CAN WE TRANSFORM LOGIC PROGRAMS INTO ATTRIBUTE GRAMMARS?

by

Tomas Isakowitz
Information Systems Department
Leonard N. Stern School of Business
New York University
40 West 4th Street
New York, New York 10003

March 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-6

Contents

Abstract

In this paper we study the relationship between Attribute Grammars and Logic Programs, concentrating on transforming logic programs into attribute grammars. This has potential applications in compilation techniques for logic programs. It does not seem possible to transform arbitrary Logic Programs into Attribute Grammars, basically because the same logic variables can sometimes be used as input and sometimes as output. We introduce the notion of an Abstract Attribute Grammar, which is similar to that of an Attribute Grammar with the exception that attributes are not classified into *inherited* and *synthesized*, and that the semantic equations are replaced by *restriction sets*. These sets represent a restriction on the values of attribute occurrences namely, all elements within each set have to be equal. We give an effective translation schema which produces an equivalent Abstract Attribute Grammar for a given Logic Program. We provide a formal proof of this equivalence. We then proceed to classify a class of Abstract Attribute Grammars that can be transformed into Attribute Grammars, and show how to achieve this transformation. By composing both transformations one can transform certain logic programs into attribute grammars. Complete proofs ar5e given.

1 Introduction

This paper studies the possibility of translating Logic Programs (LPs) into Attribute Grammars (AGs). Deransart and Maluszynski [9] show how to perform this translation for a restricted class of LPs. Our method is more general in the sense that it applies to arbitrary LPs. However the formalism into which the translation is performed (AAG) is in some sense weaker than AG. The work presented here originated in [13] and was developed independently by the author. In this paper we have adopted terminology similar to the one used in [9]. As an example of the applications that a translation of the type investigated here has, we point out to work by Attali and Franchi-Zanettacci [2]. They show how one can use attribute evaluation techniques to run TYPOL programs. TYPOL can be regarded as a subset of the class of LPs with which we deal here.

Logic programs are expressively powerful but might be computationally inefficient. How can the computational aspect be improved? One could apply techniques used for other programming languages, mainly those related to compilation. In general, LPs are interpreted. What is the difference between an interpreter and compiler? In a compiler commands are translated into sequences of machine language instructions at compile time. An interpreter translates each command as it is executed. Thus compilers are faster for execution while interpreters are better for development because they support interactive program development.

What can we do about compiling Logic Programs? Let us analyze how an interpreter for LPs works to see which instructions are re-translated every time. Given a LP Γ and a Goal G, the objective is to find out whether $\exists x_1 \ldots x_n G$ (where $x_1, \ldots, x_n$ are all the variables appearing in G) holds in Γ. In general one is interested values for the variables that make the goal G true. In more formal terms, one is interested in a substitution θ such that $\Gamma \vdash \theta(G)$. This is called an *answer substitution.* Let us consider a nondeterministic procedure to solve the problem. Recall that an Logic Program is a set of definite clauses. The *head* of a definite clause $A \leftarrow B_1, \ldots, B_k$ is is A, and its *body* is $B_1, \ldots, B_k$.

Initially the set of goals consists of the initial goal G. The following steps are repeated until the set of goals is empty.

1. Choose a goal g from the set of goals.
2. Pick a clause whose head unifies with the goal g.
3. Add the body of the clause to the set of goals.
4. Apply the unifying substitution obtained in step 2 to the new set of goals.

The procedure is clearly non-deterministic due to: a) Choosing a goal (step 1), b) Choosing a clause (step 2) and c) choosing a unifier (step 2).

Notice that this procedure might not end. It ends either when the set of goals is empty or when step 2 is unsuccessful. In the first case the composition of the substitutions obtained in step 2 is an answer substitution. If the latter case occurs, one can only say that the current branch of the computation is unsuccessful which, due to the nondeterminism of the algorithm does not mean that there exists no answer substitution. One can only negatively

answer the question "is $\exists x_1 \ldots x_n G$ provable from Γ?" provided all branches of computation are unsuccessful.

Different interpreters can deal in different ways with this non-deterministic aspect. A deterministic strategy can also be adopted. One could run an exhaustive breadth-first strategy which considers all possible choices. However, this would be unbearably slow. The standard *Prolog* [5] interpreter regards the set of clauses and the set of goals as ordered sequences. In step 1 it picks the leftmost goal, and in step 2 it pick the first clause, i.e. the one appearing earlier in the program.

One can say that in some sense steps 1 and 2 are re-translated continually by the interpreter. Step 2 is computationally expensive in two ways. It involves searching the program for clauses that could unify with the goal, i.e. candidate clauses. Secondly, a unification algorithm has to be run for each candidate clause and if successful, the resulting substitution is to be applied to the new set of goals in step 4. This operation is tantamount to parameter passing. It is on this aspect that we concentrate, proposing some techniques that might replace the implicit parameter passing represented by unification with a more explicit, directed method. This is how our results relate to *compilation.* Instead of running similar sequences of instructions whenever a unifying clause is obtained, it is possible to pre-compute some of the parameter passing operations.

1.1 Preview of the results

In this paper we study a systematic way of performing the translation from a Logic Program into an Attribute Grammar. Due to the intrinsic difference between the direction-less nature of *logic variables* and the directedness of *attributes* it is not possible to produce a semantically equivalent Attribute Grammar for an arbitrary Logic Program. We introduce a formalism, Abstract Attribute Grammar (AAG), in which there is no classification of the attributes into inherited or synthesized. The attributes are intrinsically directionless, thus we abstract over the notion of direction present in AGs, hence the prefix *Abstract* to our formalism. We give a linear time procedure to translate a LP into a semantically equivalent AAG and we provide a proof of the correctness of this method.

In AAGs equations are not written using the equal sign. Instead of writing the equations $t = t_1, \ldots, t = t_n$, we use a different notation which introduces the *restriction set* $\{t, t_1, \ldots, t_n\}$. The semantics we impose forces all members of a restriction set to be interpreted by identical objects. We feel that this simplifies the notation.

We proceed with an example, leaving the formal definitions for section 3.

Example 1.1 Consider the Logic Program for syntactic addition given by:

```
add(0,Y,Y).
add(s(X),Y,s(Z)) → add(X,Y,Z).
```

Any proof tree for this LP will consist of a single branch ending with an instance of `add(0,Y,Y)`. We define a Grammar with two symbols: a non-terminal symbol **add** and a terminal symbol **end** which represents the end of a computational branch. The terminal

end has no attributes. The predicate add is *ternary*, we will associate with it the non-terminal symbol **add** with three attributes: **x**, **y** and **z**. Using our method we obtain the following AAG:

add $\Rightarrow$ **end**	$\{\mathbf{x}(\epsilon), 0\}$
	$\{\mathbf{y}(\epsilon), \mathbf{z}(\epsilon)\}$
add $\Rightarrow$ **add**	$\{\pi^s(\mathbf{x}(\epsilon)), \mathbf{x}(1)\}$
	$\{\mathbf{y}(\epsilon), \mathbf{y}(1)\}$
	$\{\pi^s(\mathbf{z}(\epsilon)), \mathbf{z}(1)\}$

The function π^s when applied to a term of the form $s(t_1)$ returns t_1. It is a projection function. The idea behind the transformation is to relate different occurrences of the same logic variable by semantic equations. Thus the set $\{\mathbf{y}(\epsilon) = \mathbf{z}(\epsilon)\}$ in the first production comes from the first clause in the LP where Y appears as the second and third argument of *add*.

The difference between an AAG and an AG is that the attributes in AGs are directed and that the equations can be used for computation. Notice that the LP can be used to compute subtraction as well as addition. If we want to transform the above LP into a AG we have to restrict ourselves to a specific behavior of the program. Let us now transform this AAG into an AG that computes the value of **z** as the addition of **x** and **y**. We let **x** and **y** be inherited attributes (since they act as *input*) and **z** be synthesized.

add $\Rightarrow$ **end**	$zero(\mathbf{x}(\epsilon))$
	$\mathbf{z}(\epsilon) := \mathbf{y}(\epsilon)$
add $\Rightarrow$ **add**	$\mathbf{x}(1) := \pi^s(\mathbf{x}(\epsilon))$
	$\mathbf{y}(1) := \mathbf{y}(\epsilon)$
	$\mathbf{z}(\epsilon) := s(\mathbf{z}(1))$

The predicate *zero* is applied to an input attribute and represents a condition under which the production applies. In this sense we have a Functional AG as opposed to a simple AG. Notice that in the last equation we introduced the function s which is the inverse of π^s.

As additional example of the utility of our work, we note that a Functional AG can be transformed into a functional program. In our case, we interpret *add* as a function returning the value of the attribute **z**.

```
(add (λ (x y)
     (COND
          ((= x 0) y)
          (T (s (add (π^s x) y))))))
```

The functions s and π^s are to be interpreted as successor and predecessor, they can also be defined with lambda expressions. This functional program can be compiled into machine code and optimized. Thus, the whole process shows that it is possible to compile some logic programs into machine code. We leave this topic for further research.

As mentioned earlier, it is not just a nuance that in order to obtain an AG we have to restrict ourselves to a specific input/output behavior of a LP. The notion of a *direction assignment* (*d-assig*) presented in [9] provides the ability to talk about the different behaviors of the arguments of predicates appearing in a LP by classifying them into *input* or *output*. This notion extends to AAGs. We show how to obtain an AG from a given AAG and a suitable direction assignment. A proof of the correctness of this transformation is given. Putting both transformations together, we see that starting from a LP and a suitable *d-assig* it is possible to obtain a semantically equivalent AG by transiting through an AAG.

1.2 Related Work

Deransart and Maluszynski [9] show how to transform arbitrary LPs into *Relational Attribute Grammars* (RAGs). In RAGs attribute equations are replaced by first order formulae, thus RAGs are more general than AAGs which only permit equational formulae. Their translation from LPs into RAGs provides an equivalence which is only reflected in the choice of the semantic interpretation of the RAG, while in our case the equivalence is forced by the syntactic qualifications of the AAG. Thus our transformation is more precise.

To summarize, we present a formalism (AAG) which captures via syntactic equations the relations present in LPs and is still generous enough to allow for the representation of arbitrary LPs. We also provide a transformation from a subclass of AAGs into AGs. We give full correctness proofs. We feel that for the purposes of the problems we investigate here, Abstract Attribute Grammars are more suitable than Relational Attribute Grammars. First, the undirected nature of logical variables is better represented by *restriction sets* than it is by general predicates. Second, the the relationship among different positions in a clause is more transparent in our representation. We strongly feel that AAGs should be used as a tool to investigate compilation aspects of Logic Programs.

2 Preliminaries

2.1 Well Formed Terms

Well formed terms are used in the definitions of logic programs and attribute grammars. We explain how terms are build inductively from a set of variables and function symbols. Although the approach is in general many sorted, this is not needed in the scope of our paper. For the sake of simplicity sorts are left out of the discussion.

Terms are defined inductively from a set of function symbols and a set of variables. Each function symbol f takes a predetermined finite number of arguments which is called its *arity* and denoted by $arity(f)$. Function symbols of 0 arity are called *constants.*

Definition 2.1 Given a set V of variables and a set F of function symbols, the set $T_F(V)$ of *well formed terms* is defined inductively as follows:

1. each variable $v \in V$ is a term
2. if f is a function symbol of arity n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is also a term.

From 2 it follows that constants are terms. The set of terms is *freely generated* from the variables by the function symbols. This is important because it allows functions over $T_F(V)$ to be defined recursively. For a discussion of inductive sets, free generation and recursive functions see the second chapter of *Logic for Computer Science* by J. Gallier [11].

Given a syntactic characterization of the set of terms, we would like to interpret these symbols in a coherent manner. That is, we want each function symbol $f \in F$ to stand for a specific function, each constant c to stand for a specific value, and so on. This is formally done by an interpretation.

Definition 2.2 An interpretation $\mathcal{I}$ of a set of terms $T_F(V)$ is a mapping such that:

1. The set of variables is assigned a specific domain $\mathcal{I}(V) = D$ called *the semantic domain of* V.
2. For each function symbol $f \in F$ of arity n, $\mathcal{I}(f)$ is a function from D^n into D. In particular for a constant c, $\mathcal{I}(c) \in D$.

A way of relating variables to their domains is also needed. This is done with valuations. A *valuation* α for a set of variables V is a mapping assigning to each variable an element of its domain, that is: $\alpha(v) \in D$. Valuations are naturally extended to terms as follows. Given a set L of terms, an interpretation $\mathcal{I}$ and a valuation α for variables, α is extended to a valuation $\overline{\alpha}$ of arbitrary terms as follows:

1. for a variable v, $\overline{\alpha}(v) = \alpha(v)$
2. $\overline{\alpha}(f(t_1, \ldots, t_n) = \mathcal{I}(f)(\overline{\alpha}(t_1), \ldots, \overline{\alpha}(t_n))$

From the definition it follows that for constants c, $\overline{\alpha}(c) = \mathcal{I}(c)$. That $\overline{\alpha}$ is well defined follows from its definition and from the inductive definition of terms.

2.2 Logic Languages

A logic language $\mathcal{L}$ is given by a tuple $< \mathcal{P}, \mathcal{F}, \mathcal{V} >$ where:

1. $\mathcal{P}$ is a set of predicate symbols with assigned arities,
2. $\mathcal{F}$ is a set of function symbols,
3. $\mathcal{V}$ is a countably infinite set of variables.

The set of *terms* of $\mathcal{L}$ is given by $T_{\mathcal{F}}(\mathcal{V})$, the set of terms constructed from $\mathcal{V}$ and $\mathcal{F}$, it is also denoted by $Terms(\mathcal{L})$.

Notice that in a logic language $\mathcal{L}$, the set of terms is freely generated from the variables by the function symbols. We can therefore conclude that any function mapping variables in $\mathcal{L}$ to terms in $Terms(\mathcal{L})$ has a unique extension to a function over $Terms(\mathcal{L})$. We use this to define the notion of a substitution. A *substitution* is a mapping from formulae to formulae which replaces some variables by terms in a systematic manner. Formally:

Definition 2.3 Given a function $\theta : Var(\mathcal{L}) \mapsto Terms(\mathcal{L})$, its unique extension to terms $\overline{\theta} : Terms(\mathcal{L}) \mapsto Terms(\mathcal{L})$ is a *substitution.*

We will identify $\overline{\theta}$ with θ. If θ is a substitution and t a term, then $\theta(t)$ is called an *instance* of t.

Atomic formulae are of the form $P(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and P is a predicate symbol of arity n. The set of formulae is built up inductively from the atomic formulae, the logic connectives and the quantifiers. For a more detailed discussion on the formal definition of logic languages see [11]. The semantics are defined via structures and assignments to free variables as usual.

2.3 Definite Clause Programs

Logic Programming deals with the computation of relations specified by logic formulae. This section briefly outlines the main concepts which are used in the sequel. For more details, the reader is referred to the literature [1, 15].

2.3.1 Syntax

We focus our attention on a special type of logic formulae. A *definite clause* is a pair consisting of an atomic formula A and a finite set of atomic formulae $\{B_1, \ldots, B_k\}$, with $k \geq 0$, commonly written as

$$A \leftarrow B_1, \ldots, B_k.$$

In standard logic notation the clause described above is represented by the formula:

$$\forall x_1 \ldots \forall x_n (B_1 \wedge \ldots \wedge B_k) \supset A$$

where $x_1, \ldots, x_n$ are all the variables appearing in $B_1, \ldots, B_k, A$.

Definition 2.4 A *Definite Clause Program* (DCP) is a finite set of definite clauses belonging to a logic language $\mathcal{L}$.

Throughout the rest of this paper when referring to a definite clause program we might use the name *logic program* although the latter constitutes a larger class of programs. (A Logic Program can have clauses without positive litterals.)

2.3.2 Proof Theory

Following [1], a Definite Clause Program is considered to denote its least Herbrand model. It was shown in [4] that one can instead deal with the set of all atomic formulae which are logical consequences of the definite clause program. Each element in this set can be obtained by constructing a *proof tree* having the term as its root. For our purposes it is convenient to consider a definite clause program to be the specification of the set of all its proof trees.

Definition 2.5 A *proof tree* is an ordered labeled tree whose labels are atomic formulae (not necessarily ground). The set of proof trees for a given definite clause program Γ is defined inductively as follows:

1. If $A \leftarrow \emptyset$ (i.e. with empty body) is an instance of a clause of Γ, then the tree consisting of the two nodes whose root is labeled by A and whose only leaf is labeled by *end* is a proof tree.
2. If $T_1, \ldots, T_k$ for some $k > 0$ are proof trees with roots labeled $B_1, \ldots, B_k$ and $A \leftarrow B_1, \ldots, B_k$ is an instance of a clause in Γ, then the tree consisting of the root labeled with A and the subtrees $T_1, \ldots, T_k$ is a proof tree.

Example 2.6 The following definite clause program computes syntactic addition. The number n is represented by $s^n(0)$.

$$add(0, Y, Y)$$
$$add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z)$$

The tree appearing in figure 1 is a proof tree of this definite clause program. It states a proof tree for $2 + 1 = 3$.

2.4 Attribute Grammars

In this section we briefly introduce Attribute Grammars. For a more detailed treatment see [10]. Attribute Grammars were introduced by Knuth [14]. The following definition is inspired from Chirica and Martin [3] and Courcelle and Franchi Zannettacci [6], [7]. Some adaptations have been made to simplify the definition. An *Attribute Grammar* is a pair $(\mathcal{A}, \mathcal{I})$ consisting of a syntactical part $\mathcal{A}$ called an *attribute system*, and a semantic part $\mathcal{I}$ called an *interpretation*. Roughly speaking, an attribute system defines a set Σ of *function*

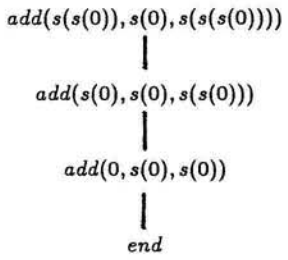

Figure 1: The proof tree for $add(s(s(0)), s(0), s(s(s(0))))$

symbols, a context-free grammar G, a set of *attributes* for each symbol in G, and a set E of semantic equations formed from the function names in Σ and the attributes. Normally the attributes and the function symbols are typed (sorted). This however is not needed in the scope of our paper and we leave it out for the sake of clarity.

Definition 2.7 An *attribute system* $\mathcal{A}$ consists of the following components:

1. A finite set Σ of function symbols.

2. A context-free grammar $G = (N, T, P, Z)$, where N is the set of *nonterminals*, T is the set of *terminals*, $Z \in N$ is the *start symbol*, and $P \subseteq N \times (N \cup T)^*$ is the set of *productions*.

3. With every symbol X of the grammar, a finite set $A(X)$ of *attributes* is associated. The cardinality of $A(X)$ will be denoted by n_X.

4. Two functions $S : X \mapsto 2^{A(X)}$ and $I : X \mapsto 2^{A(X)}$ determine which of the attributes of X are synthesized and which are inherited. If the start symbol Z has inherited attributes, or any terminal symbol has synthesized attributes, the attribute grammar is said to have *parameters*.

5. For every production $p : X_\epsilon \rightarrow X_1 ... X_n$, a finite set E_p of *semantic equations* (or *restrictions*) which satisfies the following constraints. First, the set of *attribute occurrences of* p is

$$ATOC(p) = \{z(\epsilon) \mid z \in A(X_\epsilon)\} \cup \bigcup_{i=1}^{n} \{z(i) \mid z \in A(X_i)\}.$$

Each attribute occurrence $z(i)$ has a *tag* i indicating that it is associated with the grammar symbol X_i in p. This is necessary because the same attribute z may be associated to distinct grammar symbols in the same production and to different occurrences of the same grammar symbol.

We now define the set of equations E_p associated with a production.

i) The only attributes of X_ϵ that are defined in p are synthesized. That is, for every synthesized attribute $a \in S(X_\epsilon)$, there is exactly one equation

$$a(\epsilon) = t^p_{a(\epsilon)},$$

where $t^p_{a(\epsilon)}$ is some term in $T_\Sigma(ATOC(p))$.

ii) Only inherited attributes of the right hand side are defined in p. For every k, $1 \leq k \leq n$, for every inherited attribute $y \in I(X_k)$, there is exactly one equation

$$y(k) = t^p_{y(k)},$$

where $t^p_{y(k)}$ is some term in $T_\Sigma(ATOC(p))$.

It should be noted that a semantic rule is *oriented*, in the sense that it has a left-hand side and a right-hand side which are not interchangeable.

We now turn to the semantics of Attribute Grammars. Our goal is to define the meaning assigned by an attribute grammar to a parse tree. Given an attribute grammar $\mathcal{G} = (\mathcal{A}, \mathcal{I})$, the interpretation $\mathcal{I}$ is used to provide meaning to parse trees. For every parse tree T, Chirica and Martin [3] define a system E_T of equations among variables called *attribute instances*, and show that this system has a least fixed point, which is taken as the semantics of T. For the purpose of our work, we change this definition and take the set of all solutions of E_T to be the semantics of T. Intuitively, attribute instances are copies of attribute occurrences assigned to the nodes of the parse tree, and attribute evaluation consists in computing the values of these instances.

In order to refer to nodes in parse trees, we use an *addressing scheme* due to S. Gorn [12]. The root of the tree receives as address the empty string ϵ. If a node has the address u and this node has exactly n successors, they receive the addresses $u1$,...,un from left to right. An attribute instance is an expression of the form $a(u)$, where a is an attribute of a symbol X, and u is the tree address of a node labeled X in a parse tree. Instead of dealing with parse trees we will use attributed trees in our discussion. These are obtained from parse trees by replacing each node X by the node $X\ \langle X.1(m), \ldots, X.n_X(m)\rangle$ where $X.1, \ldots, X.n_X$ are all the attributes of X and m is the position of the node.

Given an attributed tree T, the set AI_T of *attribute instances* associated with T and the *system of equations* E_T are defined inductively as follows (see [3, 16]).

Definition 2.8 If T is an attributed tree of depth 1, then the production applied at the root is some production $(p) : A \to u$, where $u \in T^*$ is a terminal string. Then,

$$E_T = E_p, \quad \text{and} \quad AI_T = ATOC(p).$$

If T is an attributed tree of depth ≥ 2, then the production applied at the root is some production $p : X_\epsilon \to X_1 \ldots X_n$, where $X_1 \ldots X_n$ contains some nonterminals, say $B_1, \ldots, B_k$. Let $X_1 \ldots X_n = u_1 B_1 u_2 \ldots u_k B_k u_{k+1}$, with $u_1, \ldots, u_{k+1} \in T^*$. Then, if the subtrees rooted at $B_1, \ldots, B_k$ are $T_1, \ldots, T_k$, for $1 \leq i \leq k$, let

$$\begin{aligned} AI'_{T_i} &= \{a(ju) \mid a(u) \in AI_{T_i}\}, \quad \text{with } B_i = X_j \\ E'_{T_i} &= \{(x = t)[a(ju)/a(u),\ a(u) \in AI_{T_i}] \mid (x = t) \in E_{T_i}\}, \\ &\quad \text{with } B_i = X_j, \end{aligned}$$

where $(x = t)[a(ju)/a(u),\ a(u) \in AI_{T_i}]$ is the result of simultaneously substituting $a(ju)$ for every occurrence of $a(u)$, for each $a(u) \in AI_{T_i}$, in the equation $x = t$. Then,

$$\begin{aligned} AI_T &= ATOC(p) \cup AI'_{T_1} \cup \ldots \cup AI'_{T_k}, \quad \text{and} \\ E_T &= E_p \cup E'_{T_1} \cup \ldots \cup E'_{T_k}. \end{aligned}$$

If the attribute grammar has attribute parameters, then for each instance $a(u)$ of an inherited parameter a associated with the root, there is an equation of the form $a(u) = x_0$ where x_0 is an initial value. Similarly the synthesized attributes of the leaves have equations which initialize them.

Notice that the equations in E_T contain terms over $AI(T)$. Denote by $AI(\mathcal{A})$ the set of all attribute instances of the attributed trees of the attribute system $\mathcal{A}$. The terms appearing in E_T are in $T_\Sigma(AI(\mathcal{A}))$, the interpretation part $\mathcal{I}$ of an attribute grammar assigns a specific domain to $AI(\mathcal{A})$ and actual *partial functions* over that domain to the function names in Σ. Given an attributed tree T, a valuation α assigning values in D to all the attribute instances in $AI(T)$ is is *valid* if no term in E_T is undefined, and all the equations in E_T are satisfied. The semantics of T is defined to be the set of all valid valuations of T. The semantics of the attribute grammar $(\mathcal{A}, \mathcal{I})$ is the set of all pairs (T, α) where T is an attributed tree and α a valid valuation for T. Notice that we use *partial* functions to interpret the function symbols.

From a computational point of view it is important that the attributes are split into inherited and synthesized; and that the semantic equations satisfy the conditions *i)* and *ii)* of page 5. These support an algorithm for finding a valid valuation for an attributed tree. The evaluation problem consists in finding a partial order on the attributed tree so that the variable elimination described in the previous example works.

2.5 Conditional Attribute Grammars

A *conditional attribute grammar* is similar to an attribute grammar except that in addition to the equations associated with each production, a predicates on some *input* attribute occurrences are present. We follow [8] for this definition. In order to proceed we need to formalize what we mean by *input* and *output* attribute occurrences.

Intuitively, *Input attribute occurrences* of a production are occurrences in its left hand side of inherited attributes and right hand side occurrences of synthesized attributes. *Output attribute occurrences* of a production are occurrences of synthesized attributes in its left hand side and occurrences of inherited attributes in its right hand side. This is formalized as follows.

Definition 2.9 Given an attribute system $\mathcal{A}$ and a production p of the form $X_\epsilon \Rightarrow X_1 \ldots X_n$, a splitting of attributes into inherited and synthesized via functions I and S induces a splitting of the attribute occurrences of p into *input* and *output* as follows.

$$Input(p) = \{a(\epsilon) | a \in I(X_\epsilon)\} \quad \bigcup \quad \bigcup_{i=1}^{n} \{a(i) | a \in S(X_i)\}$$

$$Output(p) = \{a(\epsilon) | a \in S(X_\epsilon)\} \quad \bigcup \quad \bigcup_{i=1}^{n} \{a(i) | a \in I(X_i)\}$$

In conditional attribute grammars conjunctions of literals on input attribute occurrences are introduced into the productions. The interpretation part of a Conditional Attribute Grammar associates subsets of the corresponding cartesian product to predicates. Given a decorated tree, a valuation will be valid provided not only that the equations are verified, but also that the conjunction of literals of each production is satisfied.

Definition 2.10 A Conditional Attribute Grammar is an Attribute Grammar that contains a set of predicate symbols $\mathcal{P} = P_1, \ldots, P_n$. Each production p has associated, in addition to the equations E_p, a logic formula B_p which is a conjunction of formulas of the form $P_j(t_1, \ldots, t_n)$ or $\neg P(t_1, \ldots, t_n)$ for some n-ary predicate symbol $P \in \mathcal{P}$, and some terms $t_1, \ldots, t_n$ that contain only input attribute occurrences.
The semantics of Conditional Attribute Grammars are sets of decorated trees with valid valuations as in the case of attribute grammars, except that in addition, each valid valuation has to satisfy the the formula B_p.

NOTICE: the fact that the arguments of the predicates B_p are input attribute occurrences is important since this will guarantee the computability of valid valuations. Since satisfying the logic formula B_p will involve just *checking* the values already computed.

Deransart and Maluszynski introduce *Functional Attribute Grammars* (FG) and show their relationship with Logic Programs [9]. They use the name Functional Attribute Grammar for what we here call Conditional Attribute Grammar.

2.6 Relational Attribute Grammars

Relational Attribute Grammars have less structure than Attribute Grammars. Each production has a logic formula associated with it. In order for a valuation for an attributed tree to be valid, it has to satisfy some logic formulae.

Definition 2.11 A Relational Attribute Grammar consists of

1. A finite set Σ of function symbols.
2. A finite set $\mathcal{P}$ of predicate symbols.
3. A Context Free Grammar $G = (N, T, P, Z)$,
4. With every symbol X of the grammar, a finite set $A(X)$ of *attributes* is associated.

   For every production $p : X_\epsilon \rightarrow X_1 \ldots X_n$, a logic formula R_p of the logic language $< \mathcal{P}, \Sigma, ATOC(p) >$, that is, the variables in R_p are attribute occurrences of p.
5. An interpretation which is similar to the interpretation of Attribute Grammars except that each n-ary predicate P is interpreted by a subset of the cartesian product of the domains of the attribute occurrences. The boolean operators receive their normal interpretation.

The semantics of Relational Attribute Grammars are as that of Attribute Grammars, except that RAGs have no computational semantics. Valid valuations for an attributed tree have to satisfy the logic formula associated with each address in the tree, but there is no algorithm for computing valid valuations.

3 Abstract Attribute Grammars

To the best knowledge of the author this formalism is new. It abstracts over the inherited/synthesized splitting of attributes in an attribute grammar. The syntactic part consists of an *abstract attribute system* which differs from an attribute system in that there is no splitting of attributes into inherited and synthesized; and the equations are replaced by *restriction sets.* The semantic part consists of an interpretation $\mathcal{I}$ as before.

Definition 3.1 An *abstract attribute system* $\mathcal{A}$ consists of the following components:

1. A finite set Σ of function symbols.
2. A context-free grammar $G = (N, T, P, Z)$, where N is the set of *nonterminals*, T is the set of *terminals*, $Z \in N$ is the *start symbol*, and $P \subseteq N \times V = N \cup T^*$ is the set of *productions*.
3. With every symbol X of the grammar, a finite set $A(X)$ of *attributes* is associated. The cardinality of $A(X)$ will be denoted by n_X.
4. The set of attribute occurrences is defined as in page 9. For every production $p : X_\epsilon \Rightarrow X_1 \ldots X_n$, a finite set R_p of *restriction sets* whose elements are terms in $T_\Sigma(ATOC(p))$

From any attribute grammar one can obtain an abstract attribute grammar by replacing each equation $a(i) = t$ by the restriction set $\{a(i), t\}$. Abstract Attribute Grammars can be viewed as a special type of Relational Attribute Grammars which use only the *equality* predicate and where a shorthand notation has been introduced to express equalities of the form $x_1 = x-2 = x_3 = \ldots = x_n$ as sets $\{x_1, \ldots, x_n\}$. We think however, that this restriction is important enough to be considered in a class by itself. It stands in an intermediate position inbetween Attribute Grammars and Relational Attribute Grammars because although it does not have a computational component derived purely from its syntactic part as AGs do, it does not push to the semantic level all constraints on its valid attributed trees as RAGs do. We can view AAGs as AGs devoid of procedural connotations but retaining their declarative semantics. In RAGs, all semantics are pushed to the interpretation level by assigning meaning to the different predicates. The difference relies on the fact that the meaning of the only predicate appearing in AAGs (equality) is fixed for all interpretations, while that is not true for the predicates which appear in RAGs. Furthermore, we see that AAGs are the adequate formalism to express the constraints implicit in LPs in grammar form.

Example 3.2 Consider the following abstract attribute grammar which is similar to the functional attribute grammar presented in example 1.1.

$$\begin{array}{ll} \textbf{add} \Rightarrow \textbf{end} & \{\mathbf{0}, X(\epsilon)\} \\ & \{Z(\epsilon), Y(\epsilon)\} \end{array}$$

$$\begin{array}{ll} \textbf{add} \Rightarrow \textbf{add} & \{X(1), p(X(\epsilon))\} \\ & \{Y(1), Y(\epsilon)\} \\ & \{Z(1), p(Z(\epsilon))\} \end{array}$$

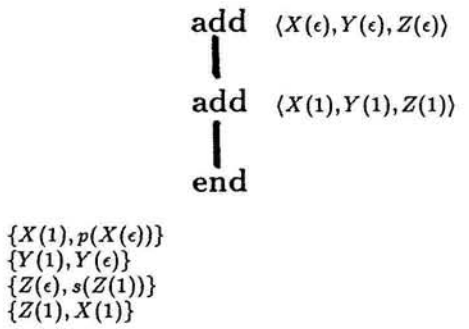

Figure 2: An attributed tree for the abstract attribute grammar

As it turns out, for the purpose of this paper, it is enough to consider just one class of interpretations to be introduced in section 4.3. These interpretations have as domain the set of terms and a fixed interpretation for some of the function symbols. We feel however, that AAG as a formalism should have a more general semantics, as we proceed to present now.

The semantics are defined using the interpretation $\mathcal{I}$. Attributed Trees are defined as for attribute grammars except that instead of a set E_T of equations, a set R_T obtained from the restriction sets of the production instances appearing in T is associated with the tree. Given an attributed tree, a valuation α is *valid* if the elements of each restriction set are assigned identical elements of the domain. If the interpretation assigns *partial functions* to the function symbols, the values assigned to the attribute instances have to belong to the domain of the functions applied to them. The semantics of an abstract attribute grammar is taken to be the set of all its valid attributed trees.

Example 3.3 For the previous example consider the interpretation:

1. D is the set of natural numbers $\mathcal{N}$,
2. p is the predecessor function (subtract 1),

Consider the attributed tree of figure 2. The following valuation is valid:

	$X(\epsilon)$	$Y(\epsilon)$	$Z(\epsilon)$	$X(1)$	$Y(1)$	$Z(1)$
α	1	4	5	0	4	4

Any valid valuation for this tree has to assign 1 to $X(\epsilon)$; $Z(\epsilon)$ will be assigned the successor of $Y(\epsilon)$.

Abstract Attribute Grammars lack an evaluation algorithm. The restrictions are simply stated, no hint as to how a valid valuation could be obtained is given. This is a drawback of the formalism for computational purposes. However, we find it suitable for dealing with logic programs in their full generality since there, as well, the semantics are declarative.

4 Transforming a Logic Program into an Abstract Attribute Grammar

4.1 Overview of the method

Let us explain our approach for converting a Logic Program into an equivalent Abstract Attribute Grammar via an example. Consider the following LP Γ (it decribes syntactic addition)

(p_1) $\quad add(0, Y, Y)$
(p_2) $\quad add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z)$

Any proof tree for this logic program will consist of a single branch ending in an instance of $add(0, Y, Y)$. The syntactic component of the proof trees of Γ is captured by the Context Free Grammar G given by:

(p_1') $\quad \mathbf{add} \Rightarrow \mathbf{end}$
(p_2') $\quad \mathbf{add} \Rightarrow \mathbf{add}$

By "erasing" the arguments of the predicate *add* in a proof tree one obtains a parse tree of G. However, these parse trees lack information about values appearing as arguments of the predicate *add*. In order to represent these arguments, we associate three attributes a, b, c with the nonterminal **add**, each corresponding to an argument of the predicate *add*. What restrictions should one place upon these attributes? We will introduce a restriction for each production and each variable or constant appearing in its corresponding clause. We adopt the following naming convention. $Paths(p, \tau)$ is the restriction related to clause p that deals with the variable (or constant) τ.

Let us look at clause (p_1). The constant 0 appears as the first argument of *add*. We represent this by introducing the restriction:

$$Paths(p_1, 0) = \{0, a(\epsilon)\}$$

The idea is that all members of $Paths(p_1, 0)$ should be made identical. Since 0 is a constant, this forces , $a(\epsilon) = 0$. We deal in a similar fashion with the variable Y. It appears as the second and third argument of *add*. We introduce the restriction:

$$Paths(p_1, Y) = \{b(\epsilon), c(\epsilon)\}$$

This set represents all positions in which the variable Y occurs. One wants to force all members of that set to be equal. Let us deal now with the second clause:

(2) $\quad add(s(X), Y, s(Z)) \rightarrow add(X, Y, Z)$

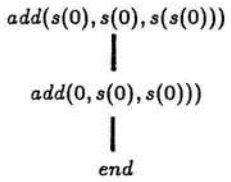

Figure 3: The proof tree for $add(s(0), s(0), s(s(0)))$.

We introduce a projection function π^s on terms. When applied to a term of the form $s(t)$, it returns the term t. If the argument of π^s is not of that form, the result is $\perp$, the undefined value. If one thinks of s as the successor function, one can think of p as the predecessor function. In order to state the restrictions corresponding to clause (2). We take one variable at a time and build a set representing all positions in which that variable occurs. For example X appears in the first argument of the left and right occurrences of *add*. On the left side it is the subterm of the first argument. This occurrence is denoted by $\pi^s(a(\epsilon))$. On the right side, it occurs as the first argument of *add*. Grouping these occurrences we obtain:

$$Paths(p_2, X) = \{\pi^s(a(\epsilon)), a(1)\}$$

The case of Y is simpler since it occurs directly as the second argument of the left and the right appearances of *add*. The set is:

$$Paths(p_2, Y) = \{b(\epsilon), b(1)\}$$

The case of Z is similar to that of X:

$$Paths(p_2, Z) = \{\pi^s(c(\epsilon)), c(1)\}$$

These are all the restrictions needed for this clause. Intuitively, a valuation is acceptable if it induces identity within each set. This should be clear by the way those sets were defined, they represent all occurrences of a variable or constant.

Our claim is that the logic program Γ and the abstract attribute grammar G' consisting of G and the sets described above, are equivalent. That is, the set of proof trees of Γ and the set of valid attributed trees of G' are equivalent.

Let us look at a proof tree for $add(s(0), s(0), s(s(0)))$ appearing in figure 3.

The attributed tree corresponding to it appears in figure 4. The valuation corresponding to the proof tree of figure 3 is:

	$a(\epsilon)$	$b(\epsilon)$	$c(\epsilon)$	$a(1)$	$b(1)$	$c(1)$
α	$s(0)$	$s(0)$	$s(s(0))$	0	$s(0)$	$s(0)$

Similarly, for any valid attributed tree there is an isomorphic proof tree. Let us analyze what all valid attributed trees of the form given in figure 4 are. The restriction $\{0, a(1)\}$ fixes the value $a(1)$ as 0. The restriction $\{\pi^s(a(\epsilon)), a(1)\}$ forces $a(\epsilon) = s(a(1)) = s(0)$. The values of $b(\epsilon)$, $b(1)$ and $c(1)$ have to be identical , and provided $c(\epsilon)$ is of the form $s(t)$, the

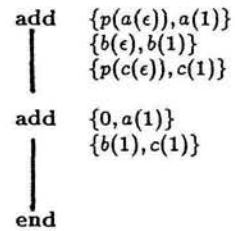

Figure 4: The corresponding attributed tree.

restriction $\{\pi^s(c(\epsilon)), c(1)\}$ forces $c(\epsilon) = s(b(\epsilon))$. It follows that the only possible values at the root of the tree are: $a(\epsilon) = s(0), c(\epsilon) = s(b(\epsilon))$. So the value of the third argument has to be the successor of the second argument. This is exactly what the logic program does. Notice that we need not specify which of the arguments are input and which are output. There is however, a problem with the following valuation:

$$\begin{aligned} a(\epsilon) &= s(0) \\ b(\epsilon) &= \perp \\ c(\epsilon) &= 0 \\ a(1) &= 0 \\ b(1) &= \perp \\ c(1) &= \perp \end{aligned}$$

This valuation satisfies the restrictions of the attributed tree but does not correspond to any proof tree. The reason is the occurrence of $\perp$. In this case it stems from the fact that $c(\epsilon)$ is not a term of the form $s(t)$. We rule out solutions which are undefined for some elements of a restriction set. This corresponds to imposing some structure on certain attributes (in this case $c(\epsilon)$).

We now proceed to give a formal treatment of the ideas just exposed.

4.2 The path function symbols

In section 4.1 we defined a function π^s that "stripped" the s from terms of the form $s(t)$. We are interested in functions that denote all occurrences of a variable or a constant within a term. Throughout this paper we use the word *atom* to refer to a variable or a constant.

Let $\mathcal{L}$ be the logical language in which the definite clauses are written. Recall that $Terms(\mathcal{L})$ denotes the set of terms of $\mathcal{L}$. Also let $Var(\mathcal{L})$ be the set of variables of $\mathcal{L}$, $Func(\mathcal{F})$ the set of function symbols of $\mathcal{L}$ and let $Atoms(\mathcal{L})$ be the set of atoms (i.e. variables and constants) of $\mathcal{L}$. For each term t, let $Atoms(t)$ denote the set of variables and constants appearing in t.

For each function symbol f of arity n appearing in $Func(\mathcal{F})$ we introduce n function symbols $\pi_1^f, \ldots, \pi_n^f$. These represent functions that select each of the arguments of f and are called *selectors*. We also need a function symbol for the identity: id, and a new constant $\perp$ to denote the *undefined value*. The collection of all selector function symbols, $\perp$ and id will be denoted by $\Sigma_{Func(\mathcal{L})}$. When this does not lead to confusion, we drop the $Func(\mathcal{L})$ from $\Sigma_{Func(\mathcal{L})}$. Let $\circ$ denote the concatenation of function symbols, and let Σ^* denote the

set of finite strings over Σ. Given a term t and an atom x appearing in it, we are interested in specifying all access paths to x. This is done as follows.

Definition 4.1 For each term $t \in Terms(\mathcal{L})$ and each $x \in Atoms(\mathcal{L})$, we define a set of terms $\Phi(t,x)$ in Σ^* recursively as follows.

1. if $x \notin Atoms(t)$ then $\Phi(t,x) = \emptyset$;
2. if $x = t$ then $\Phi(t,x) = \{id\}$
3. if $t = f(t_1, \ldots, t_n)$ then

$$\Phi(t,x) = \bigcup_{i=1}^{i=n} \left\{ \pi_i^f \circ \tau \mid \tau \in \Phi(t_i,x) \right\}$$

The elements of $\Phi(t,x)$ will be called *paths.*

Example 4.2 Consider $t = f(g(z,x),x)$ and let us compute $\Phi(t,x)$. Since x appears twice, we should get two paths.

$$\Phi(t,x) = \left\{ \pi_1^f \circ \tau \mid \tau \in \Phi(g(z,x),x) \right\} \bigcup \left\{ \pi_2^f \circ \tau \mid \tau \in \Phi(x,x) \right\} \quad (1)$$

Clearly $\Phi(x,x) = \{id\}$, hence the last component of the union evaluates to $\left\{\pi_2^f \circ id\right\}$. In order to find the value of the first expression we have to compute $\Phi(g(z,x),x)$.

$$\begin{aligned} \Phi(g(z,x),x) &= \{\pi_1^g \circ \tau \mid \tau \in \Phi(z,x)\} \bigcup \{\pi_2^g \circ \tau \mid \tau \in \Phi(x,x)\} \\ &= \emptyset \bigcup \{\pi_2^g \circ id\} \\ &= \{\pi_2^g \circ id\} \end{aligned}$$

Replacing $\Phi(g(z,x),x)$ for its value in equation 1 we obtain

$$\Phi(t,x) = \left\{ \pi_1^f \circ \pi_2^g \circ id, \pi_2^f \circ id \right\}$$

NOTE: The fact that *id* appears at the tail of each member of $\Phi(t,x)$ is a technicality. It arises from our recursive definition. Since concatenation will be interpreted as function composition and *id* will be interpreted as the identity function, we might as well erase the trailing *ids*.

4.3 The termal interpretation

Notice that the elements of $\Phi(t,y)$ are strings in Σ^*, with concatenation denoted by $\circ$. However, when considering the set $T_\Sigma(V)$ of terms build up from a set of variables V and the function symbols in Σ, we can uniquely relate elements in $T_{\Sigma^*}(V)$ with elements of $T_\Sigma(V)$ by interpreting $\circ$ as function composition. This is done by the mapping

$$(\pi_1 \circ \pi_2 \circ \ldots \circ \pi_n)(t) \mapsto \pi_n(\ldots(\pi_2(\pi_1(t)))\ldots)$$

Notice the reversed order. For example, $\pi_1^f \circ \pi_2^g \circ id(t)$ is mapped to $id(\pi_2^g(\pi_1^f(t)))$.

Let $TERMS$ denote the set of all terms in $\mathcal{L}$. We interpret π_i^f as a function $\overline{\pi_i^f}$ on $TERMS$ as follows:

$$\overline{\pi_i^f}(t) = \begin{cases} t_i & \text{if } t = f(t_1, \ldots, t_n) \\ \perp & \text{otherwise} \end{cases}$$

Let us also interpret id as the identity on $TERMS$. We now have an interpretation for all of Σ^*, which we will call NAT (since it is *natural*). For each element π in Σ^*, let $\overline{\pi}$ denote its interpretation in NAT. Since our motivation for defining the sets $\Phi(t, x)$ was to specify all access paths to x in t, we should have that for any $\pi \in \Phi(t, x)$,

$$\overline{\pi}(t) = x$$

One can easily proof this by an inductive argument on the complexity of t. We show how this works with an example.

Example 4.3 Let $t = f(g(z, x), x)$. In example 4.2 above we showed that

$$\Phi(t, x) = \left\{\pi_1^f \circ \pi_2^g \circ id, \pi_2^f \circ id\right\}$$

Thus $\overline{(\pi_1^f \circ \pi_2^g \circ id)}(t)$ should evaluate to x.

$$\begin{aligned} \overline{(\pi_1^f \circ \pi_2^g \circ id)}(t) &= \overline{id}(\overline{\pi_2^g}(\overline{\pi_1^f}(t))) \\ &= \overline{id}(\overline{\pi_2^g}(\overline{\pi_1^f}(f(g(z, x), x)) \\ &= \overline{id}(\overline{\pi_2^g}(g(z, x))) \\ &= \overline{id}(x) \\ &= x \end{aligned}$$

Since the path functions will be used in transforming logic trees, it is interesting to study how they interact with substitutions. In particular we are interested in knowing whether they commute with substitutions.

Example 4.4 Let $t = g(x)$ and let θ be a substitution such that $\theta(x) = h(a)$. We see that θ and $\overline{\pi_1^g}$ commute:

$$\begin{aligned} \theta(\overline{\pi_1^g}(t)) &= \theta(\overline{\pi_1^g}(g(x))) = \theta(x) = h(a) \\ \overline{\pi_1^g}(\theta(t)) &= \overline{\pi_1^g}(\theta(g(x))) = \overline{\pi_1^g}(g(h(a))) = h(a) \end{aligned}$$

However, if we try the same with $\overline{\pi_1^g \circ \pi_1^h}$, it does not work:

$$\begin{aligned} \theta(\overline{\pi_1^g \circ \pi_1^h}(t)) &= \theta(\overline{\pi_1^h}(\overline{\pi_1^g}(g(x)))) = \theta(\overline{\pi_1^h}(x)) = \theta(\perp) \\ \overline{\pi_1^g \circ \pi_1^h}(\theta(t)) &= \overline{\pi_1^h}(\overline{\pi_1^g}(\theta(g(x)))) = \overline{\pi_1^h}(\overline{\pi_1^g}(g(h(a)))) = h(a) \end{aligned}$$

The reason that the second function does not commute with θ is that $\overline{\pi_1^g \circ \pi_1^h}(t)$ is undefined. We can also show that those are the only cases in which this happens.

Lemma 4.5 For any $\pi \in \Sigma^*$, any term t and any substitution θ

$$\text{if } \overline{\pi}(t) \neq \perp \text{ then } \theta(\overline{\pi}(t)) = \overline{\pi}(\theta(t))$$

Proof: One proceeds by induction on the length of π. We show here the base case for π of length 1, i.e. $\pi \in \Sigma$. The inductive argument follows easily.

From the definition of Σ, π is either the identity or of the form π_i^f for some f and some i. In the first case the result clearly holds. In the second case since $\overline{\pi}(t) \neq \perp$, t is of the form $f(t_1, \ldots, t_n)$. Therefore,

$$\begin{aligned} \theta(\overline{\pi_i^f}(f(t_1, \ldots, t_n))) &= \overline{\theta(t_i)} \\ &= \overline{\pi_i^f}(f(\theta(t_1), \ldots, \theta(t_n))) \\ &= \overline{\pi_i^f}(\theta(t)) \end{aligned}$$

□

4.4 The construction

Given a logic program Γ, we show how to construct an abstract attribute grammar by transforming each clause into a production and its restriction sets. First however, we have to define the set of terminals and non-terminals of the Abstract Attribute Grammar and their attributes. For each predicate P of arity n_P appearing in Γ, there is a nonterminal P with n_P attributes $P.1, \ldots, P.n_P$. Each attribute corresponds to an argument position of the predicate P. The only non-terminal is *end*, it has no attributes. Given a clause

$$(p) \quad P_\epsilon(t_1^\epsilon, \ldots, t_{n_{P_\epsilon}}^\epsilon) \leftarrow P_1(t_1^1, \ldots, t_{n_{P_1}}^1), \ldots, P_l(t_1^l, \ldots, t_{n_{P_l}}^1)$$

we associate with it the context free production

$$(p') \quad P_\epsilon \Rightarrow P_1 \ldots P_l$$

Example 4.6 Consider the following clauses from a program for symbolic differentiation.

(a) $dif(\times(U,V), X, +(\times(B,U), \times(A,V))) \leftarrow dif(U,X,A), dif(V,X,B)$
(b) $dif(X,X,1)$
(c) $dif(u,v,0)$
(d) $dif(v,u,0)$

The corresponding abstract attribute grammar will have one non-terminal dif with three attributes $dif.1, dif.2$ and $dif.3$. The productions are:

(a') $dif \Rightarrow dif\ dif$
(b') $dif \Rightarrow end$
(c') $dif \Rightarrow end$
(d') $dif \Rightarrow end$

There is a one to one correspondence between the terms that appear as arguments of the predicates in clause p and the attributes of the non-terminals appearing in the production p'. The nomenclature used makes this clear. One can define a mapping $tr_{p'}$ from the set of attribute occurrences to the set of terms which represents this correspondence:

$$tr_{p'}(P_j.i(k)) = t_i^k$$

The restriction sets associated with the clause make sure all occurrences of the same variable have the same value. This is achieved by forcing all paths leading to that variable to be equal. The sets $\Phi(t, x)$ defined in section 4.2 are used to this end. For a given attribute occurrence σ, every member of $\Phi(tr_{p'}(\sigma), x)$ is applied to σ, all these expressions are collected into the set $Paths(p', x)$:

$$Paths(p', x) = \bigcup_{\sigma \in ATOC(p')} \{\tau(\sigma) \mid \tau \in \Phi(tr_{p'}(\sigma), x)\}$$

In other words, every path leading to x in a term is applied to its corresponding attribute.

Example 4.7 Take the first clause of the logic program appearing above:

$$(a) \quad dif(\times(U, V), X, +(\times(B, U), \times(A, V))) \leftarrow dif(U, X, A), dif(V, X, B)$$

The correspondence between attributes and terms is:

$$\begin{aligned} tr_{p'}(dif.1(\epsilon)) &= \times(U, V) \\ tr_{p'}(dif.2(\epsilon)) &= X \\ tr_{p'}(dif.3(\epsilon)) &= +(\times(B, U), \times(A, V)) \end{aligned}$$

$$\begin{aligned} tr_{p'}(dif.1(1)) &= U \\ tr_{p'}(dif.2(1)) &= X \\ tr_{p'}(dif.3(1)) &= A \end{aligned}$$

$$\begin{aligned} tr_{p'}(dif.1(2)) &= V \\ tr_{p'}(dif.2(2)) &= X \\ tr_{p'}(dif.3(2)) &= B \end{aligned}$$

Let us calculate $Paths(a', V)$. Since $\Phi(t, V) = \emptyset$ whenever V does not appear in t, one only needs to consider the following sets:

$$\begin{aligned} &\Phi(tr_{p'}(dif.1(\epsilon)), V) = \Phi(\times(U, V), V) = \left\{\pi_2^{\times} \circ id\right\} \\ &\Phi(tr_{p'}(dif.3(\epsilon)), V) = \Phi(+(\times(B, U), \times(A, V)), V) = \left\{\pi_2^{+} \circ \pi_2^{\times} \circ id\right\} \\ &\Phi(tr_{p'}(dif.1(2)), V) = \Phi(V, V) = \{id\} \end{aligned}$$

Collecting all these, one obtains

$$Paths(a', V) = \left\{\pi_2^{\times} \circ id(dif.1(\epsilon)), \pi_2^{+} \circ \pi_2^{\times} \circ id(dif.3(\epsilon)), id(dif.1(2))\right\}$$

In a similar fashion one obtains the restrictions for X, U, B and A.

We have described how to deal with the variables. The constants are treated similarly, except that one adds the constant itself to the set.

Example 4.8 The clause $dif(X, X, 1)$ of example 4.6 above is translated into the following abstract attribute grammar production.

$$(b') \quad dif \Rightarrow end \quad \begin{array}{l} Paths(b', X) = \{id(dif.1(\epsilon)), id(dif.2(\epsilon))\} \\ Paths(b', 1) = \{1, id(dif.3(\epsilon))\} \end{array}$$

This construction is repeated for each clause of the logic program. As a result one obtains a production for each clause, each production has as many restriction sets as variables and constants appear in the original definite clause. We now turn to study which values satisfy the restrictions imposed by the sets $Paths(p, x)$.

Consider the interpretation NAT for Σ^* presented in section 4.3. The domain of attributes is $TERMS$, the function symbols π are interpreted as selectors, id as the identity function on $TERMS$. If we substitute for each attribute occurrence σ its corresponding term $tr_{p'}(\sigma)$, all elements of $Paths(p', x)$ should evaluate to x. Furthermore, by assigning substitutions of the corresponding terms to the attributes, the restrictions should be satisfied as well. This is proven in the next lemma.

Lemma 4.9 Every element $\tau(\sigma) \in Paths(p', x)$ satisfies

$$\begin{array}{ll} (1) & \overline{\tau}(tr_{p'}(\sigma)) = x \\ (2) & \text{for any substitution } \theta, \\ & \quad \overline{\tau}(\theta(tr_{p'}(\sigma))) = \theta(x) \end{array}$$

Thus any valuation of the form $\alpha(\sigma) = \theta(tr_{p'}(\sigma)))$ satisfies all restriction sets of the production p'.

Proof: By definition, if $\tau(\sigma) \in Paths(p', x)$ then $\tau \in \Phi(tr_{p'}(\sigma), x)$. As explained on page 19 this implies $\overline{\tau}(tr_{p'}(\sigma)) = x$, which is exactly (1).

In order to prove (2) notice that since $\overline{\tau}(tr_{p'}(\sigma)) = x$, it is the case that $\overline{\tau}(tr_{p'}(\sigma)) \neq \perp$ hence by lemma 4.5 on page 20 any substitution θ will commute with $\overline{\tau}$ on $tr_{p'}(\sigma)$, thus

$$\overline{\tau}(\theta(tr_{p'}(\sigma))) = \theta(\overline{\tau}(tr_{p'}(\sigma))) = \theta(x)$$

□

Example 4.10 In page 21 we calculated $Paths(a', V)$:

$$Paths(a', V) = \left\{\pi_2^{\times} \circ id(dif.1(\epsilon)), \pi_2^{+} \circ \pi_2^{\times} \circ id(dif.3(\epsilon)), id(dif.1(2))\right\}$$

There we also had that $tr_{a'}(dif.1(\epsilon)) = \times(0, V)$. Clearly,

$$\begin{array}{ll} \overline{\pi_2^{\times} \circ id}(tr_{a'}(dif.1(\epsilon))) & = \overline{\pi_2^{\times} \circ id}(\times(0, V)) \\ & = V. \end{array}$$

The previous lemma shows that the terms obtained by applying a substitution to the terms appearing in a clause satisfy the restriction sets of the corresponding production. They are therefore good candidates for valid valuations. Notice that for these valuations, none of the elements of a restriction set evaluates to the undefined value. The lemma just proved shows that substitutions of the original terms are valid valuations. With the aid of the following lemma one shows the converse, namely that any valid valuation of a production can be obtained by applying a substitution to the terms of the original clause.

Lemma 4.11 Let t, t' be terms so that:

1. For any variable x appearing in t and for any $\tau \in \Phi(t,x)$, $\overline{\tau}(t') \neq \perp$,
2. For any variable x appearing in t and for any $\tau_1, \tau_2 \in \Phi(t,x)$, $\overline{\tau_1}(t') = \overline{\tau_2}(t')$,
3. for any constant b appearing in t and for any $\tau \in \Phi(t,b)$, $\overline{\tau}(t') = b$,

then there exists a substitution θ such that $t' = \theta(t)$.

Proof: Intuitively condition 1 tells us that t and t' have similar structure and condition 2 that all occurrences of a variable in t are substituted for the same term in t'. Condition 3 makes sure that constants appear in the same places in t and in t'.

Notice that a substitution is uniquely determined by the value it assigns to variables. We will define the substitution θ by specifying the value of $\theta(x)$ for every variable x appearing in t.

Let x be a variable of t, and let τ be a member of $\Phi(t,x)$ we define $\theta(x)$ as follows:

$$\theta(x) \stackrel{df}{=} \overline{\tau}(t')$$

It is not clear that θ is well defined, i.e. is independent of the choice of τ. Condition 2 forces any two members of $\Phi(t,x)$ to agree on the value they assign to t hence θ is well defined. Next we prove that θ is such that $t' = \theta(t)$. The proof is by induction on the complexity of t.

1. If t is a constant b then $\Phi(t,b) = \{id\}$. By condition (3) we have that $\overline{id}(t') = b$, hence $t' = b$ thus $\theta(t) = t = b = t'$.
2. If t is a variable x, then $\Phi(t,x) = \{id\}$ and τ is the identity. By condition (1) we have that $\overline{id}(t') \neq \perp$ therefore $t' \neq \perp$, and $\theta(t) = \theta(x) = \overline{id}(t') = t'$.
3. Suppose now that $t = f(t_1, \ldots, t_n)$. For any variable or constant x of t, any $\tau \in \Phi(t,x)$ has to be of the form $\tau = \pi_i^f \circ \tau'$ with τ' an element of Σ^*. Since $\overline{\tau}(t') = \overline{(\pi_i^f \circ \tau')}(t') \neq \perp$ we conclude that t' is of the form: $f(t'_1, \ldots, t'_n)$. We show that for $1 \leq i \leq n$, $\theta(t_i) = t'_i$. This implies $\theta(t) = t'$.

   Let us concentrate on $i = 1$, the treatment being the same for the other cases. Notice that any variable or constant appearing in t_1 has to appear in t as well. Therefore $\tau \in \Phi(t_1,x)$ if and only if $\pi_1^f \circ \tau \in \Phi(t,x)$. It is easy to show that t_1 and t'_1 satisfy conditions 1–3 of the lemma. By inductive hypothesis we conclude that the substitution θ_1 defined by $\theta_1(x) \stackrel{df}{=} \tau_1(t'_1)$ where $\tau_1 \in \Phi(t_1,x)$, is such that $\theta(t_1) = t'_1$.

We show that θ and θ_1 agree on all variables of t_1. By definition, $\tau_1 \in \Phi(t_1, x)$ implies $\pi_1^f \circ \tau_1 \in \Phi(t, x)$. Since the τ used to define the value of θ on x is also an element of $\Phi(t, x)$, condition 2 allows us to conclude $\overline{\pi_1^f \circ \tau_1}(t') = \overline{\tau(t')}$. Clearly, $\theta(x) = \overline{\tau}(t') = \overline{\pi_1^f \circ \tau_1}(t') = \overline{\tau_1}(t'_1) = \theta_1(x)$. Since θ and θ_1 agree on all variables of t_1 we have that $\theta(t_1) = \theta_1(t_1) = t'_1$ as wanted.

□

Corollary 4.12 Given a definite clause p for any valid valuation α of its corresponding production p' one can effectively construct a substitution $\theta_{p,\alpha}$ such that for each attribute occurrence σ of p',

$$\alpha(\sigma) = \theta_{p,\alpha}(tr_{p'}(\sigma))$$

Proof: Consider a valid valuation for the attribute occurrences of a production p'. Let $tr_{p'}(\sigma)$ be t and $\alpha(\sigma)$ be t' in the hypothesis of lemma 4.11. The first condition of the lemma is satisfied because valid valuations are required to be defined on all members of the restriction sets, since α is valid and $t' = \alpha(t)$. Conditions 2 and 3 are satisfied because α is valid, i.e. all elements of each restriction set agree on their value. Thus the lemma allows us to conclude that there exists a substitution $\theta_{p,\alpha}$ such that $t' = \theta_{p,\alpha}(t)$. This is exactly what we set out to prove. □

We now show that there is a one to one correspondence between the set of proof trees of a logic program and the set of valid attributed trees of the abstract attribute grammar obtained from it.

Given a proof tree T, one constructs an attributed tree $A(T)$ with valid valuation α_T as follows. If the subtree at position m of T is an instance of clause

$$(p) \quad P_\epsilon(t_1^\epsilon, \ldots, t_{n_{P_\epsilon}}^\epsilon) \leftarrow P_1(t_1^1, \ldots, t_{n_{P_1}}^1), \ldots, P_l(t_1^l, \ldots, t_{n_{P_l}}^1)$$

and the production constructed from p is

$$(p') \quad P_\epsilon \Rightarrow P_1 \ldots P_l$$

then the subtree corresponding to p' appearing at position m of $A(T)$ is shown in figure 5.

Clearly there is a one to one correspondence between the attribute instances of $A(T)$ and the arguments of the predicates in T. Let this correspondence be denoted by $\hat{tr}$. We show that the valuation α_T defined as $\alpha_T(\sigma(m)) \stackrel{df}{=} \hat{tr}(\sigma)$ is valid. Let m be a position in $A(T)$ other than the root or a leaf, and let $N_m = P_i\ \langle P_i.1(m), \ldots, P_i.n_{P_i}(m)\rangle$ be the node appearing there. N_m belongs to the instances of two productions, its lower and its upper production. We have to show that α_T is valid for both of them. Consider the lower production p' . Since T is a proof tree, the subtree of depth 1 rooted at position m is an instance of a definite clause p with substitution θ. Furthermore, p is the clause from which p' was obtained. Hence α_T operates on every attribute instance $\sigma(m)$ of N_m as follows

$$\alpha_T(\sigma(m)) = \hat{tr}(\sigma(m)) = \theta(tr_{p'}(\sigma(\epsilon)))$$

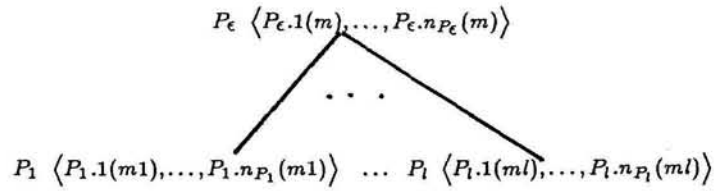

Figure 5: The subtree at position m of $A(T)$

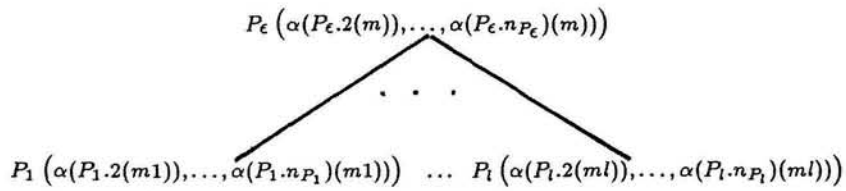

Figure 6: The subtree of $L(\langle S,\alpha\rangle)$ at position m

By virtue of lemma 4.9 α_T is valid for the lower production. In a similar fashion one proves that α_T is valid for the upper production. Clearly the same argument applies to the root of $A(T)$ which only has a lower production, and to the leaves which only have upper productions. We have proved the following:

Theorem 4.13 If T is a proof tree then $\langle A(T),\alpha_T\rangle$ is a valid attributed tree.

We now proceed to show how to obtain a proof tree from a valid attributed tree. Given a valid attributed tree $\langle S,\alpha\rangle$ a proof tree $L(\langle S,\alpha\rangle)$ is constructed as follows. If an instance of production $(p')\ \ P_\epsilon \Rightarrow P_1 \ldots P_l$ appears at position m of S as shown in figure 5 on page 25 then the subtree shown in figure 6 appears at position m of $L(\langle S,\alpha\rangle)$. The arguments of the predicates are the terms obtained by applying α to the corresponding attribute instances. We to show that this subtree corresponds to an instance of the definite clause p from which p' was obtained. Clearly α is valid for p', hence by lemma 4.11 there is a substitution θ_p such that

$$\alpha(P_i.j(mk)) = \theta_p(tr_{p'}(P_i.j(k))) = \theta_p(t_j^k)$$

This shows that the subtree of figure 6 corresponds indeed to an instance of the definite clause p as wanted. We have proven the following theorem.

Theorem 4.14 If $\langle S,\alpha\rangle$ is a valid attributed tree then $L(\langle S,\alpha\rangle)$ is a proof tree.

By starting with a proof tree T, theorem 4.13 gives a valid attributed tree $\langle A(T),\alpha_T\rangle$. By applying theorem 4.14 one obtains a proof tree $L(\langle A(T),\alpha_T\rangle)$. Clearly

$$L(\langle A(T),\alpha_T\rangle) = T$$

The mapping $T \mapsto \langle A(T),\alpha_T\rangle$ is therefore injective. One can easily see that it is also surjective. Therefore there is a bijection between the proof trees of a logic program Γ and the

valid attributed trees of its corresponding abstract attribute grammar $A(\Gamma)$. Furthermore, by construction, this bijection is computable. This is our main result, stated as follows.

Corollary 4.15 Given a Logic Program Γ, one can construct an Abstract Attribute Grammar $A(\Gamma)$ such that there is a computable one to one correspondence between the proof trees of Γ and the valid attributed trees of $A(\Gamma)$

We now revisit the example for the Logic Program to perform syntactic addition presented in the introduction.

Example 4.16 Consider the Logic Program for syntactic addition given by:

(p_1) `add(0,Y,Y).`

(p_2) `add(s(X),Y,s(Z))` $\rightarrow$ `add(X,Y,Z).`

Using our construction we come up with the following Abstract Attribute Grammar

(p_1') **add** $\Rightarrow$ **end**

$$Paths(p_1, \mathbf{0}) = \{add.1(\epsilon), \mathbf{0}\}$$
$$Paths(p_1, Y) = \{add.2(\epsilon), add.3(\epsilon)\}$$

(p_2') **add** $\Rightarrow$ **add**

$$Paths(p_2, X) = \{\pi^s(add.1(\epsilon)), add.1(1)\}$$
$$Paths(p_1, Y) = \{add.2(\epsilon), add.2(1)\}$$
$$Paths(p_1, Z) = \{\pi^s(add.3(\epsilon)), add.3(1)\}$$

This is exactly the same (modulo renaming) Abstract Attribute Grammar which we presented in the introduction and again in section 4.1.

4.5 Comparison with previous published results

In [9], Deransart Maluszynski show how to construct a Relational Attribute Grammar semantically equivalent to a Logic Program. How does our result differ? After all AAGs are special types of RAGs, hence our result does not seem to add anything new. Their transformation however, does not involve any amount of term-matching or pre-processing. They associate with each definite clause a context free production as we do. In addition, with each context free production p_j a logic formula of the form $R_j(x_1, \ldots, x_{n_j})$ is associated where $ATOC(p_j) = set x_1, \ldots, x_{n_j}$. In order to establish *semantic equivalence* between the RAG and the LP, the interpretation part of the RAG is chosen so that the predicates R_j are mapped into all n_j-tuples of terms which are instances of the terms $s_1, \ldots, s_{n_j}$ which appear (in their textual order) in the n_j different positions of the original definite clause.

Any relationship between attributes which would result from the same variables being used at different argument positions in the definite clause are absent in the syntactic part of the equivalent RAG. All agreements are pushed to the semantic level. There is no gain in terms of term-matching or possible evaluation of the LP by doing this.

However, our transformation makes explicit the agreements necessary between the different attribute occurrences at the syntactic level by making strong use of the *Path* functions. We push a semantic condition to a syntactic level, which makes the transformation more interesting.

5 Transforming Abstract Attribute Grammars into Conditional Attribute Grammars

We mentioned the lack of an algorithm to evaluate abstract attribute grammars. Since evaluation algorithms for attribute grammars have been extensively studied, it is interesting to investigate when an abstract attribute grammar can be transformed into an equivalent attribute grammar. In this section we state sufficient conditions. By applying our method, starting from a logic program one obtains a conditional attribute grammar which is similar to the one obtained by the method described in [9]. We should mention here that the difference between Attribute Grammars and Conditional Attribute Grammars is irrelevant when considering evaluation methods since those developed for Attribute Grammars also work for Conditional Attribute Grammars.

An abstract attribute grammar is said to be *reversible* if every attribute occurrence in a production can be reconstructed from its occurrences in the restriction sets . More formally:

Definition 5.1 An abstract attribute grammar $< \mathcal{A}, \mathcal{I} >$ is said to be *reversible* if for every attribute occurrence σ appearing in a production p there exists a function $f_{\sigma,p}$ over the domain of $\mathcal{I}$ which satisfies the following condition.

> Let $\{t_1, \ldots, t_k\}$ be the union of all the restriction sets in which σ appears, and let α be a locally valid valuation (i.e. valid when considering the attributed tree composed of the production p by itself). Then,
>
> $$f_{\sigma,p}(\alpha(t_1), \ldots, \alpha(t_k)) = \alpha(\sigma)$$

NOTE: the actual function $f_{\sigma,p}$ depends upon the ordering of the terms $\alpha(t_1), \ldots, \alpha(t_k)$. Also f_σ depends depends on the interpretation $\mathcal{I}$ but not on the valuation α.

Example 5.2 Every abstract attribute grammar obtained from a logic program by the construction described in section 4.4 is reversible. Consider the abstract attribute grammar for syntactic addition of example 4.16. Recall the interpretation described in section 4.3. The domain is the set of terms of a logic language, and π_1^s is the selector function for terms of the form $s(t)$. The first production is:

$$(p_1') \quad \mathbf{add} \Rightarrow \mathbf{end} \quad \begin{matrix} \{\mathbf{0}, a(\epsilon)\} \\ \{c(\epsilon), b(\epsilon)\} \end{matrix}$$

The functions $f_{a(\epsilon),p_1'}$, $f_{b(\epsilon),p_1'}$ and $f_{c(\epsilon),p_1'}$ are given by:
$f_{a(\epsilon),p_1'}(t_1, t_2) = t_2$,
$f_{b(\epsilon),p_1'}(t_1, t_2) = t_2$,
and $f_{c(\epsilon),p_1'}(t_1, t_2) = t_1$.

In this case $f_{b(\epsilon),p'_2}$, $f_{a(1),p'_2}$, $f_{b(1),p'_2}$ and $f_{c(1),p'_2}$ are similar to the first production, and $f_{a(\epsilon),p'_2}(t_1,t_2) = f_{c(\epsilon),p'_2}(t_1,t_2) = s(t_2)$ for any terms t_1, t_2.

The reversibility condition will be needed in order to reconstruct an attribute from its pieces. In order to obtain a conditional attribute grammar a splitting of attributes into inherited and synthesized is needed. For a special kind of splittings called *safe*, it will be possible to construct a conditional attribute grammar.

Since the parameter passing concept is easier captured by an *Input/Output* classification, and such a classification is implicit in the Attribute Grammar formalism, we will phrase our results in terms of *Input/Output* attributes. Recall from section 2.5 the definition of Input and Output positions of a production p of the form $X_\epsilon \Rightarrow X_1 \ldots X_n$.

$$Input(p) = \{a(\epsilon) | a \in I(X_\epsilon)\} \quad \bigcup \quad \bigcup_{i=1}^{n} \{a(i) | a \in S(X_i)\}$$

$$Output(p) = \{a(\epsilon) | a \in S(X_\epsilon)\} \quad \bigcup \quad \bigcup_{i=1}^{n} \{a(i) | a \in I(X_i)\}$$

Where $S(X)$ denotes the set of synthesized attributes of X and $I(X)$ the set of its inherited attributes. We now define *safe* splittings.

Definition 5.3 Given an Abstract Attribute Grammar $< \mathcal{A}, \mathcal{I} >$, a splitting of its attributes into inherited and synthesized, a production $(p) \quad X_\epsilon \Rightarrow X_1 \ldots X_n$ and a term t in $T_\Sigma(ATOC(p))$, we say that t is *output-free* if the i/o-splitting induced by it is such that no term in $Output(p)$ occurs in t.
A production $(p) \quad X_\epsilon \Rightarrow X_1 \ldots X_n$ is said to be *output-free* if every restriction set associated with that production has one term which is output-free.

Definition 5.4 Given an Abstract Attribute Grammar, we say that a splitting of its attributes into inherited and synthesized is *safe* if the i/o-splitting induced by it is such that every production in $\mathcal{A}$ is output-free.

Example 5.5 For the abstract attribute grammar discussed in example 5.2, consider the following splitting:

$$inherited = \{a, b\}$$
$$synthesized = \{c\}$$

The Input/Output attribute occurrences of production p'_1 are:

$$Input(p'_1) = \{a(\epsilon), b(\epsilon)\}$$
$$Output(p'_1) = \{c(\epsilon)\}.$$

The restriction sets associated with p'_1 are: $\{\mathbf{0}, a(\epsilon)\}$ and $\{b(\epsilon), c(\epsilon)\}$. Since the terms $\mathbf{0}$, $a(\epsilon)$ and $b(\epsilon)$ are output-free, so is the production p'_1.
Similarly, for production p'_2, since $a(\epsilon)$, $b(\epsilon)$ and $c(1)$ are output-free, so is the production p'_2.
This shows that for this Abstract Attribute Grammar, the splitting given above is *safe*.

Definition 5.6 An abstract attribute grammar will be called *simple* if has some safe splitting of attributes.

Theorem 5.7 Let $G =< \mathcal{A}, \mathcal{I} >$ be a reversible Abstract Attribute Grammar and let I/S be a safe splitting of the attributes of G. There exists a Conditional Attribute Grammar $G' =< \mathcal{A}', \mathcal{I}' >$ which is semantically equivalent to G (i.e. whose sets of valid decorated trees are isomorphic).

Proof: We give a constructive proof. G' is constructed by transforming each production p of G into a production p' with its associated equations. The context free components are identical as are the interpretations, except for some new function symbols are introduced in G'.

For each production p in the AAG we introduce a production p' in G'. The Context Free component is the same, the equations associated with p' are constructed given a fixed safe splitting. We set up an equation for each output position $a(j)$. Let $t_{i_1}, \ldots t_{i_{k_{a(j)}}}$ be all the terms appearing in the restriction sets which $a(j)$ appears. Since G is reversible there exists a function $f_{a(j),p}$ such that

$$\alpha(a(j)) = f_{a(j),p}(\alpha(t_{i_1}), \ldots, \alpha(t_{i_{k_{a(j)}}}))) \tag{2}$$

for any locally valid valuation α. A function symbol is added to $\mathcal{A}'$ for $f_{a(j),p}$, which we will denote by $\overline{f_{a(j),p}}$. The interpretation $\mathcal{I}'$ part of G' will interpret $\overline{f_{a(j),p}}$ by $f_{a(j),p}$.

Let $R_1, \ldots, R_{k_{a(j)}}$ be the restriction sets of which $t_{i_1}, \ldots t_{i_{k_{a(j)}}}$ are members. Since the splitting I/S of G is safe, there exist terms $h_{i_1} \in R_{i_1}, \ldots h_{i_{k_{a(j)}}} \in R_{i_{k_{a(j)}}}$ that are output-free. The following equation is associated with p' (notice that this step is non–deterministic since there might be more than one output-free term in each restriction set).

$$a(j) = \overline{f_{a(j),p}}(h_{i_1}, \ldots, h_{i_{k_{a(j)}}})$$

One also has to ensure that all the terms belonging to a restriction set are identical. In order to do so, from each term q in a restriction set, a term q' is constructed by substituting every occurrence of an output attribute in q by the term to which it is equated. In the case of $a(j)$ above, one replaces every occurrence of it by $\overline{f_{a(j),p}}(h_{i_1}, \ldots h_{i_{k_{a(j)}}})$.

For each restriction set $R_i = \left\{q^i_1, \ldots, q^i_{n_i}\right\}$ we obtain a logic formula $R'_i = (q'_{i,1} \doteq q'_{i_2}, \wedge \ldots \wedge q'_{i,n_i-1} \doteq q'_{i_{n_i}})$. (The symbol $\doteq$ stands for equality in the logic language, the dot is added to differentiate it from the equality used in equations). To avoid redundancy, expressions of the form $q' \doteq q'$ are left out. The formulae $R'_1, \ldots, R'_n$ corresponding to the restriction sets $R_1, \ldots, R_n$ are collected into the logic formula $R' = R'_1 \wedge \ldots \wedge R'_n$, which is associated with p'. Notice that R' has been constructed so that it only contains occurrences of input attributes. This completes the construction of the conditional attribute grammar G'.

We now show that G and G' have equivalent semantics. There is a natural correspondence between the attributed trees T of G and attributed trees T' of G' given by the above

transformation which relates the equations and input predicate associated with productions of G' with restriction sets of G. Notice also that α is a valuation defined for T *if and only if* it is defined for T'.

We first show that if α is a valid valuation for T, and production p occurs at address n in T, then

$$\alpha(a(nj)) = f_{a(j),p}(\alpha(h_{i_1}(n)), \ldots, \alpha(h_{i_{k_{a(j)}}}(n))) \tag{3}$$

Since α is a valid valuation for T, and any valid valuation for T is also *locally valid* for every production p that appears in T, it follows from the reversibility of G that α has to satisfy the following equality:

$$\alpha(a(nj)) = f_{a(j),p}(\alpha(t_{i_1}(n)), \ldots, \alpha(t_{i_{k_{a(j)}}}(n))) \tag{4}$$

for the terms $t_{i_1}, \ldots, t_{i_{k_{a(j)}}}$ appearing in the restriction sets associated with p. Since t_{i_l} and h_{i_l} are both members of R_{i_l} it follows from the validity of α that $\alpha(t_{i_l}(n)) = \alpha(h_{i_l}(n))$ hence equality 3 follows.

Similarly, if α is valid for T', then equality 3 follows from the fact that the equation

$$a(j) = \overline{f_{a(j),p}}(h_{i_1}, \ldots, h_{i_{k_{a(j)}}})$$

appears in T' and that $\overline{f_{a(j),p}}$ is interpreted as $f_{a(j),p}$.

We now show that α is *valid* for T *if and only if* it is valid for T' by showing that for every tree address n in T, α is valid at n *if and only if* it is valid at address n in T. Let p be the production that is used at address n and let p' be its corresponding production in G' (it follows that p' is used at address n in T'). Since the terms h_{i_l} were chosen from the same restriction set as t_{i_l}, and since α is a valid valuation , we have that $\alpha(h_{i_l}) = \alpha(t_{i_l})$. Each term q' is obtained from q by replacing each occurrence of a term of the form t_{i_l} by a term h_{i_l}. It follows that $\alpha(q'_{i,i_l}(n)) = \alpha(q_{i,i_l}(n))$ for $i = 1, \ldots, n$ and $l = 1, \ldots, n_i$. Thus a valuation α satisfies a restriction set $R_i(n) = \{q_{i,1}(n), \ldots, q_{i,n_i}(n)\}$ *iff*

$$\begin{array}{l} \alpha(q_{i,1(n)}) = \ldots = \alpha(q_{i,n_i}(n)) \text{iff} \\ \alpha(q'_{i,1(n)}) = \ldots = \alpha(q'_{i,n_i}(n)) \text{iff} \\ \text{it satisfies the logic formula } R'_i(n). \end{array}$$

We have shown that:

1. the set of attributed trees of G and G' are isomorphic,
2. given isomorphic decorated trees T of G and T' of G', and an address n in their respective domains, the set of valid valuations of T at address n is the same as the set of valid valuations of T' at address n.

It follows that G and G' have the same semantics. □

NOTE: An important point that follows from our definition of valid valuations and which is not made explicit here, is that values assigned to input positions have to be such that no term appearing in either an equation or a logic formula is undefined, i.e. they belong to the domain of the functions (or composition of functions) which are applied to them.

Example 5.8 The result of applying this construction to the addition abstract attribute grammar with the splitting described above is the following Conditional Attribute Grammar:

$$\begin{array}{ll} (p_1')\ \textbf{add} \Rightarrow \textbf{end} & c(\epsilon) = \overline{f_{c(\epsilon),p_1'}}(b(\epsilon), b(\epsilon)) \\ & (a(\epsilon) \doteq \mathbf{0} \wedge \\ & \overline{f_{c(\epsilon),p_1'}}(b(\epsilon), b(\epsilon)) = b(\epsilon)) \end{array}$$

$$\begin{array}{ll} (p_2')\ \textbf{add} \Rightarrow \textbf{add} & a(1) = \overline{f_{a(1),p_2'}}(\pi_1^s(a(\epsilon)), \pi_1^s(a(\epsilon))) \\ & b(1) = \overline{f_{b(1),p_2'}}(b(\epsilon), b(\epsilon)) \\ & c(\epsilon) = \overline{f_{c(\epsilon),p_2'}}(c(1), c(1)) \\ & (\overline{f_{a(1),p_2'}}(\pi_1^s(a(\epsilon)), \pi_1^s(a(\epsilon))) \doteq \pi_1^s(a(\epsilon)) \wedge \\ & \overline{f_{b(1),p_2'}}(b(\epsilon), b(\epsilon)) \doteq b(\epsilon) \wedge \\ & c(1) \doteq \pi_1^s(\overline{f_{c(\epsilon),p_2'}}(c(1), c(1)))) \end{array}$$

The interpretation part of the Conditional Attribute Grammar, partly obtained from example 5.2, is as follows:

$$\begin{array}{l} \mathcal{I}'(\overline{f_{c(\epsilon),p_1'}})(x,y) = x \\ \mathcal{I}'(\overline{f_{c(\epsilon),p_2'}})(x,y) = s(y) \\ \mathcal{I}'(\overline{f_{a(1),p_2'}})(x,y) = y \\ \mathcal{I}'(\overline{f_{b(1),p_2'}})(x,y) = y \\ \mathcal{I}'(\pi_1^s) = \text{the projection function on terms of the form } s(x) \end{array}$$

Given the interpretation $\mathcal{I}'$, it is easy to see that the Conditional Attribute Grammar given above is equivalent to the following notationally more intuitive Conditional Attribute Grammar.

$$\begin{array}{ll} (p_1'')\ \textbf{add} \Rightarrow \textbf{end} & c(\epsilon) = b(\epsilon) \\ & (a(\epsilon) \doteq \mathbf{0}) \end{array}$$

$$\begin{array}{ll} (p_2'')\ \textbf{add} \Rightarrow \textbf{add} & a(1) = \pi_1^s(a(\epsilon)) \\ & b(1) = b(\epsilon) \\ & c(\epsilon) = s(c(1)) \end{array}$$

This last Conditional Attribute Grammar is similar to the one obtained from the logic program for addition by the method described in [9] except that instead of $a(\epsilon) \doteq \mathbf{0}$ they introduce the predicate $instance(a(\epsilon), \mathbf{0})$; also a predicate $instance(a(\epsilon), s(X))$ is introduced in the second production by them which is not needed here because of the way valid valuations are defined, namely no term can be undefined. This forces $a(\epsilon)$ to be an instance of the term $s(X)$.

From theorem 5.7 we obtain the following corollary.

Corollary 5.9 For every reversible simple Abstract Attribute Grammar one can construct an equivalent Conditional Attribute Grammar

6 Transforming a Logic Program into a Conditional Attribute Grammar

We will now show how to transform a Logic Program with a *legal* Input/Output assignment to its predicate positions into a Conditional Attribute Grammar. (The exact meaning of legal will be stated below.) This is done in two steps. First the Logic Program is transformed into an Abstract Attribute Grammar as described in section 4. Them the transformation of the section 5 is applied to obtain an equivalent Conditional Attribute Grammar. In order to utilize the construction given in theorem 5.7 we have to make sure that the Abstract Attribute Grammar obtained from the Logic Program is indeed reversible. That is the focus of the next lemma.

Lemma 6.1 The Abstract Attribute Grammar obtained from a Logic Program using the construction described in section 4.4 is reversible.

Proof: Recall that the construction associated predicate symbols with non-terminals and predicate argument positions with attributes. The natural mapping between attributes and the terms appearing in the corresponding argument position of the clause was denoted by $tr_{p'}$, i.e. if σ is an attribute occurrence in production p', then $tr_{p'}(\sigma)$ is the term appearing in the original clause of the LP in the argument position corresponding to σ (refer to page 24).

In order to show reversibility of the Abstract Attribute Grammar we pick an arbitrary production p' (which corresponds to a definite clause p) and an arbitrary attribute occurrence σ of p' and we construct a function $f_{\sigma,p'}$ such that for any locally valid valuation α,

$$\alpha(\sigma) = f_{\sigma,p'}(\alpha(t_1), \ldots, \alpha(t_k)) \tag{5}$$

where $t_1, \ldots, t_k$ are all the terms in the restriction sets of p' in which σ appears. Intuitively, this is possible because each restriction set associated with p' represents an atom occurring in the original definite clause. Thus the atoms that make up the term corresponding to σ are scattered among the restriction sets. Since we can construct a term from the atoms appearing in it, we can reconstruct it from selected elements of the restriction sets.

Let us denote by t the term $tr_{p'}(\sigma)$. Let $s_1, \ldots, s_n$ be all the atoms (i.e. variables and constants) appearing in t. Clearly, t can be constructed from its atoms by using term constructors, hence

$$t = \phi(s_1, \ldots, s_n) \tag{6}$$

for a function ϕ that is a composition of term constructors. For example, the term $t = h(f(X, Y), \mathbf{a})$ can be constructed from X, Y and $\mathbf{a}$ by the function

$$\phi :< x_1, \ldots, x_3 > \mapsto h(f(x_1, x_2), x_3).$$

Clearly, for any substitution θ, $\theta(t) = \phi(\theta(s_1), \ldots, \theta(s_n))$. Since α is a locally valid valuation for p', we have by corollary 4.12 on page 24 that $\alpha(\sigma) = \theta_{p,\alpha}(t)$. Hence,

$$\alpha(\sigma) = \phi(\theta_{p,\alpha}(s_1), \ldots, \theta_{p,\alpha}(s_n)) \tag{7}$$

Recall that each restriction set R_i associated with p' represents all the paths leading to an atom in p, i.e. $R_i = Paths(p, s_i)$ for some s_i. Let $s'_i \in Paths(p, s_i)$, hence s'_i is of the form $\tau(\sigma_i)$ for some attribute occurrence σ_i of p'. Hence

$$\alpha(s'_i) = \overline{\tau}(\alpha(\sigma_i)) \tag{8}$$

By corollary 4.12,

$$\alpha(\sigma_i) = \theta_{p,\alpha}(tr_p(\sigma_i)) \tag{9}$$

hence,

$$\alpha(s'_i) = \overline{\tau}(\theta_{p,\alpha}(tr_p(\sigma_i))) \tag{10}$$

Since $\tau(\sigma) \in Paths(p, s_i)$, by lemma 4.9 on page 22,

$$\theta_{p,\alpha}(s_i) = \overline{\tau}(\theta_{p,\alpha}(tr_p(\sigma_i))) \tag{11}$$

Hence, from equations 7, 10 and 11 we obtain the following equality

$$\alpha(\sigma) = \phi(\alpha(s'_1), \ldots, \alpha(s'_n)) \tag{12}$$

for $s'_1, \ldots, s'_n$ members of the restriction sets of p' in which σ appears. Since α is valid, it has to agree on all members of each restriction set thus for for any terms $t_1, \ldots, t_n$ such that t_i and s'_i are in the same restriction set we have that

$$\alpha(\sigma) = \phi(\alpha(t_1), \ldots, \alpha(t_n)) \tag{13}$$

If we now denote ϕ by $f_{\sigma,p'}$, we have equality 5 as wanted. □

Now, in order to obtain a Conditional Attribute Grammar from an Abstract Attribute Grammar, theorem 5.7 requires the AAG to be not only reversible but also *safe*. Therefore, in order to transform a Logic Program into a Conditional Attribute Grammar we will also need to impose certain conditions on the LP that will result in a safe splitting of the attributes of the AAG. We borrow our notation from [9].

Definition 6.2 Given a Logic Program, a *direction assignment* (*d-assig*) is a mapping of the arguments of each predicate symbol occurring in the LP into the set $\{\downarrow, \uparrow\}$.

A *d-assig* splits the argument positions of each definite clause into *input positions* and *output positions* similarly to the way a splitting of attributes into *inherited* and *synthesized* results in an *Input/Output* splitting of the attribute occurrences of a production. Given a definite clause, its *input positions* are $\downarrow$ positions of the head of the clause and $\uparrow$ positions of its body. *Output positions* are $\uparrow$ positions of its head and $\downarrow$ positions of its body.

The condition needed to obtain an Attribute Grammar is that each output position be computable from input positions. This can be done if each variable which occurs in some term in an output position also occurs in a term which is in an input position. This will guarantee that the value of the variable be instantiated when it is needed.

Definition 6.3 A *d-assig* is called *safe* if each variable occurs in at least one input position.

Since there is a one-to-one correspondence between argument positions of predicates and attributes, a *d-assig* imposes a splitting of the attributes: attributes corresponding to argument positions to which the *d-assig* gives the value $\downarrow$ are *inherited*, while attributes corresponding to argument positions that receive the value $\uparrow$ are *synthesized*. It is easy to see that *input* argument positions of a definite clause p correspond to *input* attribute occurrences of the corresponding production p' and similarly for the *output* positions and attribute occurrences. Thus, it makes sense to ask the question of the *safeness*, in Abstract Attribute Grammar terms, of the *d-assig*. We have the following lemma.

Lemma 6.4 Let Γ be a Logic Program, let G be its equivalent Abstract Attribute Grammar obtained using our construction. Also, let d be a *d-assig* for Γ and let I/O be its induced splitting on the attributes of G. If d is *safe* so is I/O.

Proof: Let p be a definite clause appearing in Γ and let p' be its corresponding production in G. We have to show that each restriction set associated with p' has an *output-free* element.

Recall that each restriction set R is of the form $Paths(p, s)$ for some atom s. If s is a constant, then $s \in Paths(p, s)$ by definition of $Paths(p, s)$, hence R is output-free. If s is a variable X, by the safeness of Γ, there exists an input position of p where X appears. Let σ be the attribute corresponding to that input position. By definition, σ is an *input* attribute occurrence of p'. Since $R = Paths(p, X)$ and X appears in $tr(\sigma)$, there has to be a term of the form $\tau(\sigma)$ in R. Since σ is an input attribute occurrence, $\tau(\sigma)$ is an output-free term, hence R has an output-free element. Since R was chosen arbitrarily, this completes our proof. □

By putting together lemmas 6.1 and 6.4 with theorem 5.7 we obtain the following corollary.

Corollary 6.5 Let Γ be a Logic Program and let d be a safe *d-assig* for Γ, there exists a Conditional Attribute Grammar which is semantically equivalent to Γ.

Proof: First, construct an Abstract Attribute Grammar G which is equivalent to Γ by using the construction of section 4.4. Lemma 6.1 tells us that G is reversible. Lemma 6.4 shows that the induced I/O splitting is safe. Thus, the conditions for theorem 5.7 are met allowing us to construct a Conditional Attribute Grammar G' which is semantically equivalent to G, hence to Γ. □

Definition 6.6 A Logic Program for wich there exists a safe *d-assig* will be called a *simple* Logic Program.

From our previous results, the following corollary follows.

Corollary 6.7 Every simple logic program has an equivalent conditional attribute grammar.

In general, the problem we are trying to tackle is that of finding an answer substitution for a query posed to a Logic Program. A query that induces, by virtue of its ground terms, a safe splitting of the argument positions, will be called a *safe query.* Given a Logic Program Γ and a safe query, an equivalent Conditional Attribute Grammar can be constructed on the fly to assist in its *unification-free* evaluation (how exactly this evaluation proceeds is beyond the scope of this paper). The process will become more efficient if the amount of processing to be done for each query could be reduced.

Notice that the reversing functions $f_{\sigma,p'}$ used in lemma 6.1 do not depend upon a specific valuation nor upon a specific splitting of the attributes. Thus, these functions can all be precomputed for all attribute occurrences and productions at once when first transforming a Logic Program into an Abstract Attribute Grammar. Then, in order to construct a Conditional Attribute Grammar in response to a *safe* query, it will be enough to determine the *input* and *output* positions of each production and then assemble the necessary equations and predicates by using the reversing functions precomputed when constructing the Abstract Attribute Grammar.

We now compare our results with the transformation from logic programs to conditional attribute grammars described in [9]. Their *Construction 3* shows how to obtain a Conditional Attribute Grammar equivalent to a Logic Program with respect to a safe *d-assig. Proposition 1* states that every simple Logic Program can be transformed into an equivalent Conditional Attribute Grammar. This is similar to our corollary 6.7.

Given Logic Program with a safe *d-assig* our transformation will exhibit an equivalent Conditional Attribute Grammar by performing an intermediate transformation into a Abstract Attribute Grammar. Thus we have two steps in our transformation: from a Logic Program to an Abstract Attribute Grammar and from an Abstract Attribute Grammar into a Conditional Attribute Grammar. What is the gain?

The method described in [9] requires a specific input/output assignment to transform a logic program into a conditional attribute grammar. With our method however, one can construct a generic abstract attribute grammar for a given logic program without dealing with input/output assignments. When a specific *input/output* behavior is imposed, its safeness can be checked and the corresponding conditional attribute grammar constructed from the abstract attribute grammar. Furthermore, as we discussed above, the amount of processing needed to produce a Conditional Attribute Grammar from a specific query can be substantially lowered by precomputing the reversing functions.

The advantage of our method when compared to theirs is that a significant portion of the translation can be done independently of the specific input/output assignment, which means that portions of the *compilation* process can be done for the entire logic program independently of any query. It is the part that generates an abstract attribute grammar equivalent to the original logic program. Whenever a specific query is to be evaluated, an input/output assignment is imposed. Our method then proceeds to check for safeness of the assignment and then to the construction of the equivalent conditional attribute grammar whenever possible. In [9] a complete translation has to be performed each time query is posed. Our method therefore achieves a greater level of efficiency by identifying aspects of the logic program which are independent of the particular *i/o* assignment.

7 Conclusion

In this paper we have investigated the relationship between Logic Programs and Attribute Grammars. The lack of an input/output behavior in the former clashes with the nature of the latter. This led us to introduce Abstract Attribute Grammars. We have shown the close relationship between both formalisms, and we have provided a construction that transforms *any* logic program into an equivalent abstract attribute grammar.

We have given sufficient conditions for transforming an Abstract Attribute Grammar into an equivalent Conditional Attribute Grammar. These conditions apply in the cases of Abstract Attribute Grammars obtained from Logic Programs via the construction described in section 4.4.

We have also shown how our work ties in with the work by Deransart and Maluszynski [9]. When constrained to the domain of their investigations (*simple* logic programs), the end results of applying the transformations described there and here do coincide. The types of constructions described here and in [9] can be viewed as an attempt to exclude unification from the computation of proof-trees, by replacing it by a form of pre-processed matching. Our approach is more powerful in a number of ways:

1. The transformation from Logic Programs into Abstract Attribute Grammars is **not** restricted to a particular class of LPs.

2. Although, Deransart and Maluszynski are able to transform arbitrary Logic Programs into equivalent Relational Attribute Grammars, their transformation does not provide any degree of term matching. Our transformation from Logic Programs to Abstract Attribute Grammars does.

3. There is a certain amount of pre-processed matching that can be done *independently* of a particular query being posed to a Logic Program. Our transformation from Logic Programs to Abstract Attribute Grammars captures that. In [9], all meaningful transformations are dependent upon a specific i/o assignment, hence to a specific class of queries.

4. By performing a greater amount of preprocessing, which involves pre-computing the *reversing functions* at the time a Logic Program is transformed into an Abstract Attribute Grammar, we show how to construct of a Conditional Attribute Grammar equivalent to the Logic Program with respect to a specific safe query without performiong any amount of term matching. This sets the ground for a scheme to do Unification-Free evaluation of Logic Programs.

In this paper we have presented two transformations: LPs to AAGs and AAGs to LPs. We have formally shown the correctness of these transformations and explained what their advantage is. We have compared our results to those published elsewhere and we have argued for the generality and rigurosity of our approach.

8 Further Research

We are also interested in continuing our investigations on the following topics:

- *Evaluation*: how to use attribute evaluators to run Logic Programs that have been transformed into Abstract Attribute Grammars?
  One can look at the Abstract Attribute Grammar as a *compiled* form of the logic program. The exact way in which this intermediate code can be used is not clear at the moment. One way of tacking this problem is to use logic programming interpreter techniques although it is not clear how efficient and practical this would be. Another possibility is to adapt Attribute Grammar techniques to AAG.

- We have given *sufficient* conditions for transforming an Abstract Attribute Grammar into an equivalent Conditional Attribute Grammar, the topic of finding *necessary* conditions is worth studying.

- *Functionality*: by un-freezing the interpretation of functions in a Logic Program and by using Conditional Attribute Grammar evaluation techniques, it seems plausible to add *functional programming* capabilities to logic programs.

- *Natural Language Processing*: It is possible to transform some DCGs into AGs. In doing so one can separate the parsing process from the rest of the computation which can be dealt by an attribute evaluator. This leads to efficient implementations of Natural Language Processing systems whose prototypes are built in Prolog and then transformed into AGs. Characterizing the class of DCGs for which this will work is worth studying.

It is our thesis that the preprocessing of clauses of a Logic Program proposed here will substantial improve the run time for a large class of LPs.

The author is very greatful to Jean Gallier for his stimulating discussions, to Pierre Deransart for his comments on earlier versions of this paper and to the referees of this paper for their constructive criticism.

References

[1] K.R. Apt and M.H. Van Emden. Contributions to the theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, 1982.

[2] I. Attali and P. Fracnchi-Zannettacci. Unification Free Execution of TYPOL programs by Semantic Attribute Evaluation. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 160–177, 1988.

[3] L. Chirica and D. Martin. An Order–Algebraic definition of Knuthian Semantics. *Mathematical Systems Theory*, (13):1–27, 1979.

[4] K. L. Clark. Predicate Logic as a Computational Formalism. Research Monograph 79/59, Imperial College, London, 1979.

[5] W. F. Clocksin and C.S. Melish. *Programming in Prolog.* Springer-Verlag, 1984.

[6] B. Courcelle. Attribute Grammars: Theory and Applications. In *Lecture Notes in Computer Science*, pages 75–95. Springer–Verlag, 1981.

[7] B. Courcelle and P. Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes. *Theoretical Computer Science*, 17(2):235–258, 1982.

[8] Bruno Courcelle and Pierre Deransart. Proofs of Partial Correctness for Attribute Grammars with Applications to Recursive Procedures and Logic Programming. *Information and Computation*, 78(1):1–55, 1988.

[9] P. Deransart and J. Maluszynski. Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming*, 1(2):119–225, 1985.

[10] Jean H. Gallier. An efficient evaluator for Attribute Grammars with Conditional Rules. Technical report, Computer and Information Sciences Department, University of Pennsylvania, Philadelphia, PA, 1985.

[11] Jean H. Gallier. *Logic for Computer Science.* Harper and Row, 1985.

[12] S. Gorn. Explicit Definitions and Linguistic Dominoes. In John Hart and Satoru Tazkasu, editors, *Systems and Computer Science.* Hedonist Press, 1965.

[13] Tomás Isakowitz. On the Relationship between Logic Programs and Attribute Grammars. Master's thesis, CIS Department, University of Pennsylvania, Philadelphia, PA 19104, December 1985.

[14] D. Knuth. Semantics of Context Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[15] R. A. Kowalski. Predicate Logic as a Programming Language. In J. Rosenfeld, editor, *Information Processing 74*, pages 556–574. North–Holland, 1974.

[16] B.M. Mayoh. Attribute Grammars and Mathematical Systems. *SIAM on Computing*, 3(10):503–518, 1981.