Using logical formulas to improve the expressiveness of Relational Attribute Grammars

1Kablan Barbar, 2Anthony Beurivé, 3May Dehayni
1Department of Applied Mathematics, Faculty of Sciences 2, Lebanese University, Fanar, Lebanon
2LaBRI, University of Bordeaux 1, France
3Department of Applied Mathematics, Faculty of Sciences 1, Lebanese University, Hadath, Lebanon
1 kbarbar@ul.edu.lb, 2 a.beurive@gmail.com, 3 maydehayni@ul.edu.lb

ABSTRACT

Considering the theory of attribute grammars, we use logical formulas instead of traditional functional semantic rules. Following the decoration of a derivation tree, a suitable algorithm should maintain the consistency of the formulas together with the evaluation of the attributes. This may be a Prolog-like resolution, but this paper examines a somewhat different strategy, based on production specialization, local consistency and propagation: given a derivation tree, it is interactively decorated, i.e. incrementally checked and evaluated. The non-directed dependencies are dynamically directed during attribute evaluation.

Keywords: Input/Output attribute grammars, local consistency, logical programming, propagation, relational attribute grammars.

1. INTRODUCTION

Attribute grammars (abbreviated AGs) were first introduced by Knuth to describe syntactic-based translations [1]. This approach is a purely declarative programming paradigm syntax directed. They have been widely studied and used, especially as a compilation technique in the field of programming languages by Aho et al. [2], or as formal specifications for more general tree transductions and semantics by Alblas et al. [3], Deransart et al. [4], Deransart and Jourdan [5], and Engelfriet [6].

An AG is a context-free grammar (CFG) which non-terminal symbols are decorated with inherited and synthesized attributes, and productions are enriched with semantic rules defining assignments for the attributes. The goal is to give some “meaning” to the terms obtained from the grammar. The semantic rules show dependencies between attributes, revealing the order to compute their values. In short, some attributes should be computed before other ones because the formers are parameters of the latters.

When using AG, one common task is to avoid circular dependencies for any derivation tree of the grammar. Knuth [1] presented an exponential-space algorithm for the circularity problem. The intrinsically exponential complexity of this problem was first proved by Jazayeri et al. [7], who reduced the acceptance problem of writing pushdown acceptors to the circularity problem. Jazayeri [8] (and the correction by Dill [9]) tried to provide a simpler construction of AGs by reducing the acceptance problem of space-bounded alternating Turing machines.

Another task is to determine efficient methods to compute all the attributes in a given derivation tree. In the classical way, the AG is statically analyzed to anticipate the whole dependencies of any derivation tree. The attributes in the semantic rules are defined as input (or output) only in order to find convenient properties of the dependencies by Courcelle [10]. Yet, these restrictions reduce the expressiveness of AGs.

Classical AGs lack of expressiveness has resulted in limited use outside the domain of static language processing. This leads Parigot et al. to extend the classical formalism into the notion of Dynamic Attribute Grammars (DAG) to enhance the expressiveness and to allow describing computations on structures that are not just trees [11]. This results in a language that is comparable in power to most functional languages, with a distinctive declarative character. Kikuchi and Katayama define the semantics of general AGs by using semantic functions whose inputs are structures derived from the underlying grammar and whose outputs are attributed structures [12]. Then they provide classifications of general AGs based on the abstract properties of semantic functions. In [13], an extension of AGs that works over extended CFG is introduced; it allows arbitrary regular expressions on the right-hand side of productions. Viewed as a query language, extended AGs are particularly relevant as they can take into account the inherent order of the children of a node in a structured document.

In this paper we will rather emphasize the fact that using non-directed semantic rules (i.e. relations or constraints) greatly enhances the declarative power of AGs, because it is no more presumed which attribute should be evaluated first. At “execution time” dependencies are dynamically built when some attribute is declared as input attribute. In section3, we recall basic formal definitions of concepts like CFG and productions. In Section4, we adapt the definition of Relational AGs in [10] to fit our needs. Section5 reveals input/output productions as our main tool to describe static and dynamic evaluation. Such productions are connected together in Section6 to build partially evaluated derivation trees. They are compared together in Section7 to show possible transformations from one into another. Then, in Section8 we incrementally evaluate the attributes of a
derivation tree using step by step transformations of the production occurrences.

2. RELATED WORKS

Courcelle et al. introduce the Relational AG as a formal tool to define such non-directed relational semantic rules separately from an abstract CFG [10]. It does not give any operational clue to satisfy the relations in a decorated tree. Yet, if an attribute in a production occurrence is declared being an input attribute, some other attributes that depend on may be evaluated. As any attribute can become an input attribute in a production occurrence, the search for dependencies is parameterized with the set of current input attributes.

This study is related to previous works that brings together AGs and logic [14, 15] or AGs and constraint satisfaction [16, 17, 18]. In Maluszynski [19], a unified view of AGs and logic programs is presented. The author compares both formalisms and shows that AGs have some features that are not present in logic programs. He proposes some extensions in the field of logic programming in order to enrich logic programs with extra features. Batory [20] describes the interpretation of grammar representations in terms of propositional logic formulae. Isakowitz introduces Abstract Attribute Grammars (AAG) in order to study the transformation from Logic Programs into AAG and vice versa [21]. He provides a construction that transforms any logic program into an equivalent AAG. The motivation for much of this work comes from the need for verifying the correctness of feature model selections that represent individual products. Ruffolo and Manna define semantic models in order to exploit domain knowledge for managing both structured and unstructured information [22]. These semantic models are executable, flexible and agile representation of domain knowledge. They are expressed by means of the Codex Language obtained combining Disjunctive Logic Programming and AGs.

Dynamic and incremental use of AGs is also concerned in [11, 23, 24]. Reps, Teitelbaum, and Demers develop the Cornell Synthesizer Generator that is an incremental evaluator generation tool [25]. It offers the possibility of replacing a sub-tree of a syntactic tree by another sub-tree: the propagation of new attribute values on the whole tree is then automatically processed. By combining dynamic and incremental aspects, attribute dependencies can be dynamically directed, and that waiting for the values of all the input attributes is not necessary in order to evaluate some parts of the decorated tree. Unlike Prolog, enumeration of all the solutions is not considered here because there is usually infinitely many, unless some extra information. On the contrary, partial solutions are managed. Thus, each time an attribute gets a value in a production occurrence, the current set of input attributes is enriched, potentially leading to new dependencies and incremental propagations in the derivation tree.

3. ABSTRACT CONTEXT-FREE GRAMMARS

In this section, we recall basic formal definitions of concepts that will be used further on, like CFG, productions and derivation trees.

Definition1 (CFG) A context-free grammar is a tuple \((N, T, Z, P)\) where:
- \(N\) is the alphabet of non-terminal symbols;
- \(T\) is the alphabet of terminal symbols; \(N \cap T = \emptyset\)
- \(Z\) is the axiom of the grammar, \(Z \in N\); \(Z\) must be the root of any derivation tree of the grammar;
- \(P\) is a set of context-free productions (see Definition2).

Definition2 (CFP) A context-free production \(p\) in a CFG \((N, T, Z, P)\) is a tuple \(X_0 \rightarrow X_1 ... X_n\) where:
- \(X_0\) is an occurrence of an element of \(N\);
- \(X_1 \ldots X_n\) are occurrences of elements of \(N \cup T\).

In the production \(p\): \(X_0\) is the left-hand side of \(p\), while \(X_1 \ldots X_n\) is the right-hand side. The elements in a production \(p\), even if they are occurrences of identical terminal or non-terminal symbols, are characterized by their positions in \(p\), in Dewey notation (the symbol 0 denotes the empty word \(\varepsilon\)). Thus, there is a straightforward tree representation of a production, where the root is \(X_0\) and leaves are \(X_1 \ldots X_n\).

Definition3 (ACFG) An abstract CFG is a tuple \((N, P)\) where \(N\) and \(P\) appear in a CFG tuple \((N, T, Z, P)\). If a production \(p\) in \(P\) appears in a CFG tuple \((N, T, Z, P)\), it can be replaced by a sub-tree of a syntactic tree.

ACFG are the essence of CFG. We can also note that the axiom \(Z\) disappears: any non-terminal in the left-hand side of a production can be the root of a derivation tree of the grammar.

Example1 shows an ACFG that will be used with attributes in section 3 to illustrate the factorial function. Example2 shows an ACFG for strictly binary trees.

Example1 (FacACFG) An abstract CFG representing a free monoid:
\[
N = \{\text{Fac}\} \\
P = \{p_1 : \text{Fac} \rightarrow \text{Fac}, p_2 : \text{Fac} \rightarrow \varepsilon\}
\]

Example2 (BinACFG) An abstract CFG for strictly binary trees:
\[
N = \{\text{Bin}\} \\
P = \{p_1 : \text{Bin} \rightarrow \text{Bin Bin}, p_2 : \text{Bin} \rightarrow \varepsilon\}
\]

Let \(G\) be an ACFG. A production \(X_0 \rightarrow X_1 ... X_n\) in \(G\) gives rise to a set of trees which roots represent occurrences of \(X_0\) and direct sub trees derived from productions with \(X_1 \ldots X_n\) as left-hand sides. In a derivation tree, the occurrences of (the roots of) the productions and the
occurrences of the non-terminals are characterized by their positions in Dewey notation, obtained by concatenating their position in the production to the global position of their parent in the tree.

![Abstract Derivation Trees](image)

Fig.1 shows two derivation trees for FacACFG and BinACFG. The Fac occurrences of the production $p_1$ appear at positions 0, 1 and an occurrence of $p_2$ is at position 1.1; while the Bin occurrences of the production $p_1$ appear at positions 0, 1, 2 and 2.1.

The way a derivation tree is constructed is an important point for us, as it partly determines why and how the attributes are computed. Look again at the trees in Fig.1, and imagine that they are interactively built. Each time, we have to choose which non-terminal to develop, using the productions of the grammar as construction rules. Clearly, different strategies exist, whether bottom-up, top-down, or in either direction, to finally obtain a derivation tree. In the next sections, we show how these strategies determine orientations between attributes. But in most cases, we will consider a tree that already exists before any orientations or evaluations.

4. RELATIONAL ATTRIBUTE GRAMMARS

In this section, we recall the definition of relational attribute grammars (RAG). Definitions of “classical” AGs can be found in [26] and [10], but we do not use them here. It is probably possible to transform AGs into RAGs, e.g. by considering directed semantic rules as non-directed, thus breaking the distinction between inherited (evaluated during a top-down tree traversal) and synthesized attributes (evaluated during a bottom-up traversal) in the specification of the grammar.

Deransart and Courcelle introduced the RAGs to prove the validity of a “classical” AG with respect to a specification. A specification consists in a collection of logical formulas, each formula being associated to a production and establishing the relationships between the attributes of this production.

From a different point of view, logical formulas can replace the semantic rules to directly specify which relations the attributes of a production should respect. In the sequel, we adopt this point of view. Such formulas enhance the declarative power of classical AGs, while it becomes more difficult to prove their correctness and to compute attributes, because of the use of a possibly too general logical language.

We use slightly different notations than in [27] and [4]. We don’t define the sorts of the attributes, nor give precise definition for the logical parts of the grammars. These restrictions are not significant for the purpose of this paper because we just want to outline a specification level independent from the underlying logical language.

**Definition 4 (RAG)** A Relational Attribute Grammar is a tuple $(N, P, A, \varphi, I)$ where:

- $(N, P)$ is an ACFG;
- $A$ is the alphabet of attribute names;
- $\varphi$ is a finite set of formulas from a logical language $L$;
- $I$ is the interpretation of $L$.

Each element $X$ in $N$ is decorated with a subset of $A$, denoted $A_X$. An attribute symbol appearing in $A_X$ is called the occurrence of the attribute $a$ in $X$, or the attribute $a$ of $X$. For a production $p$ in $P$, the attributes $a, b, \ldots$ of a non-terminal $X$ appearing at position $i$ are denoted $a_i, b_i, \ldots$ so they form a set denoted $A_{X_i}$. The set $A_{X}$ of the attributes appearing in $p$ is equal to $\bigcup_{X \in P} A_{X}$. Each production in $P$ is associated to a unique formula in $\varphi$ as explained in Definition 5. $N$ and $P$ form the grammatical part (syntax) of the RAG, while $A, \varphi,$ and $I$ form the labeling formalism part (semantics).

**Definition 5 (RAP)** A relational Attribute Production in a RAG $(N, P, A, \varphi, I)$ is a tuple $(p, \varphi_p)$ where:

- $p$ is a context-free production from $P$;
- $\varphi_p$ is the formula of $\varphi$ associated to $p$.

The set of variables in $\varphi_p$ should be a subset of $A_p$. If not specified, these variables are existentially quantified. For each context-free production $p$ in $P$, there is a unique RAP $(p, \varphi_p)$ (thus, we may ambiguously use $p$ to denote the context-free production $p$ or the RAP $(p, \varphi_p)$).

Example 3 shows a RAG for the factorial function. The attributes $n$ and $r$ of the non-terminal Fac respectively denote the argument of the recursive call and the corresponding result $n!$.

**Example 3 (FacRAG)** A RAG for the factorial function:

$N = \{\text{Fac}\}$

$P = \{p_1: \text{Fac} \rightarrow \text{Fac}, p_2: \text{Fac} \rightarrow \varepsilon\}$

$A_{\text{Fac}} = \{n, r\}$

$A_{p_1} = \{n_0, r_0, n_1, r_1\}$

$A_{p_2} = \{n_0, r_0\}$

$\varphi_{p_1} = (n_0 = n_1 + 1 \land r_0 = r_1 \times n_0)$

$\varphi_{p_2} = (n_0 = 0 \land r_0 = 1)$

$1 : L \rightarrow \text{Bool} \cup \text{Nat}$; Nat denotes natural numbers with
the usual operations.

A more general function is shown in Example 4. The formulas associated to the productions specify the relations between the occurrences of the attributes. In these formulas, the symbol \( \sim \) denotes an equivalence predicate, not an assignment function, so it is not explicitly specified how the attributes should be computed. It strongly depends on the way a derivation tree is constructed, and on extra information like the values of some unknown attributes.

**Example 4 (ParamFac\(_{\text{RAG}}\))** A RAG for a kind of factorial function with unspecified start conditions:

\[
\begin{align*}
N &= \{\text{ParamFac}\} \\
P &= \{p_1: \text{ParamFac} \to \text{ParamFac}, p_2: \text{ParamFac} \to \epsilon\} \\
A_{p_1} &= \{n_0, r_0, n_1, r_1\} \\
A_{p_2} &= \{n_0, r_0\} \\
\varphi_{p_1} &= (n_0 = n_1 + 1 \land r_0 = r_1 \times n_0) \\
\varphi_{p_2} &= \text{true, the formula is always true} \\
I &= L \to \text{Bool} \cup \text{Nat} \geq 0; \text{Nat} \geq 0 \text{ is the domain of positive natural numbers with the usual operations.}
\end{align*}
\]

Let G be a RAG. The set of relational derivation trees determined by G is the same as the set of derivation trees determined by the ACFG in G, the occurrences of the non-terminals being decorated with additional vertices representing their attributes. An attribute in the tree has the same position as its related non-terminal occurrence. The relations between the attributes are represented by edges or hyper edges. Consequently, for a derivation tree \( t \), the attributes (vertices) together with the relations (edges) form a graph called the attribute graph of \( t \) and denoted \( \text{Graph}(t) \). In Section 2, some strategies to generate derivation trees were outlined. The attribute graph of a form a graph called the attribute graph of non-terminals being decorated with additional vertices.

Fig. 2, a top-down strategy would wait an occurrence of \( p_2 \) before evaluating, while a bottom-up strategy, starting at the occurrence of \( p_2 \), could evaluate the attributes during the construction of the tree. Each strategy would consider the values 0 and 1 in \( \varphi_{p_2} \) as input. In the tree at the right on Fig. 2, the situation is clearly different: some extra information is needed to compute even a part of the attributes; else there are infinitely many solutions.

**Definition 6 (I/O RAP)** An Input/Output RAP is a tuple \((p, \text{In}, \text{Out}, v')\):

- \( p \) is a relational production from a relational grammar \((P, A, P, I)\);
- \( \text{In} \) is a subset of \( A_p \), called the input set, which elements are called the input assignment attributes;
- \( v \) is a mapping from \( \text{In} \) to the domain of interpretation, called the input assignment;
- \( \text{Out} \) is a subset of \( A_p \) - \( \text{In} \), called the output set, which elements are called the output attributes;
- \( v' \) is a mapping from \( \text{Out} \) to a domain of functional terms which variables are the attributes of \( \text{In} \). These terms have to be interpreted with \( I \) and \( v \); \( v' \) is called the output assignment.

Let \( a \) an output attribute. We have \( v'(a) = f(a_1, \ldots, a_k) \), with \( f \) a functional term and \( a_1, \ldots, a_k \) some input attributes (the indexes do not correspond to positions in \( p \)). The output assignment \( v' \) should be a “logical consequence” of \( \varphi_p \) with the assignment \( v \) in the interpretation \( I \), so we write: \( (I, v) \models (\varphi_p \land a = f(a_1, \ldots, a_k)) \).

The most important problem, not addressed here, is to automatically determine proper functional terms \( f \) in \( v' \), just by considering \( \varphi_p \) and \( v \), i.e. information relative to the production, and not from a global knowledge about the
grammar. These terms should precisely reflect functional properties of $\varphi_p$.

The set $A_p = In - Out$ is the set of attributes which are neither input nor output attributes. They are called the unknown attributes. It is possible to tell properties of these attributes just by looking at $\varphi_p$ and $v$. One of these properties is that the set of their possible values is not reduced to a singleton, so they don’t appear in $Out$.

Because we give no restrictions on the formulas in $\varphi$, we have to distinguish the values of the input attributes: in general, different input values for the same attribute may induce different output sets, e.g. the formula $((a=0 \land b=1) \lor (a=1 \land e=a-1))$ gives different output sets for different values of $a$.

Eventually, $\varphi_p$ can be invalidated by $v$. This situation is called a conflict. Depending on the logical language and its interpretation, verifying the validity of a formula under a particular assignment may be “hard” or, worse, undecidable, but it is not the subject of this paper. An assignment $v$ is said valid if it does not invalidate $\varphi_p$.

In the sequel, we will only consider productions for which there is no conflict, i.e. we consider only valid assignments. This restriction will be relaxed in future papers.

**Example 5** Two I/O productions obtained from a production $p_1$:

$p_1 = \langle \text{ParamFac} \rightarrow \text{ParamFac}, n_0 = n_1 + 1 \land r_0 = r_1 \times n_0 \rangle$

$p_1' = \langle \text{ParamFac}, n_1, r_1, \text{Out}, \text{In}, v_1 \rangle$

$v_1(n_0) = 3$

$v_1(r_1) = 8$

$v_1(r_0) = r_1 \times n_0$

As explained in section 2, given two productions $p_1$ and $p_2$ of an ACFG, an occurrence of $p_1$ can be connected to an occurrence of $p_2$ if the left-hand side of $p_1$ appears in the right-hand side of $p_2$, or inversely, if the left-hand side of $p_2$ appears in the right-hand side of $p_1$. The same principle applies for RAPs and their attribute graphs.

Yet, considering I/O productions, we need to define more precisely how they can be connected together in terms of assignments of attributes. Doing this we straight define the derivations trees of a grammar containing I/O productions.

**Definition 7 (IOAG)** An Input/Output Attribute Grammar is a tuple $(G, P')$ where:

- $G$ is a RAG $(N, P, A, \varphi, I)$
- $P'$ is the set of I/O productions obtained from $p$ and $\varphi$;

$P' = \bigcup_{p \in P} IOp_p$

Each production in a RAG implies several I/O productions. If the occurrences of two RAPs $p_1$, $p_2$ can be connected together, so can the occurrences of two I/O productions derived from $p_1$ and $p_2$, with additional constraints on the input and output sets and on the assignments.

Let $p_1$, $p_2$ be two RAPs, and $p_1'$, $p_2'$ two I/O productions respectively derived from $p_1$ and $p_2$. Suppose that the left-hand side of $p_1'$ appears in the right-hand side of $p_1$ at position $k$.

$p_1 = \langle X_0 \rightarrow X_1 \ldots X_k \ldots X_{n2}, \varphi_{p1} \rangle$

$p_2 = \langle X_0 \rightarrow Y_1 \ldots Y_{n2}, \varphi_{p2} \rangle$

$p_1' = \langle p_1, In_1, Out_1, V_1 \rangle$

$p_2' = \langle p_2, In_2, Out_2, V_2 \rangle$

$Y_0$ and $X_k$ are the same non-terminal symbol. When an occurrence of $p_2'$ is connected at position $k$ to an occurrence of $p_1$, the attributes of $Y_0$ and $X_k$ become the same vertices in the attribute graph of the resulting tree. In notations, it is important that the indexes of the attributes in both occurrences correspond to their positions in the tree, such that further on, there is no ambiguity in the input and output sets and in the assignments.

When we connect occurrences of $p_1$ and $p_2$, the same way as occurrences of $p_1$ and $p_2$, the following constraints apply on input and output sets:

1. $In_1 \cap A_{p_1,X_k} \subseteq (In_2 \cup Out_2) \cap A_{p_2,X_k}$
2. $In_2 \cap A_{p_2,X_k} \subseteq (In_1 \cup Out_1) \cap A_{p_1,X_k}$
3. $Out_1 \cap A_{p_1,X_k} \subseteq In_2 \cap A_{p_1,X_k}$
4. $Out_2 \cap A_{p_2,X_k} \subseteq In_1 \cap A_{p_1,X_k}$

Formula (1) (resp. (2)) indicates that an input attribute of $X_k$ (resp. $Y_0$) should be an input or output attribute of $Y_0$ (resp. $X_k$). Formula (3) (resp. (4)) indicates that an output attribute of $X_k$ (resp. $Y_0$) should be an input
attribute of $Y_0$ (resp. $X_k$). Moreover, still considering the same connection between occurrences of $p_1$ and $p_2$, the following constraints apply on the assignments:

\begin{align}
(5) & \forall a \in In_1 \cap A_{p_1, X_k} [a \in Out_1 \land v_1(a) = (\forall (v'_1(a)) a \in In_1 \land v_1(a) = v_2(a)] \\
(6) & \forall a \in In_2 \cap A_{p_2, X_k} [a \in Out_2 \land v_2(a) = (\forall (v'_2(a)) a \in In_2 \land v_2(a) = v_1(a)] \\
(7) & \forall a \in Out_1 \cap A_{p_1, X_k} [a \in In_2 \land v'_1(a) = v_2(a)] \\
(8) & \forall a \in Out_2 \cap A_{p_2, X_k} [a \in In_1 \land v'_2(a) = v_1(a)]
\end{align}

Formula (5) (resp. (6)) indicates that the value of an input attribute of $X_k$ (resp. $Y_0$) should be equal to the interpretation of the value of the same attribute considered as an output attribute of $Y_0$ (resp. $X_k$). Formula (7) (resp. (8)) indicates that the interpretation of the value of an output attribute of $X_k$ (resp. $Y_0$) should be equal to the value of the same attribute considered as an input attribute of $Y_0$ (resp. $X_k$). Moreover, still considering the interpretation of the value of the same attribute considered as an input attribute of $Y_0$ (resp. $X_k$). In short, a common attribute of both occurrences is either an input attribute for both, or an output attribute for both, or an input attribute for each, or an output attribute for one.

**Example 6** Connecting occurrences of I/O productions: $p_1$ is a RAP; $p_1$, $p_2$, and $p_3$ are I/O productions obtained from $p_1$. An occurrence of $p_2$ can be connected at the position 1 in an occurrence of $p_1$, because $v_1(n_1) = v_2(n_0)$. This is not the case for any occurrence of $p_2$, because $v_1(n_1) \neq v_2(n_0)$. However, an occurrence of $p_3$ can be connected at position 1 in an occurrence of $p_2$, because $v_2(n_1) = v_3(n_0)$.

$$p_1 = \langle \text{ParamFac} \rightarrow \text{ParamFac}, n_0 = n_1 + 1 \land r_0 = r_1 \times n_0 \rangle$$

Let $G$ be an IOAG. The set of input/output derivation trees determined by $G$ is the set of trees in which the occurrences of the I/O productions are correctly connected together. The attribute graph associated with a derivation tree contains directed edges between input and output attributes. An edge links an input attribute $a$ to an output attribute $b$ if $a$ is a parameter of $b$ in the output assignment.

**Fig. 4** shows two such derivation trees and trees and their attribute graphs. Some attributes are spotted out (dashed lines with arrows) because they are global (or external) input attributes, i.e. attributes that are input in every production occurrence they appear in. The other attributes are input in one production occurrence and output in another, if not just output. The directed edges give the direction of the evaluation, i.e. the interpretation of the functional expressions from input to output attributes.

**7. COMPARING INPUT/OUTPUT PRODUCTIONS**

In this section, we study the growth of the input set of a single production and the induced consequences on its output. In an I/O production, turning some unknown attributes into input attributes accrues to change this production into another, potentially with new output. It gives a means to compare I/O productions in terms of input attributes and assignments.

Let $p$ be a RAP. An I/O production $p_1$ derived from $p$ can be transformed into another I/O production $p_2$ also derived from $p$, by extending the input set of $p_1$. The input set of $p_2$ should be the union of the input set of $p_1$ together with this extension. Informally, as the input set grows from $p_1$ to $p_2$, the output set grows and the set of unknown attributes diminishes. When the set of unknown attributes is empty, the I/O production is said to be saturated, and cannot be more transformed.

**Definition 8** (Specialization) Let $p$ be a RAP, and $p_1 = \langle \forall n_1, v_1, \text{Out}, v'_1 \rangle$, $p_2 = \langle \forall n_2, v_2, \text{Out}, v'_2 \rangle$ two I/O productions derived from $p$. $p_2$ is said to be a specialization of $p_1$, which is denoted $p_2 \prec p_1$, if: $n_1 \subseteq n_2 \land \forall a \in \text{In}_1, v_1(a) = v_2(a)$

**Fig. 3** and **Fig. 4** show such derivation trees and trees and their attribute graphs. Some attributes are spotted out (dashed lines with arrows) because they are global (or external) input attributes, i.e. attributes that are input in every production occurrence they appear in. The other attributes are input in one production occurrence and output in another, if not just output. The directed edges give the direction of the evaluation, i.e. the interpretation of the functional expressions from input to output attributes.
We also say that \( p'_2 \) is more saturated than \( p'_1 \).

The consequences of these constraints are the followings: \( \text{Out}_1 \subseteq \text{Out}_2 \) and \( \forall a \in \text{Out}_1, v_1(a) = v_2(a) \).

The relation \( \prec \) is a partial and well-founded order for \( \text{IO}_p \). A unique upper bound, called the empty I/O production, exists; it corresponds to \( p \) with an empty input set. Every I/O production in \( \text{IO}_p \) is a specialization of this one. On the contrary, and in non trivial cases, there are infinitely many incomparable lower bounds which are the saturated I/O productions of \( \text{IO}_p \) (productions without unknown attributes).

The set \( \text{In}_2 - \text{In}_1 \) is the set of input attributes that appear in \( p_2 \) but not in \( p_1 \). Let \( n \) be the cardinal of \( \text{In}_2 - \text{In}_1 \). If \( n = 0 \), then \( p_2 = p_1 \). If \( n = 1 \), \( p_2 \) is said to be a one attribute specialization of \( p_1 \), which is denoted \( p_2 \prec_1 p_1 \).

If \( n \geq 1 \), there exists at least one chain \( p_2 \prec p_1 \prec \ldots \prec p'_{n-1} \prec p'_{n} \) of \( n \) one-attribute specializations from \( p_1 \) to \( p_2 \) (each specialization \( \prec \) adds one new input attribute from \( \text{In}_2 - \text{In}_1 \)).

A specialization \( p'_2 \prec p'_1 \) reveals several parameters to describe \( p_2 \) using \( p_1 \). As every I/O production is the result of at least one chain of one-attribute specializations starting from the empty I/O production, it is straightforward to use a chain to define an I/O production. This would be a word in a language \( \text{SL}_p \) of specializations. Instead of fully specifying every I/O production derived from \( p_1 \), \( \text{SL}_p \) would describe how they share specification parts together.

The language \( \text{SL}_p \) can be represented with an automaton \( A_p \) which states represent I/O productions, and transitions represent extensions for input and output sets and assignments. The initial state is the empty production, and the final states are the saturated productions. Yet, \( A_p \) is infinite since \( \text{IO}_p \) is.

**Example 7** Specializing I/O productions: \( p \prec_1 p_1 \).

\( p_2 \succ p \prec_2 p_1 \succ_2 p_2 \) are I/O productions derived from a RAP \( p_1 \).

\( p \prec_1 \) is the empty I/O production. \( p \prec_1 \) and \( p \succ_2 \) are incomparable saturated I/O productions.

\( p_1 = \text{ParamFac} \to \text{ParamFac} \), \( n_0 = n_1 + 1 \wedge r_0 = r_1 \times n_0 \)

\[ p'_1 = \text{ParamFac} \to \text{ParamFac} \]

\[ p'_2 = \text{ParamFac} \to \text{ParamFac} \]

\[ p'_2 = \text{ParamFac} \to \text{ParamFac} \]

\[ p'_1 \preceq_1 p'_T \]

\[ p'_1 \preceq_2 p'_2 \]

\[ p'_1 \preceq_1 p'_2 \]

\[ p'_1 \preceq_1 p'_1 \]

\[ p'_2 \preceq_1 p'_T \]

The fact that \( A_p \) is infinite is not a good point in practice, so we propose now having a finite automaton. The objective is to partition \( \text{IO}_p \) into finitely many classes. Each class should verify the equivalence of the input and output sets of the I/O productions, and the equivalence of the output assignments in terms of functional expressions. The abstraction relies in grouping together I/O productions with the same “behavior” but different input assignments.

There exist algorithms that compute an abstract interpretation of Horn clauses by using an abstract domain [ground, nonground] for the values of the variables [28, 29, 30]. In the case of I/O productions, we will respectively use known instead of ground and unknown instead of nonground, and produce finite automata with abstract interpretations of the logical formulas in the productions.

Considering a relational production \( p \), we present an algorithm that starts at the (singleton) class containing the empty I/O production derived from \( p \). Finding a new class is considering a one-attribute specialization concerning a previously unknown attribute \( a \). The abstract interpretation of \( (\varphi_h \land a = \text{known}) \) gives the set of (output) attributes which values are known, indicating the set of I/O productions contained in the new class. The algorithm terminates when no new class can be produced from the classes already found. A class is represented by a pair \( (\text{In}, \text{Out}) \) where \( \text{In} \) is the set of input attributes which values are known and \( \text{Out} \) is the set of output attributes which values are known. The procedure abstract-interpretation is like the one developed in Barbar et al. [28] for the computation of abstract interpretation of Prolog programs.
Consider an I/O derivation tree $t_f$ which is only built with occurrences of empty I/O productions. First, we choose an unevaluated attribute in $T_f$. Generally, any attribute belongs to two production occurrences; otherwise it is an attribute of a leaf or of the root of $T_f$. Giving a value for this attribute is specializing the two production occurrences.

Consequently, new output values are locally produced in each occurrence (see Section 6). At this point, the tree is no more a valid I/O derivation tree, but now the connected production occurrences receive the output values as input, so they are specialized in their turn.

Such local specializations propagate in the tree from occurrences to occurrences, and stop where it gives no output. The tree is globally specialized through that propagation of local specializations, and the result is an I/O derivation tree $t$ verifying $t \prec t_f$. In fact, we can write $t \prec t_f$ because we did choose only one new input attribute in the whole tree.

We may repeat the preceding steps, until all the production occurrences in the tree are saturated, resulting in a saturated tree.

Example 9 Tree Specialization: starting from an empty I/O tree, the value of an attribute is given, leading to the propagation (dotted arrows) of specializations.

We present an algorithm for the incremental propagation in a derivation tree. This algorithm uses abstract values for the attributes, as shown in Section 6. It starts with a new input attribute input-a for the tree. A pair $(p, a)$ denotes a production occurrence and a new input attribute for this production occurrence. The algorithm uses such pairs to specify what attributes remain to be propagated in some productions.

The procedure contains takes an attribute in the tree and returns the set of production occurrences to which it belongs. The procedure state takes a production occurrence in the tree and returns the pair (In, Out) of
current input and output attributes of this occurrence. The procedure `specialize` takes a production occurrence and a new input attribute and realizes the specializations of the production occurrence, as described in Section 6. The symbol $\times$ denotes the Cartesian product of sets.

```plaintext
P= contains(input-a);
S=P \times \{input-a\};
While S \neq \emptyset
    { 
        S=S-(\{(p,a)\});
        (In',Out')= state(p);
        specialize(p, In'\cup\{a\});
        (In'',Out')= state(p);
        For each a' \in Out'-Out
            { 
                P= contains(a');
                S=S \cup (p \times \{a'\});
            }
    }
```

9. CONCLUSION

In this paper, we released the inherited and synthesized natures of attributes in AGs, by considering them more like variables in logical formulas, than variables in static functional expressions. Yet, we used functional expressions to reflect attribute evaluation, giving configurations in terms of input and output attributes. For each formulae, we studied the whole set of valid configurations, instead of considering just one configuration.

Using RAGs seems a trivial task: one gives an ACFG, attributes for non-terminals and relations that bind these attributes together. Then, one supposes that the relations are satisfied all along the interactive edition of a derivation tree, including tree growing and attribute evaluation.

In fact, the operational point strongly depends on the logical part of the grammar. What if we cannot construct functional expressions from logical formulas? And even if these expressions are constructed for productions, we are faced with evaluation problems relating to non-local attribute dependencies in derivation trees. This cannot be done just by separately considering each production.

This problem is one of a wide class of evaluation problems that all depend on one point: circular attribute dependencies. In general, knowing to locally use some formulas is not sufficient for conjunctions of these formulas, because of strong connections between the variables. We hope to give some results concerning these problems in future studies.

However, we believe that local specializations and propagations can improve the expressiveness of semantic rules and the comprehensibility of the mechanism of attributes evaluation. In the case that one accepts the induced limitations, RAGs together with specialized productions (denoted by $\prec$) represent an interesting formal tool to specify dynamic attribute dependencies. Incremental evaluation is a useful consequence of dependency.

REFERENCES


