics of Context-Free Languages, *Math. Systems Theory,* 2:127-145 (1968) and 5:95-96 (1971).

[15] Kowalski, R.A., Predicate Logic as a Programming Language, in: *Information Processing 74,* Rosenfeld,J.,(ed.), North-Holland 1974, 556-574.

[16] Mellish, C.S., Automatic Generation of Mode Declarations for Prolog Programs, DAI, Edinburgh, Draft, 1981

[17] Mycroft, A. and O'Keefe, R.A., A Polymorphic Type System for Prolog, *Artificial Intelligence* 23:295-307 (1984).

[18] Naish, L., Automatic Generation of Control for Logic Programs, University of Melbourne, TR 83/6 DCS (1983).

[19] Nilsson, J.F. and Nonfjall, H., A Prolog Compiler. Department of Computer Sc., Technical University of Denmark, Technical Report 1123. 1984

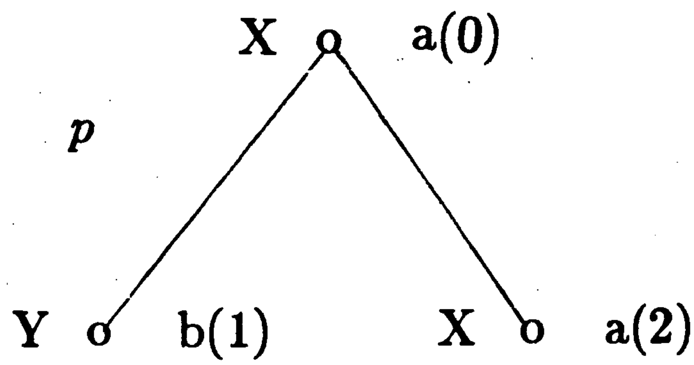
[20] Pereira, F.C.N. and Warren, D.H.D., Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artificial Intelligence*, 13:231-278 (1980).

[21] Plaisted, D.A., The Occur-Check Problem in Prolog, in: *Proc. of the 1984 Int. Symposium on Logic Programming, Atlantic City,* 272-280.

[22] Robinson, J.A., A Machine-Oriented Logic based on the Resolution Principle, *J.ACM* 12:23-41 (1965).

[23] Shapiro, E.Y., A Subset of Concurrent Prolog and its Interpreter, ICOT-TR-003 (1983).

[24] Smolka, G., Making Control and Data Flow in Logic Programs Explicit, in: *ACM Symposium on Lisp and Functional Programming,* Austin 1984, 311-322.



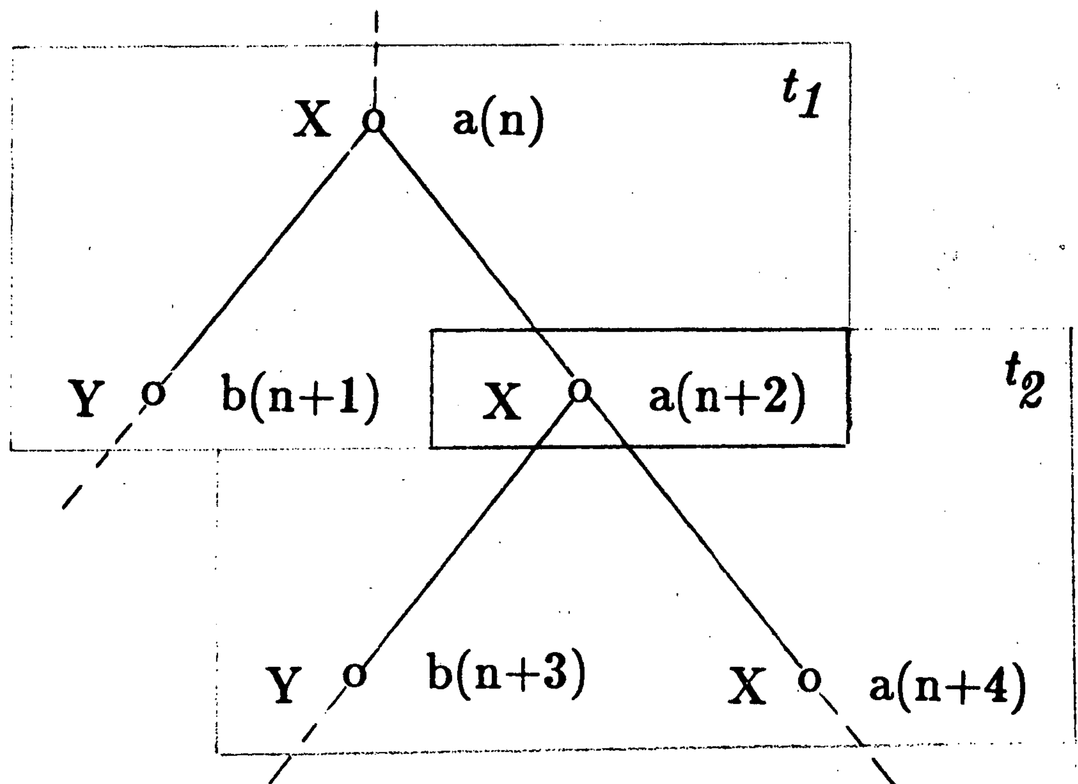


Fig. 1

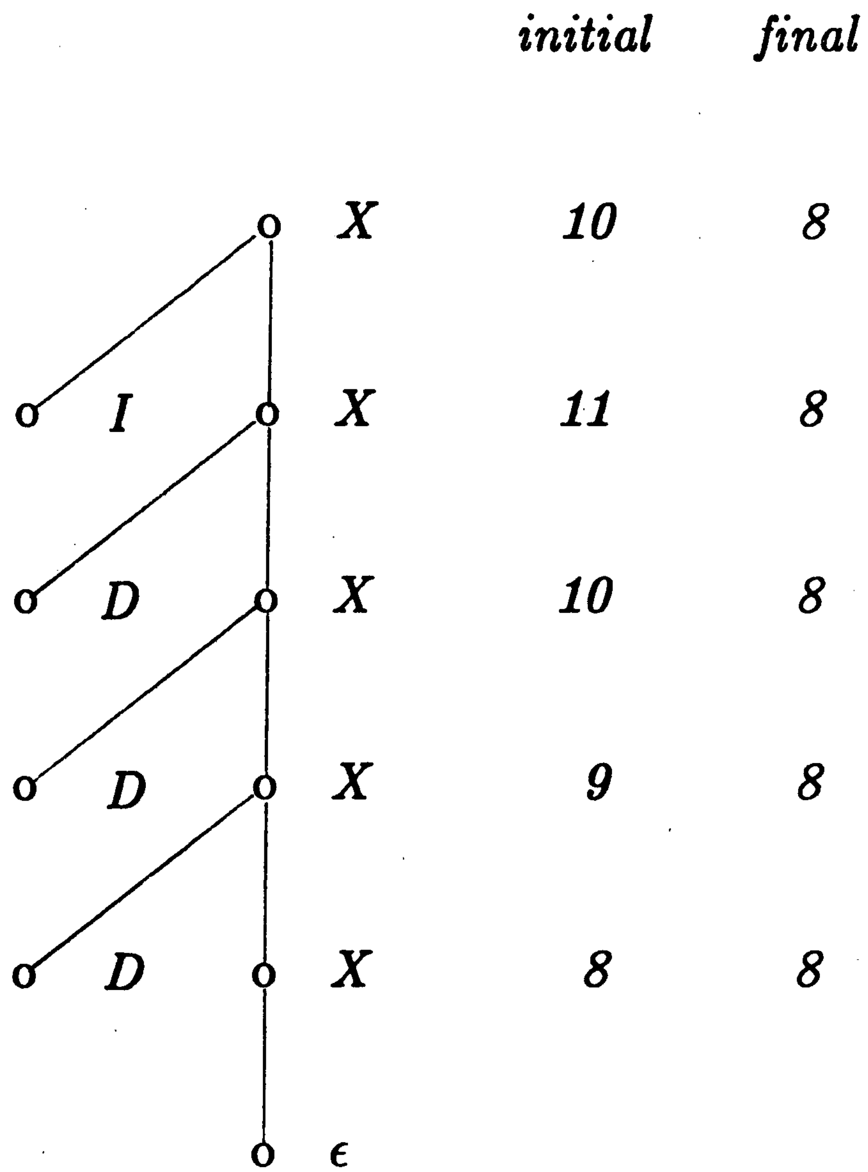


Fig. 2

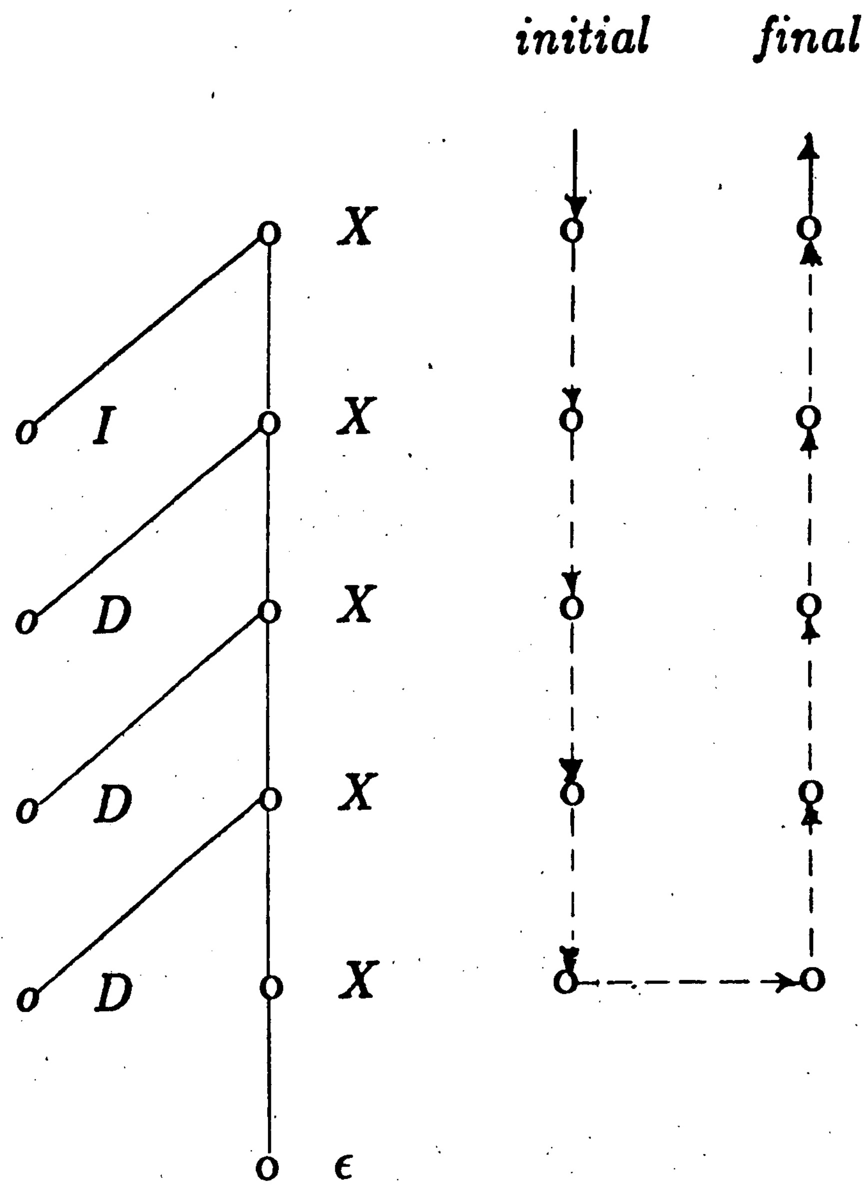
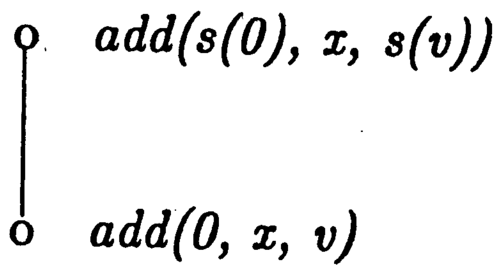


Fig. 3



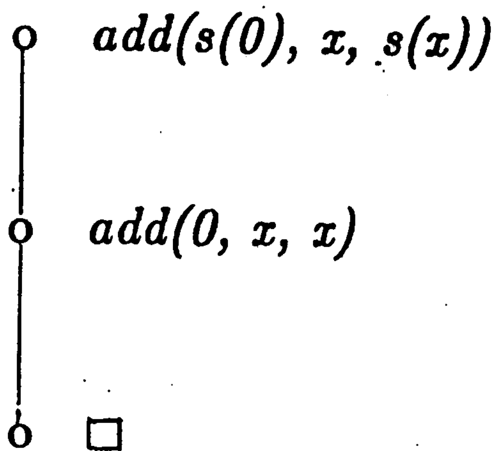
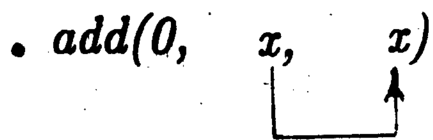


Fig. 4



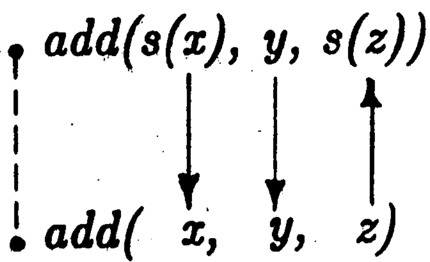


Fig. 5

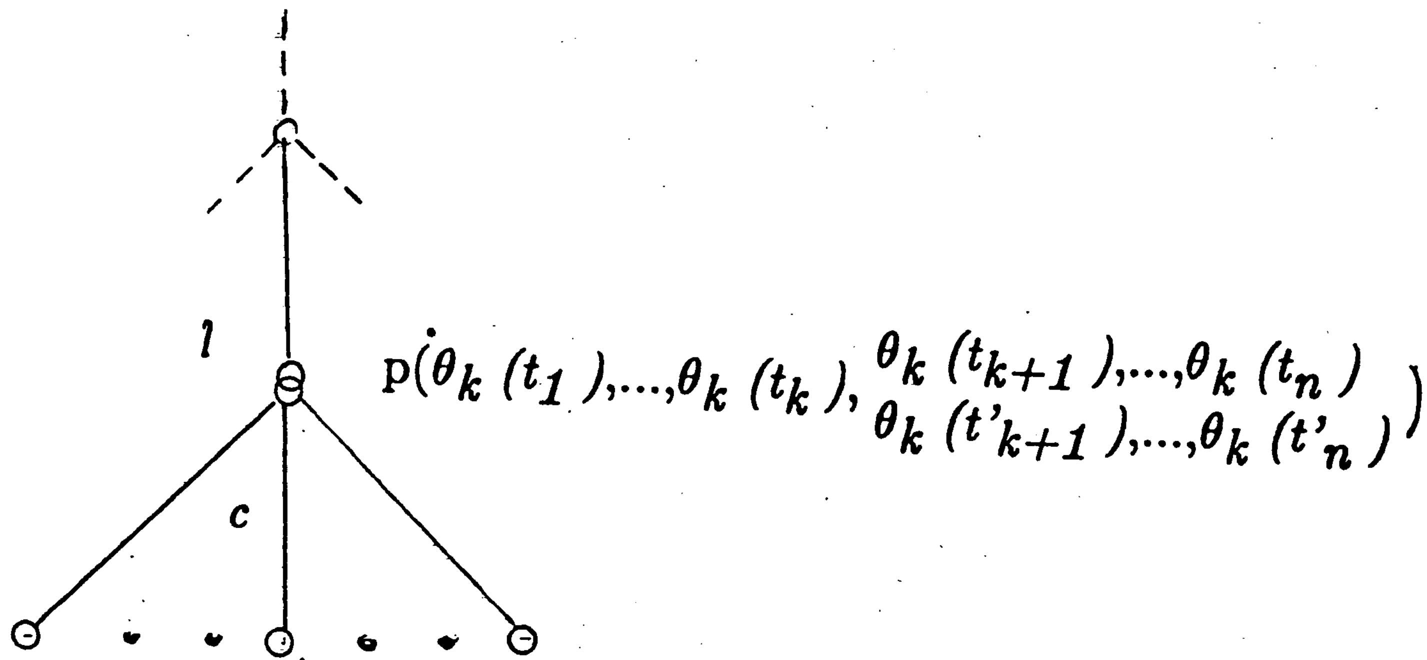


Fig. 6

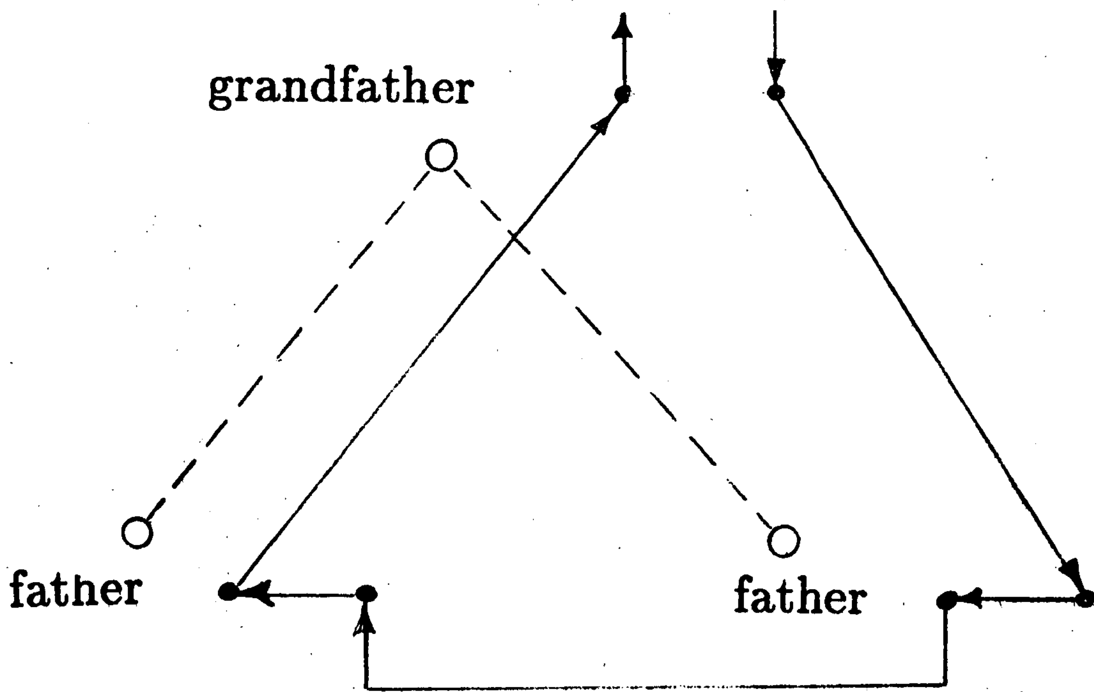


Fig. 7

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