Two Transformations on Attribute Grammars Improving the Complexity of their Evaluation

ÉVA GOMBÁS and MIKLÓS BARTHA

A. JÓZSEF UNIVERSITY
BOLYAI INSTITUTE
SZEGED, ARADI VÉRTANÚK TERE 1.
6720 — HUNGARY

1. Introduction

Several papers have been written recently on designing efficient evaluators for attribute grammars (*AGs*). Some of these papers (e.g. [6], [7]) provide techniques to optimize the time complexity of the evaluators for certain classes of *AGs* (the class of absolutely noncircular *AGs* in the referenced papers), other ones (e.g. [5], [9]) try to reduce the storage requirement of the evaluators. The same goal of these papers is, however, to optimize evaluation by improving the evaluator itself in some respect. Our aim is to improve the *AG* to be evaluated — by the application of a suitable transformation — not the evaluator (of a fixed type) by which the evaluation is actually performed. Of course, this approach cannot provide general optimization results as the previous one, but in some cases it can be quite powerful. In this paper we present two transformation techniques and show how they work in restricted classes of *AGs*.

It is known that every *AG* can be converted to an equivalent one which uses only synthesized (s-)attributes. The underlying idea is the following: the value of a new s-attribute computed at any (nonterminal) node of a derivation tree becomes a function that describes how the corresponding old s-attribute depends on the old inherited (i)-attributes at the same node of the tree in the original *AG*. An exact algebraic formulation of this method, which will be referenced as the "convert to functional domains" (c.f.d.) principle, can be found e.g. in [2]. On the one hand it is clear that for an *AG* having s-attributes only, the structure of the evaluator is the simplest possible (only one left-to-right pass is needed). On the other hand, it is in general much more costly to deal with functional domains during evaluation, than to make multiple visits to the nodes of the derivation trees. Therefore, the c.f.d. principle cannot be used as a general transformation technique improving the complexity of evaluation. But it can be used successfully in a less drastical form for restricted classes of *AGs*. Indeed, both transformations presented in this paper are eventually applications of the c.f.d. principle.

The trick we are going to apply in the first transformation is based on the following well-known method of designing a one-pass assembler. If a post-definite label occurs in some instruction of the source program, then the assembler will translate an incomplete object code from that instruction, and it will update the adress field of this object code instruction (chained together with all those instructions referring to the same postdefinite label) as soon as the referenced label becomes definite. The term "reference to a post-definite label" corresponds to the term "reference to an i-attribute occurrence on the right-hand side of the semantic rules" in an *AG*, thus, we simplify evaluation by ignoring or postponing the computation of certain i-attribute instances at the nodes of the derivation trees. An example for this transformation is given in Sect. 2, and it is generalized in Sect. 4. To characterize the *AGs* for which the transformation is applicable, we introduce the class of *VSE AGs* in Sect. 3 and investigate the basic properties of this class. The class of *VSE AGs* is the visit-oriented counterpart of the class of *ASE AGs* [8], and it is in strong connection with the classes of *OAG* [10] and simple multi-visit *AGs* [3].

The second transformation technique is described in Sect. 5. It uses the c.f.d. principle with its full power, i.e. all the i-attributes are eliminated. The transformation can be applied, however, for a more restricted class of *AGs*, the class of linear string-valued *AGs*. By linearity we mean that in the Bochmann normal form of the semantic rules every attribute occurrence can be referenced at most once on the right-hand sides. This concept was originally defined for attributed tree transducers in [1].

In Sect. 4 we introduce two complexity measures for the evaluation of the complete derivation trees (*cd*-trees) of an *AG* under a fixed visit-oriented evaluator (cf. [9]). The visit complexity of a *cd*-tree t is the average number of visits made to a node during the evaluation of t. The pure computation complexity of t is the total amount of computation needed to assign value to all the attribute instances of t. The collection of pairs constructed from these two numbers for all the *cd*-trees, together with the type of the evaluator characterizes the evaluation complexity of the *AG* in a satisfactory way. We shall show that our transformations indeed reduce the complexity of evaluation in this sense.

2. Definitions and Examples

Although we assume familiarity with attribute grammars [11], we repeat some of the basic concepts here to fix our notations for the forthcoming sections.

An attribute grammar is a 5-tuple

$$\mathcal{G} = (G, A, v, \{D_a | a \in A\}, \{r_p | p \in P\}),$$

where

$G=(N, T, P, S')$ is a context-free grammar, called the underlying *CF*-grammar of $\mathcal{G}$. N and T denote the set of nonterminal and terminal symbols, respectively; P is the set of productions and $S' \in N$ is the start symbol. We assume that G is "augmented" by the top-production $S' \to S$ ($S \in N$ as well), so that S' does not appear in any other production. We shall write a production $p \in P$ in the form

$$p\colon\ F_0 \to w_0 F_1 w_1 \ldots F_n w_n,$$

where $F_j \in N$ and w_j is a string of terminal symbols for each $j \in [0, n]$. For nonnegative integers k, l, $[k, l]$ denotes the set $\{k, k+1, \ldots, l\}$; $[k]$ is a shorthand for $[1, k]$, as usual. Since terminal symbols play no essential role in attribute grammars, the above production will rather be written as p: $F_0 \to F_1 \dots F_n$. Accordingly, by a node of a derivation tree we always mean a nonterminal node.

$A = A_S \cup A_I$ is a finite set of attributes, $A_S \cap A_I = \emptyset$. The elements of A_S and A_I are called synthesized (s-) and inherited (i-) attributes, respectively.

v: $N \to 2^A$ is a mapping; if $a \in v(F)$, then we say that $F \in N$ has attribute a. $S(F)$ and $I(F)$ will denote the sets $v(F) \cap A_S$ and $v(F) \cap A_I$, respectively. We assume that every nonterminal has at least one attribute, S' has only s-attributes.

$\{D_a | a \in A\}$ is the family of attribute domains. An attribute $a \in A$ takes its value from the set D_a.

$\{r_p | p \in P\}$ is the family of semantic rules. If p: $F_0 \to F_1 \dots F_n$, then r_p consists of the following rules (equations):

$$a_0(F_{j_0}) = f(a_1(F_{j_1}), \dots, a_m(F_{j_m}))$$

for each

$$a_0 \in \begin{cases} S(F_0) & \text{if } j_0 = 0; \\ I(F_j) & \text{if } j = j_0 > 0, \end{cases}$$

where $j_i \in [0, n]$ and $a_i \in v(F_{j_i})$ for every $i \in [0, m]$; f: $D_{a_1} \times \dots \times D_{a_m} \to D_{a_0}$ is a (computable) function. The above rule will be abbreviated later on as $a_0(F_{j_0}) = rhs\ (a_0, F_{j_0})$. We say that $a_i(F_{j_i})$ is a definition or a reference to attribute occurrence a_i of nonterminal (occurrence) F_{j_i} in a rule corresponding to production p depending on whether it occurs on the left-hand side or right-hand side of the rule. If there are several occurrences of the same nonterminal in p, then these occurrences will be distinguished by subscripts, as usual. The condition that if $a_i(F_{j_i})$ is referred on the right-hand side in any rule of r_p, then

$$a_i \in \begin{cases} S(F_j) & \text{if } j = j_i > 0 \\ I(F_0) & \text{if } j_i = 0 \end{cases}$$

is the well-known Bochmann normal form (n.f.) condition. We shall violate this condition only if this makes the semantic rules shorter to write down.

The underlying idea of the following example is well-known from compiler literature (see e.g. [12]). We show how to compile a Boolean expression so that the length of the generated code depends only on the relations which the expression is built up from.

Example 2.1. Let $\mathcal{G}$ be the following AG. The underlying CF-grammar G has productions:

$$B' \to B;\ B \to B \text{ or } D | D;\ D \to D \text{ and } C | C;\ C \to \text{not} C | (B) | R,$$

where B', B, D, C and R are all the nonterminals with B' being the start symbol. (Note that the syntax satisfies both the LR-1 and operator precedence conditions.) Of course, G is incomplete in the sense that it is impossible to generate any terminal string using the above productions only. Therefore we assume that the grammar G is "continued" in such a way that the nonterminal R derives relations e.g. between arithmetic expressions. This part of the grammar is, however, not relevant from the point of view of our example. In this way G generates well-formed Boolean expressions, and by $\mathcal{G}$ we would like to translate these expressions to assembly language code. To this end we define the following attributes and corresponding domains:

code:	string of assembly instructions, the generated code;
len:	integer, the length of the generated code;
loc:	integer, the location (or adress) of the first instruction of the generated code;
$\uparrow$:	integer, the location where control should be passed if the corresponding Boolean expression is true;
$\downarrow$:	integer, the location where control should be passed if the corresponding Boolean expression is false.

code and len are s-attributes, while loc, $\uparrow$ and $\downarrow$ are i-attributes. Every nonterminal, except B' has all these attributes, $\nu(B')=\{\text{code}\}$. The semantic rules corresponding to the productions are listed below.

$B' \to B$	code (B') = code (B), loc $(B)=l_0$, $\uparrow(B)=\uparrow_0$, $\downarrow(B)=\downarrow_0$

(l_0, $\uparrow_0$ and $\downarrow_0$ are constant locations).

$B_1 \to B_2$ or D	code (B_1) = code (B_2) code (D), len (B_1) = len (B_2) + len (D), loc (B_2) = loc (B_1), loc (D) = loc (B_1) + len (B_2), $\uparrow(B_2)=\uparrow(D)=\uparrow(B_1)$, $\downarrow(B_2)$ = loc (D), $\downarrow(D)=\downarrow(B_1)$.
$D_1 \to D_2$ and C	code (D_1) = code (D_2) code (C), len (D_1) = len (D_2) + len (C), loc (D_2) = loc (D_1), loc (C) = loc (D_1) + len (D_2), $\uparrow(D_2)$ = loc (C), $\uparrow(C)=\uparrow(D_1)$, $\downarrow(D_2)=\downarrow(C)=\downarrow(D_1)$.
$C_1 \to$ not C_2	code (C_1) = code (C_2), len (C_1) = len (C_2), loc (C_2) = loc (C_1), $\uparrow(C_2)=\downarrow(C_1)$, $\downarrow(C_2)=\uparrow(C_1)$.

In the remaining four productions: $B\to D$, $D\to C$, $C\to(B)|R$ the value of the attributes is transferred without any change from one nonterminal to the other. □

Let t be a cd-tree of $\mathcal{G}$, and assume that the value code (u) of attribute instance code at any node u labelled by R is such a code that, when executed, it passes control to $\uparrow(u)$ or $\downarrow(u)$ depending on whether the corresponding relation below u is true or false, respectively. Then it is obvious from the semantic rules that this property is inherited by all the nodes of t. Clearly, the augmentation $B'\to B$ is not necessary in practice, because $\mathcal{G}$ is just a portion of a large AG defining the compiler semantics of a programming language, and the locations l_0, $\uparrow_0$, $\downarrow_0$ are inherited from the context.

To define dependency relations between the attribute occurrences of a production in an AG we assume that the family of domains $\{D_a|a\in A\}$ is extended to a many sorted algebra, which is called the attribute algebra of $\mathcal{G}$, and the functions f on the right-hand sides of the semantic rules are polynomials of the attribute algebra. Thus, if p: $F_0\to F_1...F_n\in P$, then we say that attribute occurrence $a(F_i)$ depends on $b(F_k)$ in p if there is an equation of the form $a(F_i)=f(...b(F_k)...)$ among the n.f. of the rules in r_p. The dependency graph for production p (denoted by $dp(p)$) is the graph having as nodes the attribute occurrences of all nonterminals F_j of p, $j\in[0,n]$, and in which there is an arc running from node $b(F_k)$ to node $a(F_i)$ iff $a(F_i)$ depends on $b(F_k)$ in p. See Fig. 1 for the dependency graph for production $B\to B$ or D of Example 2.1.

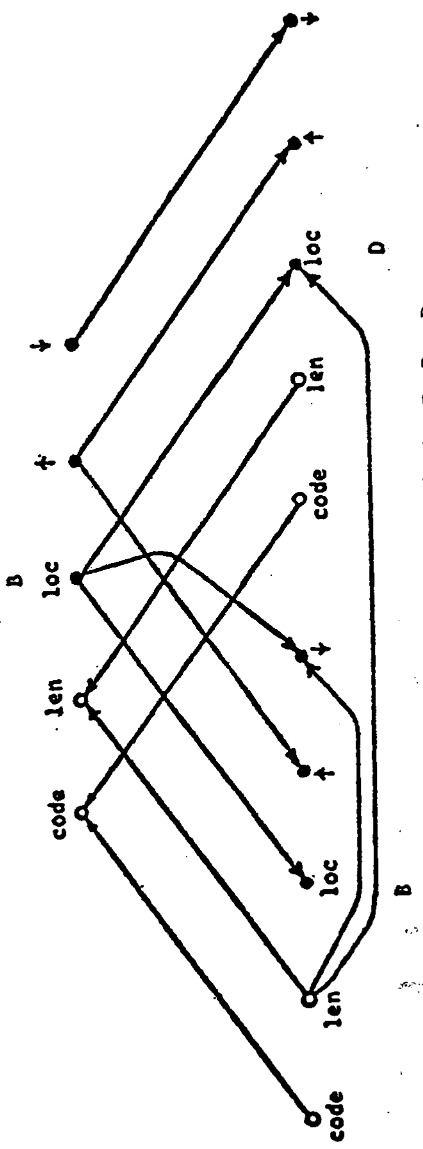

Fig. 1. The dependency graph for production $B \to B$ or D.

Several tree walking strategies exist for evaluating the *cd*-trees of an *AG*. The reader is assumed to be familiar with the notion of visit and pass, and with at least some of the papers [3], [4], [8], [10]. It can be seen directly from Fig. 1 that the *cd*-trees of our example *AG* $\mathcal{G}$ cannot be evaluated in one pass, nor in one visit $(\downarrow(B_2)$ depends on len (B_2) in the production $B_1 \to B_2$ or $D)$. On the other hand it is clear that $\mathcal{G}$ satisfies the *ASE* property [8]. loc and len can be computed in the first left-to-right pass, and the remaining attributes in the second pass (which can be either left-to-right or right-to-left).

$\mathcal{G}$ can be transformed into an equivalent one-pass *AG* $\mathcal{G}'$ by the following trick. We drop the *i*-attributes $\uparrow$ and $\downarrow$, and compute the code of any sub-Boolean expression by leaving holes in the adress field of the crucial "jump to $\uparrow(R)$" and "jum to $\downarrow(R)$" instructions generated while compiling the relations contained in that subexpression. At the same time, we maintain two chains to register the locations of the "$\uparrow$-holes" and "$\downarrow$-holes" in the code. The holes will be filled in by the "old" values of $\uparrow$ and $\downarrow$ computed in $\mathcal{G}$ at the rootnode of the subexpression, but in $\mathcal{G}'$ we compute and fill in these values only later, when it becomes possible moving upwards in the derivation tree. The explicit construction is the following.

Example 2.2. The underlying *CF*-grammar of $\mathcal{G}'$ is the same grammar G, and it is equipped with the following attributes and corresponding domains:

$\langle$code, $\uparrow c$, $\downarrow c\rangle$:	a triple consisting of the generated code and two chains containing the locations of $\uparrow$-holes and $\downarrow$-holes in the code,
len:	integer, the length of the code,
loc:	integer, the location of the first instruction of the code.

Again, every nonterminal except B' has all these attributes, but now loc is the only *i*-attribute; $v(B')=\{\text{code}\}$. In fact code, $\uparrow c$ and $\downarrow c$ will be treated as three different *s*-attributes; we gathered them up just for the sake of the generalization we are going to introduce in Sect. 4. The semantic rules of $\mathcal{G}'$ are the following.

$B' \to B$ code $(B')=\mathit{rollup}\,(\mathit{rollup}\,(\text{code}\,(B), \uparrow c(B), \uparrow_0), \downarrow c(B), \downarrow_0)$,
loc $(B)=l_0$;

where *rollup* (w, c, x_0) is a function which substitutes iteratively a constant string x_0 into another string w at all the locations registered in a chain c.

$B_1 \to B_2$ or D code $(B_1)=\mathit{rollup}\,(\text{code}\,(B_2), \downarrow c(B_2), \text{loc}\,(D))$ code (D),
$\uparrow c(B_1)=\uparrow c(B_2)\uparrow c(D)$, $\downarrow c(B_1)=\downarrow c(D)$, loc $(B_2)=$loc (B_1),
len $(B_1)=$len $(B_2)+$len (D), loc $(D)=$loc $(B_1)+$len (B_2).

$D_1 \to D_2$ and C code $(D_1)=\mathit{rollup}\,(\text{code}\,(D_2), \uparrow c(D_2), \text{loc}\,(C))$ code (C),
$\uparrow c(D_1)=\uparrow c(C)$; $\downarrow c(D_1)=\downarrow c(D_2)\downarrow c(C)$, loc $(D_2)=$loc (D_1),
len $(D_1)=$len $(D_2)+$len (C), loc $(C)=$loc $(D_1)+$len (D_2).

$C_1 \to$ not C_2 code $(C_1)=$code (C_2), $\uparrow c(C_1)=\downarrow c(C_2)$, $\downarrow c(C_1)=\uparrow c(C_2)$,
len $(C_1)=$len (C_2), loc $(C_2)=$loc (C_1).

The rules corresponding to the remaining four productions are again "simple" rules. □

It is clear that $\mathcal{G}$ and $\mathcal{G}'$ are equivalent in the sense that they compute the same code for every Boolean expression. To compare the evaluation complexity of $\mathcal{G}$ and $\mathcal{G}'$ we make the following two observations.

a) $\mathcal{G}'$ is clearly one-pass.

b) In $\mathcal{G}'$ we have to compute the "old" values of attribute instances $\uparrow$ and $\downarrow$ only at certain nodes of a *cd*-tree (i.e. at exactly those points where a chain must be rolled up), and not both at all nodes as we do it in $\mathcal{G}$. For this reason we can say that, although the operations of maintaining the chains and rolling them up (which corresponds to a substitution of depth one) bring some extra cost into the evaluation, the total amount of computation needed to evaluate a *cd*-tree by $\mathcal{G}'$ is approximately the same as by $\mathcal{G}$.

Thus, by a) $\mathcal{G}'$ is more efficient than $\mathcal{G}$.

3. The Visit-Oriented Semantic Evaluator

We would like to extend the transformation technique outlined in Example 2.2 to a restricted subclass of simple multi-visit *AGs* (see [3]). The class that we are going to introduce — the class of *VSE AGs* — is the visit-oriented counterpart of the class of *ASE AGs* introduced in [8]. Those familiar with this work of Jazayeri and Walter know that it is not clear from the paper whether the authors mean the *ASE* property in a global sense, i.e. for all the attributes, or in a local sense, i.e. for all the attribute occurrences of the semantic rules. We assume here that they mean it in the global sense. Anyway, this question is not too relevant, and Proposition 3.2 shows the obvious connection between the two alternatives. The local version of the *ASE* property was redefined in [4] in a generalized form, with the new name simple multi-*ALT*.

The *VSE* Property.

In the sequel let $\mathcal{G}=(G, A, v, \{D_a|a\in A\}, \{r_p|p\in P\})$ with $G=(N, T, P, S')$ be a fixed *AG*.

Definition 3.1. The localized grammar of $\mathcal{G}$ is the *AG*

$$\mathrm{lc}(\mathcal{G}) = (G, A', v', \{D_{a'}|a'\in A'\}, \{r'_p|p\in P\}),$$

where

— $A' = \{(a, F)\in A\times N | F\in N,\ a\in v(F)\}$;

— $v'(F) = \{(a, F)| a\in v(F)\}$ for all $F\in N$;

— $D_{(a,F)} = D_a$ for all $F\in N$, $a\in v(F)$;

— if $p\in P$ is of the usual form and

$$a_0(F_{j_0}) = f(\ldots a_i(F_{j_i})\ldots)$$

is in r_p, then there exists a corresponding rule in r'_p of the form:

$$(a_0, F_{j_0})(F_{j_0}) = f(\ldots(a_i, F_{j_i})(F_{j_i})\ldots)$$

and r'_p consists of exactly these rules. □

Proposition 3.2. $\mathcal{G}$ is strictly alternating simple multi-*ALT* iff $lc(\mathcal{G})$ is *ASE*.

Proof. Obvious. □

Let B be a finite set. By an ordered partition of B we mean a finite sequence $(B_1, ..., B_m)$ of subsets of B such that $\bigcup_{i=1}^{m} B_i = B$. (Note that any of the B_i might be $\emptyset$.) Recall from [3] that a set of ordered partitions for $\mathcal{G}$ is a set Π containing an ordereu partition $\pi(F)$ of $v(F)$ for each $F \in N$. Π is called a simple multi-visit (*smv*) set of (ordered) partitions if for every *cd*-tree there is a computation sequence (cf. [3]) for it respecting Π. $\mathcal{G}$ is *smv* if there exists an *smv* set of partitions for it.

Definition 3.3. Let Π be an *smv* set of partitions for $\mathcal{G}$ and $\varphi = (A_1, ..., A_m)$ an ordered partition of A. $\mathcal{G}$ is *m-VSE* with respect to (w.r.t.) (Π, φ) if the following condition holds. For every $F \in N$, if $\pi(F) = (B_1, ..., B_{k_F})$, then there exists an injective and monotonic mapping $\varrho_F\colon [k_F] \to [m]$ such that $B_k \subseteq A_{\varrho_F(k)}$ for each $k \in [k_F]$. That is, $\pi(F)$ is the projection of φ to $v(F)$. $\mathcal{G}$ is *(m-)VSE* if it is *(m-)VSE* w.r.t. some (Π, φ). In this case φ is called a *VSE* partition for $\mathcal{G}$. □

Example 3.4. The *AG* of Example 2.1 is 2-*VSE* w.r.t. (Π, φ), where

$$\varphi = (\{\text{loc}, \text{len}\}, \{\uparrow, \downarrow, \text{code}\}),$$

and for each $F \in N \setminus \{B'\}$, $\pi(F) = \varphi$; $\pi(B') = (\{\text{code}\})$. □

Let $\mathcal{G}$ be *m-VSE* w.r.t. (Π, φ). The sets B_k, $k \in [k_F]$ in $\pi(F) = (B_1, ..., B_{k_F})$ are called the local visit-sets of F in contrast with the "global" visit-sets $A_1, ..., A_m$ in φ. If $\varrho_F(k) = c$, then we shall say that c is the global visit-number (*gv*-number) of the local visit-set $B_k(F)$. Let $p\colon F_0 \to F_1 ... F_n \in P$ with

$$\pi(F_j) = (B_1^j, ..., B_{k_{F_j}}^j)$$

for each $j \in [0, n]$. The visit sequence (see [3]) of the visit-set $B_k^0(F_0)$, $k \in [k_{F_0}]$ in p (denoted by $Vs_p(B_k^0)$) is a concrete list of descendant visit-sets, i.e. a list consisting of some visit-sets $B_l^j(F_j)$, $j \geq 1$. Let c be the *gv*-number of $B_k^0(F_0)$. We associate a global visit sequence $Gvs_p(c)$ with $Vs_p(B_k^0)$ in a natural way: $Gvs_p(c)$ is a list of pairs of integers such that to any member $B_l^j(F_j)$ on $Vs_p(B_k^0)$ there corresponds a member (d, j) on $Gvs_p(c)$ (at the same position, of course), where d is the *gv*-number of $B_l^j(F_j)$. In this way we can consider Gvs_p as a vector of m lists. For each $c \in [m]$, if $c = \varrho_{F_0}(k)$ for some $k \in [k_{F_0}]$, then $Gvs_p(c)$ is the above list, otherwise $Gvs_p(c)$ is the empty list.

The *cd*-trees of $\mathcal{G}$ can now be evaluated by the help of the following procedure:

```
procedure visit (c, u); integer c; node u;
comment c is a global visit-number;
comment let p: $F_0 \to F_1 ... F_n$ be the production applied at u;
begin
    compute the instances of $I(F_0) \cap A_c$ at node u;
    for i=1 to length (Gvs_p(c)) do
    begin
        comment take the i-th member of the list Gvs_p(c);
        (d, j)=take (i, Gvs_p(c));
        comment make a visit to the j-th son of u;
        visit (d, son (j, u))
    end;
    compute the instances of $S(F_0) \cap A_c$ at node u
end
```

The part of the "main program" which evaluates a *cd*-tree t can be written as:

```
for c=1 to m do
    visit (c, root(t)).
```

The main point of the procedure visit above is that we evaluate the local visit-sets at each node as being the projections of the corresponding known global visit-sets. This makes the procedure so simple compared with e.g. the simple multi-visit evaluation procedure in [4].

There are some situations when it is more appropriate to compute the final value of certain attribute instances by several assignments placed in different visits. To handle such situations we allow some instances of $v(F_0)\cap A_c$ at node u to be "marked" in the procedure *visit* (c, u) above. The value of these attribute instances can be updated later by the call of the following procedure.

```
procedure update (c, u); integer c; node u;
comment let $F_0$ be the label of u;
begin
    recompute the marked instances of $v(F_0)\cap A_c$ at node u;
    modify the present
    marking for the sake of further updates to u, if necessary;
end
```

Since the procedure *update* can be considered to be a visit of depth zero, we assume that update visits are also placed as distinguished members onto the global visit-sequence lists. Update visits will be used in the next section.

The following two definitions are adopted from [3]. An ordered partition $\pi = (B_1, \ldots, B_m)$ of a subset of A is reduced if $B_k \neq \emptyset$ for any $k\in[m]$. π is good if, whenever $m\geqq 2$, B_1 contains at least one s-attribute, B_m contains at least one i-attribute and for every $k\in[2, m-1]$, B_k contains both i- and s-attributes.

Lemma 3.5. If $\mathcal{G}$ is *VSE* w.r.t. (Π, φ), then φ can be assumed to be reduced.

Proof. The result is a direct consequence of Lemma 2.1 in [3]. $\square$

Theorem 3.6. If $\mathcal{G}$ is VSE w.r.t. (Π, φ), then φ can be assumed to be good.

Proof. By Lemma 3.5 we can assume that $\varphi=(A_1, \ldots, A_m)$ is reduced. Moreover, we can assume that Π is also reduced, i.e. $\pi(F)$ is reduced for every $F\in N$. Suppose first that A_c contains only s-attributes for some $c\in[2, m]$. Let F be a nonterminal such that $\pi(F)=(B_1, \ldots, B_{k_F})$ and there exists $k\in[k_F]$ with $c=\varrho_F(k)$. Define

$$\pi'(F) = \begin{cases} \pi(F) & \text{if } k=1 \text{ or } \varrho_F(k-1) < c-1, \\ (B_1, \ldots, B_{k-1}\cup B_k, \ldots, B_{k_F}) & \text{if } \varrho_F(k-1) = c-1. \end{cases}$$

By virtue of Theorem 2.2 in [3], the replacement of $\pi(F)$ by $\pi'(F)$ in Π preserves the *smv* property. Thus, making this replacement for all appropriate $F\in N$ we get a set Π' which is still *smv* and, together with $\varphi'=(A_1, \ldots, A_{c-1}\cup A_c, \ldots, A_m)$ it satisfies the *VSE* condition. The same argument shows that, if A_c contains only i-attributes for some $c\in[m-1]$, then the partition $\varphi''=(A_1, \ldots, A_c\cup A_{c+1}, \ldots, A_m)$ together with its projections $\{\pi''(F)|F\in N\}$ remains *VSE*. In this way it is clear that, applying a finite number of such transformations on φ and Π, we shall end up with a good ordered partition $\tilde{\varphi}$. It must be noted, however, that the corresponding set of *smv* partitions $\tilde{\Pi}$ need not be good at all. □

Testing the *VSE* Property.

Lemma 3.7. $\mathcal{G}$ is *smv* iff *lc* ($\mathcal{G}$) is *VSE*.

Proof. Obvious. □

Theorem 3.8. The following problems are *NP*-complete:
(i) deciding whether an arbitrary *AG* is *VSE*,
(ii) deciding whether an arbitrary *AG* is 2-*VSE*.

Proof. (i) is an immediate consequence of Lemma 3.7 and Theorem 4.1 in [3], because the size of *lc* ($\mathcal{G}$) is polynomially related to the size of $\mathcal{G}$. To prove (ii) it is enough to observe that the example *AG* $\mathcal{G}(F_0)$ constructed in the proof of Theorem 4.1 in [3] for a Boolean expression F_0 is simple 2-visit iff $lc\,(\mathcal{G}(F_0))$ is 2-*VSE*. (Of course, this is not true for an arbitrary *AG*.) □

In spite of these negative results it is worth computing the relation "forced before" (see [3]) of the attributes as it was done also in [10]. Let $r\subseteq A\times A$ be any relation and p: $F_0\to F_1\dots F_n\in P$. Define the graph *idp* (p, r) to be an extension of *dp* (p) with the arcs

$$a(F_j)\to b(F_j) \quad \text{iff } arb$$

for all $j\in[0, n]$, and let $idp\,(p, r)^+|F_j$ denote the restriction of the transitive closuie of $idp\,(p, r)$ to the attribute occurrences of the j-th nonterminal (occurrence) in p. Then the relation forced before is the smallest relation $fb\subseteq A\times A$ such that

$$idp\,(p,fb)^+|F_j\subseteq fb|v(F_j)$$

for all productions p: $F_0\to F_1\dots F_n$ and each $j\in[0, n]$. Clearly, fb must be respected by any *VSE* partition φ for $\mathcal{G}$. It is easy to give an algorithm which computes the relation fb in polynomial time (see [10] for a similar construction). When fb is computed and it is cycle free, then we can either attempt to construct a *VSE* partition φ directly, as it was done in [10], or to design a more complicated backtrack algorithm to search for a suitable φ. We must know, however by Theorem 3.8, that a really good backtrack algorithm will presumably have exponential time complexity.

4. Improving the evaluation of VSE AGs

We start this section by introducing two complexity measures for the evaluation of the *cd*-trees of an *AG*. For a *cd*-tree t define the visit complexity of t to be the ratio of the total number of visits to the nodes of t during a concrete visit-oriented evaluation (recall that every noncircular *AG* is at least pure multi-visit), and the number of (nonterminal) nodes of t. The pure computation complexity of t is the total amount of computation needed to assign value to all the attribute instances of t (the number of visits is irrelevant here). Suppose that $\mathcal{G}$ and $\mathcal{G}'$ are two equivalent *AG*, i.e. they have the same underlying *CF*-grammar and they compute the same values at the root S' of every *cd*-tree, just by the help of different sets of attributes. To compare the efficacy of $\mathcal{G}$ and $\mathcal{G}'$ we have to consider three points.

1. The evaluator applied for $\mathcal{G}$ and $\mathcal{G}'$ (i.e. pure multivisit, simple multi-visit, *ASE*, *VSE*, etc.).
2. The visit complexity of each *cd*-tree in $\mathcal{G}$ and $\mathcal{G}'$.
3. The pure computation complexity of each *cd*-tree in $\mathcal{G}$ and $\mathcal{G}'$.

We say that $\mathcal{G}'$ is more efficient than $\mathcal{G}$ if $\mathcal{G}'$ is not worse than $\mathcal{G}$ in any of the above three respects, and it is strictly better in at least one of them. From this point of view the *AG* $\mathcal{G}'$ of Example 2.2 is indeed more efficient than the *AG* $\mathcal{G}$ of Example 2.1, because of the reasons a) and b) explained at the end of Sect. 2.

To generalize the transformation technique described in Example 2.2 assume that $\mathcal{G}$ is *m-VSE* w.r.t. (Π, φ), furthermore it satisfies the following three conditions.

(C1) For every $p \in P$ and $c \in [m]$, if (d, j) is a member of $Gvs_p(c)$, then $d \leqq c$.

(C2) There exists a distinguished *gv*-number $v \in [2, m]$ with the following property. Let $p\colon F_0 \to F_1 \dots F_n$ be any production, and suppose that F_0 has a local visit set $B(F_0)$ the *gv*-number of which is v. If $C(F_j)$, $j \in [n]$, is a member of $Vs_p(B)$ such that some *i*-attribute occurrence $b(F_j) \in C(F_j)$ depends on an attribute occurrence $b'(F_l) \in D(F_l)$ in p, then the *gv*-number of $D(F_l)$ is not equal to (or equivalently, it is strictly less than) v, except when all the three conditions below are satisfied:

a) $D(F_l) \equiv B(F_0)$,
b) the *gv*-number of $C(F_j)$ is v,
c) b' is also an *i*-attribute and the semantic rule for $b(F_j)$ is: $b(F_j) = b'(F_0)$.

We shall say that such an exception rule is a simple rule.

(C3) *$A_v \cap A_S$ contains only string-valued attributes.*

<u>Construction of the simplified *AG* $\mathcal{G}'$.</u>

On the analogy of Example 2.2 we define the *AG*

$$\mathcal{G}' = (G, A', v', \{D'_a \mid a \in A'\}, \{r'_p \mid p \in P\})$$

as follows:

$A' = A \setminus A_v \cup A'_v$, where if

$$A_v \cap A_s = \{\alpha_1, \dots, \alpha_{s_v}\} \quad \text{and} \quad A_v \cap A_I = \{\beta_1, \dots, \beta_{i_v}\},$$

then

$$A'_v = \{(\alpha_y, c_1, \dots, c_{i_v}) \mid y \in [s_v]\} \cup \{\beta_1, \dots, \beta_{i_v}\}.$$

$A'_I = A_I$, $A'_S = A' \setminus A'_I$. In the "chained" *s*-attribute $(\alpha_y, c_1, \dots, c_{i_v})$, c_z $(z \in [i_v])$ represents the chain of those locations which point to the "β_z-holes" in the string corresponding to attribute α_y. Note that by Theorem 3.6 we can assume that $i_v \geqq 1$.

For any $F \in N$ we first define the set

$$v''(F) = v(F) \setminus A_v \cup \{(\alpha_y, c_1, \dots, c_{i_v}) \mid y \in [s_v],\ \alpha_y \in v(F)\},$$

then define

$$v'(F) = \begin{cases} v''(F) & \text{if } v \text{ is the greatest } gv\text{-number at } F, \\ v''(F) \cup \{\beta_z \mid z \in [i_v],\ \beta_z \in v(F)\} & \text{otherwise.} \end{cases}$$

That is, we supply F with the i-attributes of $A_v \cap v(F)$ iff there exists a local visit-set of F with gv-number greater than v. In fact, these attributes will always be evaluated in the $(v+1)$-th global visit.

If $a \in A \cap A'$ then $D'_a = D_a$, else (i.e. if a is a chained s-attribute $(\alpha_y, c_1, \ldots, c_{i_v})$) D'_a is the cartesian product of D_{α_y} and i_v chains (strings) of integer locations.

Let $p\colon F_0 \to F_1 \ldots F_n \in P$ and consider a rule

$$r\colon\ a_0(F_{j_0}) = rhs\,(a_0, F_{j_0})$$

in r_p (now we assume that r_p is in n.f.). To construct the corresponding rule r' in r'_p we distinguish two cases.

Case (a). $a_0(F_{j_0}) = \alpha_x(F_0)$ for some $x \in [s_v]$. In this case the left-hand side of r' is $(\alpha_x, c_1, \ldots, c_{i_v})(F_0)$, and the right-hand side of r' is obtained from $rhs\,(a_0, F_{j_0})$ by

(i) replacing any reference to an s-attribute $\alpha_y(F_j)$, $j \geqq 1$ by

$$(*) \qquad rollup\,\big(\alpha_y(F_j), c_{z_1}(F_j), rhs(\beta_{z_1}, F_j), \ldots, c_{z_l}(F_j), rhs(\beta_{z_l}, F_j)\big),$$

where $z_1, \ldots, z_l$ are all the numbers z such that the (existing) rule in r_p defining β_z is not a simple rule. The function *rollup* is the obvious generalization of the one used in Example 2.2, allowing several chains to be rolled up at the same call.

(ii) Ignoring (i.e. replacing with marked holes) all the references to the i-attributes $\beta_z(F_0)$, $z \in [i_v]$, and adjusting the value of the chains in $(\alpha_x, c_1, \ldots, c_{i_v})(F_0)$ in an appropriate way (obvious details are omitted).

Case (b). $a_0(F_{j_0}) \neq \alpha_x(F_0)$ for any $x \in [s_v]$. In this case r' is of the form

$$r'\colon\ a_0(F_{j_0}) = rhs'(a_0, F_{j_0}),$$

where $rhs'(a_0, F_{j_0})$ is obtained from $rhs\,(a_0, F_{j_0})$ by replacing any reference to an s-attribute $\alpha_y(F_j)$ $(j \geqq 1)$ by

$$(**) \qquad rollup\,\big(\alpha_y(F_j), c_1(F_j), rhs'(\beta_1, F_j), \ldots, c_{i_v}(F_j), rhs'(\beta_{i_v}, F_j)\big).$$

(Note that all the chains are rolled up here.)

Although we construct a "corresponding" r' for each rule r in r_p (this is necessary because of the recursion in $(**)$ above), r'_p should contain only those rules r' which define correct attribute occurrences in $\mathcal{G}'$ (an occurrence $a_0(F_{j_0})$ is correct in $\mathcal{G}'$ if $a_0 \in v'(F_{j_0})$).

Proposition 4.1. $\mathcal{G}'$ is correct and it is equivalent to $\mathcal{G}$.

Proof. By the correctness of $\mathcal{G}'$ we mean that all the attribute occurrences in the semantic rules r' of $\mathcal{G}'$ are correct. The left-hand side of the rules is clearly correct by construction, so we only have to prove that the references on the right-hand sides are also correct. In Case (a) it is enough to check that the expressions $rhs\,(\beta_z, F_j)$ in $(*)$ $(z \in \{z_1, \ldots, z_l\})$ do not refer to incorrect attribute occurrences. Indeed, in this case $Gvs_p(v)$ contains (v, j) by condition (C1), hence by (C2) $rhs\,(\beta_z, F_j)$ is always correct (note that the rules of $\mathcal{G}$ are in n.f., as we assumed). In Case (b) observe that, by (C1) and (C2), if $a_0(F_{j_0})$ is correct on the left-hand side of r', then $a_0(F_{j_0})$ is computed in such a local visit of F_0 which follows the one with gv-number v.

Consequently; F_0 has β_z in $\mathscr{G}'$ for each $z \in [i_v]$ by the construction of v'. We have to consider the expressions *rhs'* (β_z, F_j) in $(**)$. Two subcases are possible.

(i) The rule defining $\beta_z(F_j)$ in r_p is a simple rule. In this case *rhs'* (β_z, F_j) is correct by the above observation.

(ii) The rule defining $B_z(F_j)$ is not a simple rule. An easy inductive argument shows that *rhs'* (β_z, F_j) contains only correct attribute occurrences.

Note that the semantic rules of $\mathscr{G}'$ are also in n.f. The rest of the proof, i.e. that $\mathscr{G}$ and $\mathscr{G}'$ are equivalent, is left to the reader. □

Proposition 4.2. $\mathscr{G}'$ is $(m-1)$-*VSE*.

Proof. Let

$$\varphi' = (A_1, \ldots, A_{v-1} \cup (A'_v \cap A'_S), A_{v+1} \cup (A'_v \cap A'_I), \ldots, A_m),$$

and for each $F \in N$ let $\pi'(F)$ be the projection of φ' to $v'(F)$. In the proof of Proposition 4.1 we observed already that only those rules refer to occurrences of i-attributes β_z $(z \in [i_v])$ in $\mathscr{G}'$ which define occurrences of attributes computed in local visits following the ones with *gv*-number v in $\mathscr{G}$. This shows that Π' is also an *smv* set of partitions, thus $\mathscr{G}'$ is *VSE* w.r.t. (Π', φ'). □

Evaluating the *AG* $\mathscr{G}'$.

We must admit that, although the reduction achieved in the visit complexity, the pure computation complexity of the *cd*-trees has increased. This is due to the fact that, while rolling up chains we have to recompute the "old" value of certain i-attribute occurrences several times. To solve the problem we use update visits introduced in Sect. 3. To this end we put the attributes β_z, $z \in [i_v]$, forward into the joint global visit-set $A'_v \cap A_{v-1}$ of φ' (and, of course, into the corresponding joint local visit-set of each nonterminal, too). However, at the call *visit* $(v-1, u)$ to a node u we do not compute the instances of these i-attributes (or just give some unimportant initial value to them), but mark them together with all the instances of the chained s-attributes belonging to A'_v at u. Then, *update* $(v-1, u)$ should be called instead of the first *rollup* call of type $(**)$ — detailed in Case (b) of the construction of the rules of $\mathscr{G}'$ — for a chained s-attribute instance α_y at node u. Note that this *update* call can always be designed as a fixed member in that global visit-sequence of the father of u which contains also the pair $(v-1, j)$ corresponding to the call *visit* $(v-1, u)$. Then, every further *rollup* call for a chained s-attribute instance at u can be replaced by a simple reference to the corresponding "old" s-attribute instance. The problem of recomputation is not solved completely, however, because we did not handle *rollup* calls of type $(*)$ in Case (a). This would require a more sophisticated marking procedure for the *update* visits, the details of which is left to the reader.

Theorem 4.3. $\mathscr{G}'$ is (generally) more efficient than $\mathscr{G}$.

Proof. By Propositions 4.1 and 4.2, $\mathscr{G}$ and $\mathscr{G}'$ are equivalent, and they are both *VSE*. Once we have eliminated recomputation, we can say that the pure computation complexity of every *cd*-tree is approximately the same in both *AG*. It would not be honest, however, to state categorically that, by Proposition 4.2, the visit complexity of the *cd*-trees in $\mathscr{G}'$ is less than that in $\mathscr{G}$. To be exact, an *update* visit is also a visit, although it concerns only one node of a derivation tree. There are extreme examples of *cd*-trees where an *update* visit must be made to every node of the tree. Such an example is illustrated in Fig. 2. Circles represent local visit-sets in the graph of the figure, and the *gv*-number of the visit-sets is written inside the circles. Members of the visit-sequences are represented as descendants of the corresponding circles. It can be seen that the visit complexity of such kind of *cd*-trees remains the same in $\mathscr{G}'$. But, taking into account all the *cd*-trees of the grammar we can in general say that $\mathscr{G}'$ is indeed better than $\mathscr{G}$ from the point of view of visit complexity. This is always the case when e.g. v is the greatest *gv*-number, like in Example 2.1.

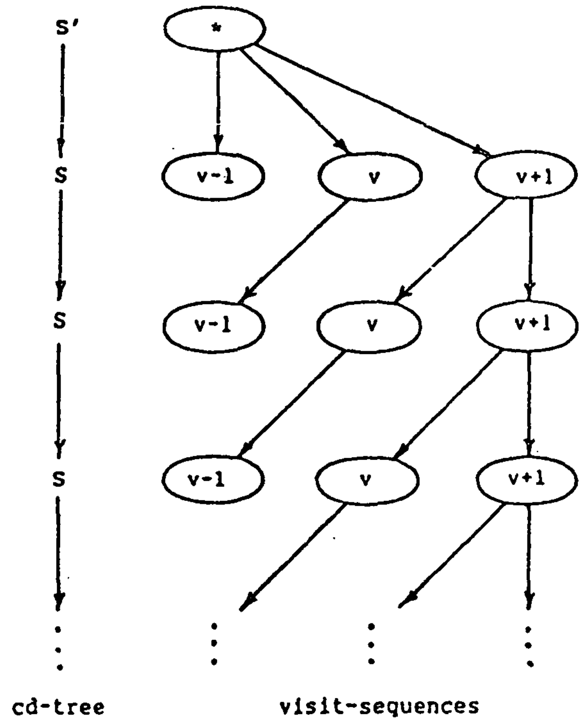

Fig. 2. An extreme example.

5. Improving the Evaluation of Linear String-valued AGs

Our second transformation technique eliminates all the *i*-attributes of $\mathscr{G}$, therefore it is more powerful than the chaining technique. This transformation can be applied, however, for a more restricted class of *AGs*, the class of linear string-valued *AGs*. Although we shall use the c.f.d. principle here with its full power, the attribute domains (as well as the algebra!) except for one attribute will not be changed.

In this section $\mathcal{G}$ will be a purely string-valued AG. By this we mean that the domain of all the attributes is Σ^*, the set of all strings over Σ, for some finite alphabet Σ, and the only operation applied in the semantic rules is concatenating strings. As usual, Σ^* denotes also the free monoid generated by Σ, thus, we can say that the attribute algebra of $\mathcal{G}$ is Σ^*. Dealing with polynomials over Σ^* we shall use different copies of the set $Z=\{z_1, \ldots, z_m, \ldots\}$ as variable symbols (e.g. $X=\{x_1, \ldots, x_m, \ldots\}$, $Y=\{y_1, \ldots, y_m, \ldots\}$). Z_m will denote the set $\{z_1, \ldots, z_m\}$. These sets of variables are assumed to be disjoint from Σ. For simplicity assume that every nonterminal, except S', has the same set of attributes consisting of k synthesized and l inherited attributes, and these attributes are numbered from 1 to k and from 1 to l, respectively. S' has only s-attributes $1, 2, \ldots, k$. (The name of the attributes is irrelevant in this section.) Now, since the right-hand side of the semantic rules are polynomials over Σ^*, and these polynomials can also be represented as strings in $(\Sigma \cup Z)^*$, the semantic rules r_p corresponding to production p: $F_0 \to F_1 \ldots F_n$ can be condensed into a sequence of $k+l\cdot n$ strings:

$$\alpha(p) \in ((\Sigma \cup X_{k\cdot n} \cup Y_l)^*)^{k+l\cdot n},$$

where each occurrence of any variable in $\alpha(p)$ corresponds to a reference to an appropriate attribute occurrence: any occurrence of $x_{k\cdot(j-1)+i}$, where $j\in[n]$ and $i\in[k]$ is a reference to the i-th s-attribute occurrence of F_j, while y_r $(r\in[l])$ corresponds to the r-th i-attribute occurrence of F_0. The first k components of $\alpha(p)$ define the s-attribute occurrences of F_0, and the following components define the i-attribute occurrences of $F_1, \ldots, F_n$ from left to right in n segments each containing l components. Observe that r_p is in n.f.

Definition 5.1. A sequence of strings $\gamma\in((\Sigma \cup Z_m)^*)^s$ (s is a nonnegative integer) is linear if each variable occurs at most once in γ. $\mathcal{G}$ is linear if $\alpha(p)$ is linear for every $p\in P$.

Let Dt (F) denote the set of all derivation trees with root $F\in N$. It is clear that — provided $\mathcal{G}$ is noncircular — the value of all the attribute instances of the nodes of a tree $t\in$ Dt (F) is uniquely determined by fixing the values of the i-attribute instances at the root of t. Let us fix these values to $y_1, \ldots, y_l$, respectively ($y_1, \ldots, y_l$ are variable symbols, as we agreed), and compute the value of the s-attribute instances at the root with the attribute domains enlarged to $(\Sigma \cup Y_l)^*$ during this computation. We obtain a sequence:

$$\beta(t) \in ((\Sigma \cup Y_l)^*)^k.$$

Clearly, $\beta(t)$ represents the sequence of polynomials that describe how the s-attributes depend on the i-attributes at the root of t. It is important to note that if $\mathcal{G}$ is linear, then it is essentially noncircular. By this we mean that, although there might be circles in the dependency graph of a derivation tree (cf. [3]), these circles are "self-contained", i.e. they do not bother the computation of the attribute instances of the root. (In other words, the attribute instances contained in the circles are always useless.) For this reason, if $\mathcal{G}$ is linear, then $\beta(t)$ always exists, moreover, it is easy to see that $\beta(t)$ is always linear.

We are able to compute the polynomials $\beta(t)$ by the help of the following algorithm.

Algorithm 5.2.

Input: a production $p\colon F_0 \to F_1 \dots F_n \in P$,
strings $\beta_j \in ((\Sigma \cup Y_l)^*)^k$ for each $j \in [n]$
$(\beta_j = \beta(t_j)$ for some hypothetical $t_j \in \mathrm{Dt}\,(F_j))$.
Output: $\beta \in ((\Sigma \cup Y_l)^*)^k$
$(\beta = \beta(t)$ for $t = p(t_1, \dots, t_n) \in \mathrm{Dt}\,(F_0))$.
Method: Suppose that

$$\alpha(p) = (a_1, \dots; a_k, b_{1,1}, \dots, b_{1,l}, \dots, b_{n,1}, \dots; b_{n,l}),$$

and set the initial value of the string variables $w_{i,j}$ and u_i $(i \in [k],\ j \in [n])$ to the i-th component of β_j and to a_i, respectively. Then apply the following procedure and set $\beta = (u_1, \dots, u_k)$.

```
procedure substitute;
begin for all $j \in [n]$ do
    for $i=1$ to $k$ do
    comment substitute $b_{j,r}$ for $y_r$ in $w_{i,j}$ for each $r \in [l]$;
    $w_{i,j} = w_{i,j}[y_r \leftarrow b_{j,r},\ r=1 \text{ to } l]$;
    repeat
        for $i=1$ to $k$ do
        $u_i = u_i[x_{k\cdot(j-1)+s} \leftarrow w_{j,s},\ j=1 \text{ to } n,\ s=1 \text{ to } k]$
        until $(u_i \in (\Sigma \cup Y_l)^*$ for every $i \in [k])$
end
```

Lemma 5.3. Let $\mathcal{G}$ be noncircular. If $\beta_j = \beta(t_j)$ for some derivation trees $t_j \in \mathrm{Dt}\,(F_j)$ $(j \in [n])$, then after the execution of Algorithm 5.2, $\beta = \beta(t)$ for $t = p(t_1, \dots, t_n)$.

Proof. Immediate from the construction. □

Further on we assume that $\mathcal{G}$ is linear. Then $\beta(t)$ can always be "splitted" by the partial mapping

$$\xi\colon ((\Sigma \cup Y_l)^*)^k \to (\Sigma^*)^{k+l} \times (Y_l \cup \{\#\})^{*(k+l+1)}$$

which we define as follows ($\#$ is a new symbol, $\Delta^{*(s)}$ denotes the set of strings over Δ not longer than s). Let $\gamma = (c_1, \dots, c_k) \in ((\Sigma \cup Y_l)^*)^k$ be linear. Put

$$\gamma_\# = \# c_1 \# c_2 \# \dots \# c_k \#,$$

and consider the symbols of $Y_l \cup \{\#\}$ occurring in $\gamma_\#$ as delimeters. Then the last component of $\xi(\gamma)$ is the string of delimeters occurring in $\gamma_\#$ read from left to right, while the first $k+l$ components of $\xi(\gamma)$ are those substrings of $\gamma_\#$ lying between the delimeters. If there are less than $k+l$ such substrings, then the remaining components of $\xi(\gamma)$ are set to λ (λ denotes the empty string). ξ is partial, since it can be applied only for linear elements. On the other hand, ξ is clearly injective, so we can speak of ξ^{-1}, the inverse of ξ.

Now we are ready to define the transformed AG $\mathcal{G}'$, which has only $k+l+1$ s-attributes, numbered again from 1 to $k+l+1$. Every nonterminal, except S' has all these attributes, S' has only the first k ones. For the sake of uniformness, however, we shall construct the semantic rules for the production $S' \to S$ in such a way that they define the "dummy" attribute occurrences $k+1, ..., k+l+1$ of S', too. The first $k+l$ attributes are the so called "derived" attributes with domain Σ^*, while the last one is the "control string" (denoted by cs) having $(Y_l \cup \{\#\})^{*(k+l+1)}$ as its domain. We define the semantic rules corresponding to a production $p: F_0 \to F_1 ... ... F_n$ in the following way. With the notation of Algorithm 5.2 set

$$\beta_j = \xi^{-1}(x_{(j-1)\cdot(k+l+1)+1}, ..., x_{(j-1)\cdot(k+l+1)+d_j-1}, \lambda, ..., \lambda, \mathrm{cs}(F_j)),$$

where $d_j = \mathit{length}(\mathrm{cs}(F_j))$, (supposing just for the moment that $X_{(k+l+1)\cdot n} \subseteq \Sigma$) and apply Algorithm 5.2. Then $\xi(\beta)$ represents the polynomials over Σ^* that define the derived attribute occurrences of F_0 in $\mathscr{G}'$, and it gives the definition of $\mathrm{cs}(F_0)$, too, in the last component.

Theorem 5.4. $\mathscr{G}$ and $\mathscr{G}'$ are equivalent.

Proof. Let $t \in \mathrm{Dt}(F)$ be any derivation tree. It can be proved by a straightforward induction on the depth of t using Lemma 5.3 that $\xi(\beta(t)) = \beta'(t)$, where $\beta'(t)$ is the sequence of values of all the attribute instances computed at the root of t in $\mathscr{G}'$. Observe that the (dummy) value of $\mathrm{cs}(S')$ is always $(\#, ..., \#)$. Hence, if $t \in \mathrm{Dt}(S')$, then $\beta(t)$ and the first k components of $\beta'(t)$ coincide. □

It is evident that the visit complexity of the *cd*-trees in $\mathscr{G}'$ is the best possible. On the other hand, since the form of the semantic rules became more complicated, one would think that the pure computation complexity has also increased in $\mathscr{G}'$. But this is not true. Indeed, let t be a *cd*-tree and suppose for simplicity that all the attribute instances of t are useful in the evaluation of t. If $\beta(t) = (w_1, ..., w_k)$ for some strings $w_i \in \Sigma^*$ $(i \in [k])$, then each w_i is the concatenation of such "atomic" strings in Σ^* that can be found on the two sides of the variable occurrences, or stand as constants on the right-hand side of the semantic rules. If w_i consists of m_i atomic strings, then $\sum_{i=1}^{k} m_i$ is a reasonable estimate for the pure computation complexity of t in $\mathscr{G}$. The same reasoning holds for $\mathscr{G}'$, too, moreover by construction, the atomic strings in $\mathscr{G}'$ are already some composites of the ones in $\mathscr{G}$. Thus, by Theorem 5.4 we obtain that the corresponding sum $\sum_{i=1}^{k} m_i'$ in $\mathscr{G}'$ is generally less than $\sum_{i=1}^{k} m_i$. The difference is compensated, however, by the extra cost of computing the control strings at each node of t.

Finally, let us mention that it is also possible to restrict the scope of our second transformation to one or more visits. For example, if $\mathscr{G}$ is *ASE* and the restriction of the semantic rules to the attribute occurrences contained in the same pass (say the m-th) is linear, then we can eliminate the i-attributes from the m-th pass and postpone their evaluation to the $(m+1)$-th pass (if any). At the same time the s-attributes of the m-th pass can be put forward to the $(m-1)$-th pass. The elaboration of a similar condition for *VSE AG* is left to the reader.

Summary

The design of efficient attribute evaluators is one of the most important requirements for compilers based on attribute grammars. Another interesting question is that, given a fixed evaluator, is it possible to optimize the attribute grammar (*AG*) to be evaluated by performing a suitable transformation on it. In this paper we present two such transformation techniques and show how they optimize evaluation by a simple visit-oriented attribute evaluator, called the *VSE* evaluator (Visit-oriented Semantic Evaluator). The class of *VSE AGs* is introduced as the visit-oriented counterpart of the class of *ASE AG*s. We also study the basic properties of *VSE AGs* relying on the strong connection between the class of *VSE AG*s and the classes of *OAG* and simple multi-visit *AG*s.

References

[1] BARTHA, M.: Linear attributed tree transformations. Acta Cybernet. 6, 125—147 (1983).
[2] BARTHA, M.: An algebraic definition of attributed transformations. Acta Cybernet. 5, 409—421 (1982).
[3] ENGELFRIET, J., FILE, G.: Simple multi-visit attribute grammars. J. Comput. System Sci. 24, 283—314 (1982).
[4] ENGELFRIET, J., FILE, G.: Passes, sweeps and visits in attribute grammars. Technische Hogeschool Twente, Onderafdeling der Informatica Memorandum Nr. INF—82—6, 1982.
[5] FARROW, R., YELLIN, D.: A comparison of storage optimizations in automatically-generated attribute evaluators. Acta Informat. 23, 393—427 (1986).
[6] FILE, G.: Attribute evaluation by recursive procedures. Theoret. Comput. Sci. 53, 25—65 (1987).
[7] GOMBÁS, E., BARTHA, M.: A multi-visit characterization of absolutely noncircular attribute grammars, Acta Cybernet. 7, 19—31 (1985).
[8] JAZAYERI, M., WALTER, K. G.: Alternating semantic evaluator. Proc. ACM Annual Conference 1975, pp. 230—234 (1975).
[9] KASTENS, U.: Lifetime analysis for attributes. Acta Informat. 24, 633—651 (1987).
[10] KASTENS, U.: Ordered attribute grammars. Acta Informat. 13, 229—256 (1980).
[11] KNUTH, D. E.: Semantics of context-free languages. Math. Systems Theory 2, 127—145 (1968); Correction: Math. Systems Theory 5, 95—96 (1971).
[12] WAITE, W. M., GOOS, G.: Compiler Construction. Berlin—Heidelberg—New York: Springer 1983.

Received June 27, 1988