On Eliminating Type 3 Circularities of Ordered Attribute Grammars

Shin Natori[1], Katsuhiko Gondow[2], Takashi Imaizumi[3], Takeshi Hagiwara[4] and Takuya Katayama[5]

1: University of Tokyo, Tokyo, Japan
natori@is.s.u-tokyo.ac.jp
2: Japan Advanced Inst. of Science and Technology, Ishikawa, Japan
gondow@jaist.ac.jp
3: Tokyo Inst. of Technology, Tokyo, Japan
image@cs.titech.ac.jp
4: Niigata University, Niigata, Japan
hagiwara@ie.niigata-u.ac.jp
5: Japan Advanced Inst. of Science and Technology, Ishikawa, Japan
katayama@jaist.ac.jp

Abstract

Ordered attribute grammars (OAGs for short) are a useful class of attribute grammars (AGs). For some attribute grammars, even though they are not circular, OAG circularity test reports that they are not ordered and fails to generate attribute evaluators because some approximation introduces circularities(called *type 3 circularities* in this paper). First we discuss that it is sometimes difficult for programmers to eliminate type 3 circularities by hand. Secondly, in order to reduce this difficulty, we propose a new AG class called OAG* that produces less type 3 circularities than OAG while preserving the positive characteristic of OAG. OAG* uses a *global dependency graph GDS* that provides a new approximation algorithm, which is different from the existing GAG and Eli/Liga systems. We also show that we obtained good results with our experimental implementation.

1. Introduction

Ordered attribute grammars (OAGs for short) introduced by Kastens in 1980 [Kas80] are a useful class of attribute grammars [Knu68][Knu71] (AGs for short), since:

- OAGs are large enough to include many practical AGs.
- The problem whether a given AG is ordered is decided in polynomial time for the size of the AG.
- OAGs can automatically generate efficient attribute evaluators.
- OAGs are well-suited for incremental attribute evaluation techniques that make it possible to generate interactive programming environments such as syntax-oriented editors [RT84].

For some attribute grammars, even though the grammars are not circular, OAG circularity test fails to generate attribute evaluators. This results from the approximation of attribute dependencies that is required in order to reduce to a polynomial time the complexity of the determination of the attribute evaluation order. If no approximation is used, the problem becomes NP-complete. In such a case, the grammar is defined as l-ordered AGs [EF82]. In the manual of the Synthesizer Generator$^{\mathrm{TM}}$ (SG for short) [Gra96], circularities due to the approximation used are called *type 3 circularities.* In this paper, we follow the SG's terminology.

Kastens stated in [Kas80] that type 3 circularities usually do not occur in practical applications such as compilers. This may be true for defining programming languages, but as we show later, it is not always true when defining programming environments using the SG. Actually, from our experience in implementing the MAGE2 editor [SK90][GISK93][HGIK97] we observed that many type 3 circularities can appear and that eliminating such circularities is a time consuming process based on programmers' trial and error. In our approach, programming environments are represented as attribute trees. They have scattered semantics since the useful information is scattered as attribute values on the attribute trees. In contrast, compilers have gathered semantics as compiled codes are stored in a root attribute. The scattered information is likely to be described as independent threads of attribute dependencies resulting in many type 3 circularities. The purpose of this paper is to introduce a technique for reducing the problem caused by type 3 circularities in OAGs.

As described in [EF82], arranged orderly AGs –that is, an AG where approximation is used and extra (virtual) dependencies are added in order to eliminate type 3 circularities– is equivalent to l-ordered AGs. This fact implies that the problem of eliminating type 3 circularities is NP-complete. Thus, instead of eliminating all type 3 circularities in (probably) exponential time, we would like to find better approximation in polynomial time. The existing Eli/Liga [Eli][Kas89] and GAG [Kas84] have an algorithm to do it in polynomial time; just after the computation of the partition for *one* symbol (i.e., before that of the partition for the next symbol), they feedback new dependencies computed from the partition as if the dependencies are direct ones (i.e. as augmenting dependencies).

We propose a new AG class OAG* that produces less type 3 circularities in a different way from the above existing systems, while preserving the good characteristics of OAG. OAG* has the following characteristics:

- The problem if an AG is OAG* is decided in polynomial time.
- If the given AG is OAG*, the decision procedure also computes the partial order of attributes that can be used to construct visit sequences. In this point, OAG* is the same as OAG.
- OAG* is a proper superset of OAG.
- In OAG*, there are less type 3 circularities than in OAG. Especially, typical type 3 circularities appearing in OAG due to the independent threads of attribute dependencies do not appear in OAG*.

The rest of the paper is organized as follows. Section 2 introduces our AG notation. In Section 3, we present the definitions of Ordered AGs, Arranged orderly AGs [Kas80], l-ordered AGs [EF82] and their properties. In Section 4, we introduce OAG* as a new AG class. The basic idea behind OAG* is to use the global dependency graph GDS to construct the extended dependency graph EDP. Section 5 gives our experimental OAG* implementation based on the Synthesizer Generator$^{\mathrm{TM}}$ [Gra96] and a good result for our MAGE2 editor [SK90][GISK93][HGIK97]. Section 6 outlines the Eli/Liga's approximation algorithm, and compares OAG* and Eli/Liga. Section 7 summarizes our conclusions and gives the directions of our future work. Section 8 is an appendix that provides an example of AG called G_5. G_5 is OAG* but not OAG; it illustrates how both methods of OAG and OAG* work for a little larger AG than the other examples given in the paper.

2. Attribute Grammars

To introduce our notation, this section provides a tuple-style definition of AGs.

2.1. Definition of AGs

An AG is defined by a 3-tuple $AG = (G, A, R)$, where G is an underlying context free grammar, A a finite set of *attributes* and R a finite set of *semantic rules*. A context free grammar is defined by a 4-tuple $G = (N, T, S, P)$, where N is a finite set of *nonterminals*, T a finite set of *terminals*, $S \in N$ a start symbol[1], and P a finite set of *production rules*.

We call a symbol X occurring in a production rule p a *symbol occurrence* which is written as $p : X_i$ where i denotes the occurring position. For example, in a production rule $p : X \to X\ Y$, $p : X_0$ and $p : X_1$ are associated with the same symbol X but they represent different symbol occurrences (subscripts are added for this distinction).

Each nonterminal is associated with two disjoint finite sets $Inh(X) \in A$ and $Syn(X) \in A$. An element of $Inh(X)$ is called an *inherited attribute*, and that of $Syn(X)$ is called a *synthesized attribute*.

For each symbol occurrence $p : X_i$ and each attribute $X.a \in Inh(X) \cup Syn(X)$, an *attribute occurrence* written as $p : X_i.a$ is associated with p. The set of all attribute occurrences associated with p is written as $AO(p)$.

A set $R(p) \in R$ of semantic rules associated with a production rule $p : X_0 \to X_1 \cdots X_n$ is defined as follows[2]:

$$R(p) = \{p : X_i.a = f(\ldots, p : X_j.b, \ldots) | ((i = 0 \wedge X_0.a \in Syn(X_0)) \vee (1 \leq i \leq n \wedge X_i.a \in Inh(X_i))) \wedge \\ ((j = 0 \wedge X_0.b \in Inh(X_0) \vee (1 \leq j \leq n \wedge X_i.b \in Syn(X_i)))\}$$

Here we say "$p : X_i.a$ depends on $p : X_j.b$", which is represented by $(p : X_j.b, p : X_i.a)$. Similarly we say "$X.a$ depends on $X.b$" by mapping the relation $(p : X_i.b, p : X_i.a)$ into the relation among attributes.

2.2. An Example of AG: G_1

Here is an example of AG called G_1. G_1's derivation trees are binary trees where the attributes X.pre_down and X.pre_up count nodes in pre-order, while X.post_down and X.post_up count nodes in post-order. Fig.1 shows an example of the attributed trees of G_1.

p1: R → X
  p1:X.pre_down = 0
  p1:X.post_down = 0
p2: X → ϵ
  p2:X.pre_up = p2:X.pre_down+1
  p2:X.post_up = p2:X.post_down+1

p3: X → X X
  p3:X_2.pre_down = p3:X_1.pre_down
  p3:X_3.pre_down = p3:X_2.pre_up+1
  p3:X_1.pre_up = p3:X_3.pre_up
  p3:X_3.post_down = p3:X_1.post_down
  p3:X_2.post_down = p3:X_3.post_up+1
  p3:X_1.post_up = p3:X_2.post_up

In this paper, it is important to distinguish between attributes (noted $X.a$) and attribute occurrences (noted $p : X_i.a$). In G_1, for example, there exist four attributes:

$$\text{X.pre\_down}, \text{X.pre\_up}, \text{X.post\_down}, \text{X.post\_up}$$

[1] We refer to both terminals and nonterminals as symbols.
[2] In this paper, we assume Bochmann normal form [Boc76] for simplicity.

and twenty attribute occurrences:

p1 : X.pre_down,	p1 : X.pre_up,	p1 : X.post_down,	p1 : X.post_up,
p2 : X.pre_down,	p2 : X.pre_up,	p2 : X.post_down,	p2 : X.post_up,
p3 : X_1.pre_down,	p3 : X_1.pre_up,	p3 : X_1.post_down,	p3 : X_1.post_up,
p3 : X_2.pre_down,	p3 : X_2.pre_up,	p3 : X_2.post_down,	p3 : X_2.post_up,
p3 : X_3.pre_down,	p3 : X_3.pre_up,	p3 : X_3.post_down,	p3 : X_3.post_up

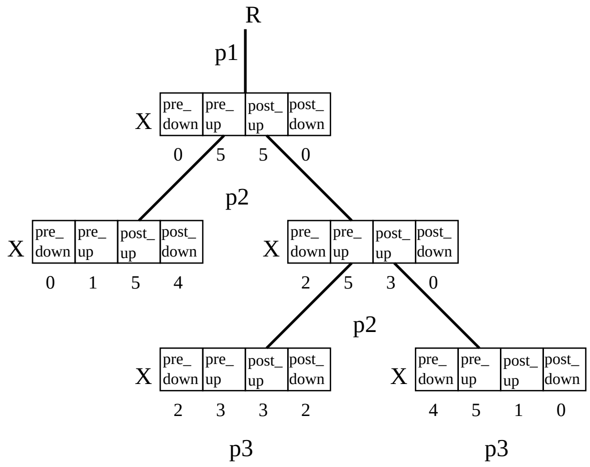


Figure 1: An example of attributed trees of G_1

3. Ordered AGs and Arranged Orderly AGs (=l-ordered AGs)

3.1. Ordered Attribute Grammars

This section gives the definition of Ordered Attribute Grammars introduced by Kastens [Kas80].

The below algorithm decides whether a given AG is ordered. In other words, the below algorithm is the definition of OAGs itself. (The definition is the same as Kastens's in [Kas80] except for some differences in the notation.)

Step 1 DP: *dependencies* among attribute occurrences associated with *production* rules

$DP = \{(p : X_i.a, p : Y_j.b) |$ an attribute occurrence $p : Y_j.b$ depends on $p : X_i.a$ in some $R(p)\}$

If DP is cyclic, G is not ordered and this algorithm is terminated. This circularity is called *type 1 circularity* [Gra96]. No AG in Bochmann normal form has type 1 circularity.

Step 2 IDP: *induced DP*

$IDP = DP \cup \{(p : X_i.a, p : X_i.b) | (p' : X_j.a, p' : X_j.b) \in IDP^+ \wedge \{p : X_i.a, p : X_i.b\} \subseteq AO(p)\}$ where IDP^+ represents the non reflexive transitive closure of IDP.

If IDP is cyclic, G is not ordered and this algorithm is terminated. This circularity is called *type 2 circularity* [Gra96].

Step 3 IDS: *induced* dependencies among attributes of *symbols*

$$IDS = \{(X.a, X.b) | (p : X_i.a, p : X_i.b) \in IDP^+\}$$

Step 4 $A_{X,n}$: a disjoint partition of $Inh(X) \cup Syn(X)$

$$A_{X,1} = Syn(X) - \{X.a | (X.a, X.b) \in IDS^+\}$$
$$A_{X,2n} = \{X.a | X.a \in Inh(X) \wedge (\forall X.b : (X.a, X.b) \in IDS^+ \Rightarrow \exists m < 2n : X.b \in A_{X,m})\} - \bigcup_{k=1}^{2n-1} A_{X,k}$$
$$A_{X,2n+1} = \{X.a | X.a \in Syn(X) \wedge (\forall X.b : (X.a, X.b) \in IDS^+ \Rightarrow \exists m < 2n+1 : X.b \in A_{X,m})\} - \bigcup_{k=1}^{2n} A_{X,k}$$

Step 5 DS: a completion of IDS

$$DS = IDS \cup \bigcup_{X \in N} \{(X.a, X.b) | X.a \in A_{X,k} \wedge X.b \in A_{X,k-1}\}$$

Intuitively, DS expands IDS to satisfy the following relation.

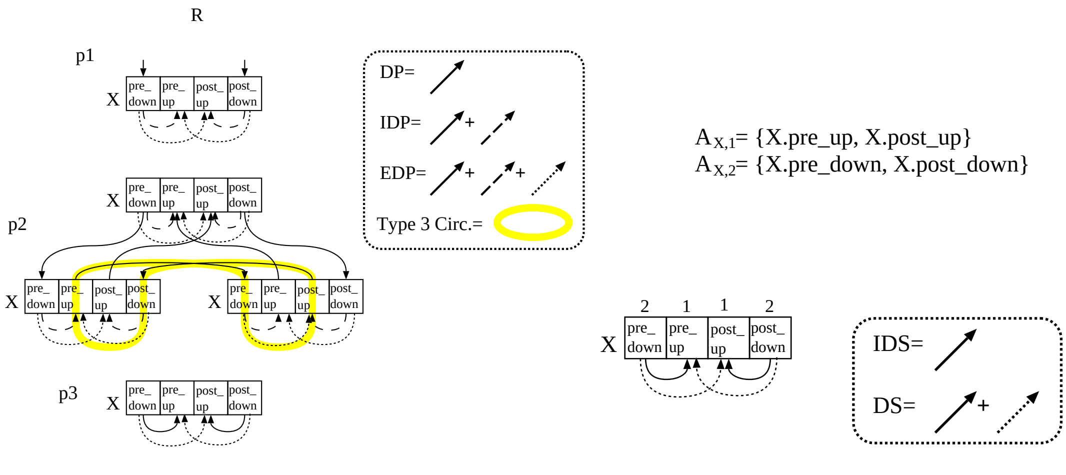
$$\forall X \in N, \forall a \in Inh(X), \forall b \in Syn(X) : (X.a, X.b) \in DS^+ \vee (X.b, X.a) \in DS^+$$

Step 6 EDP: *extended* DP (with DS)

$$EDP = DP \cup \{(p : X_i.a, p : X_i.b) | (X.a, X.b) \in DS \wedge \{p : X_i.a, p : X_i.b\} \subseteq AO(p)\}$$

If EDP is cyclic, G is not ordered. This circularity is called *type 3 circularity* [Gra96]. If EDP is cyclic free, G is ordered.

For example, Fig. 2 shows IDS, DS, DP, IDP and EDP of the AG G_1 defined in Sect. 2.2. In Fig. 2(a), EDP is cyclic, which means G_1 has a type 3 circularity, so for OAG, G_1 is not ordered.



(a) DP, IDP and EDP of G_1 (b) IDS and DS of G_1

Figure 2: DP, IDP, EDP, IDS and DS of G_1

3.2. Arranged Orderly Attribute Grammars

It is often possible to eliminate type 3 circularities from an AG by adding extra (virtual) dependencies [Kas80]. AGs whose type 3 circularities can be eliminated by adding such extra dependencies are called *arranged orderly*. One way of doing this consists of using a conditional expression as follows:

$$\begin{array}{rcl} p : X.a & = & p : Y.b \\ & & \downarrow \\ p : X.a & = & \mathbf{if}(\mathbf{true},\ p : Y.b,\ p : Z.c) \end{array}$$

In the above, an extra dependency $(p : Z.c, p : X.a)$ is added while retaining the G_1's original meaning [Gra96].

More precisely, an AG is arranged orderly if there exists $ADS = \{(X.a, X.b) | \{X.a, X.b\} \subseteq A\}$ such that EDP computed from $DP \cup \{(p : X_i.a, p : X_i.b) | (X.a, X.b) \in ADS \wedge \{p : X_i.a, p : X_i.b\} \subseteq AO(p)\}$ is cyclic free. ADS is called *augmenting dependencies* [Kas80].

For example, G_1 is arranged orderly by an extra dependency (X.post_up, X.pre_down). To add the dependency into G_1, one may change the semantic rule as follows:

$$\begin{array}{rcl} \mathrm{p2} : \mathrm{X}_2.\mathrm{pre\_down} & = & \mathrm{p2} : \mathrm{X}_1.\mathrm{pre\_down} \\ & & \downarrow \\ \mathrm{p2} : \mathrm{X}_2.\mathrm{pre\_down} & = & \mathbf{if}(\mathbf{true},\ \mathrm{p2} : \mathrm{X}_1.\mathrm{pre\_down},\ \mathrm{p2} : \mathrm{X}_2.\mathrm{post\_up}) \end{array}$$

which introduces an extra dependency (p2 : X_2.post_up, p2 : X_2.pre_down). Note that there are other ways to introduce the dependency (X.post_up, X.pre_down) into G_1 such as (p1 : X.post_up, p1 : X.pre_down) and (p2 : X_3.post_up, p2 : X_3.pre_down). Furthermore, there are six possible ADSs in all that can be used to eliminate the type 3 circularity:

$$\begin{array}{l} \{(\mathrm{X.post\_up}, \mathrm{X.pre\_down})\} \\ \{(\mathrm{X.pre\_down}, \mathrm{X.post\_up})\} \\ \{(\mathrm{X.post\_down}, \mathrm{X.pre\_up})\} \\ \{(\mathrm{X.pre\_up}, \mathrm{X.post\_down})\} \\ \{(\mathrm{X.post\_up}, \mathrm{X.pre\_down}), (\mathrm{X.post\_down}, \mathrm{X.pre\_up})\} \\ \{(\mathrm{X.pre\_down}, \mathrm{X.post\_up}), (\mathrm{X.pre\_up}, \mathrm{X.post\_down})\} \end{array}$$

Thus, it is easy to find an appropriate augmenting dependency to eliminate a type 3 circularity from G_1. However, there are many cases where it is difficult (or even impossible in some cases (ex. G_2 in Fig. 3)) to do so.

3.3. Properties

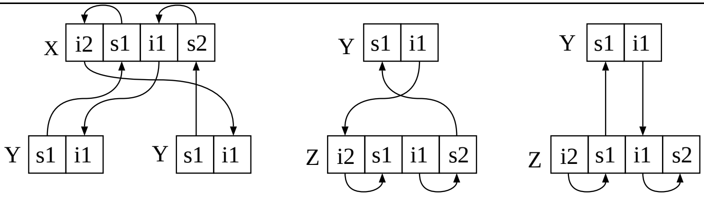


Figure 3: G_2 has a type 3 circularity, but is not arranged orderly(=l-ordered) .

1. Some type 3 circularities can not be eliminated.

   G_2 shown in Fig. 3 does not have any type 1 or 2 circularities but has a type 3 circularity. We show that G_2 is not arranged orderly as follows.

   All candidates of augmenting dependencies introduce the following dependencies into DS:

   $$(X.s1, X.i1) \text{ or } (X.i1, X.s1),\ (Y.s1, Y.i1) \text{ or } (Y.i1, Y.s1),\ (Z.s1, Z.i1) \text{ or } (Z.i1, Z.s1)$$

   since the fact $\forall X.i \in Inh(X), \forall X.s \in Syn(X) : (X.i, X.s) \in DS \vee (X.s, X.i) \in DS$ holds. Any combination of $2^3 = 8$ possibilities makes EDP computed from $DP \cup ADS$ cyclic.

2. l-ordered AGs = Arranged orderly AGs

   J. Engelfriet et al. introduced l-ordered AGs and stated l-ordered AGs = Arranged orderly AGs in [EF82].

   l-ordered AGs have similar properties to OAGs except that the problem whether an AG is l-ordered is NP-complete whereas OAGs can be tested in polynomial time. An AG G is l-ordered iff there exists a set of total orders $\{LO(X) | X \in N\}$ such that for each $LO(X)$, $LO(X)$ is a total order on $Syn(X) \cup Inh(X)$ and

   $$DP \cup \{(p : X_i.a, p : X_i.b) | (X.a, X.b) \in \bigcup_{X \in N} LO(X) \wedge \{p : X_i.a, p : X_i.b\} \subseteq AO(p)\}$$

   is cyclic free. If an AG is l-ordered, the attribute instances of a node can always be evaluated in the order $LO(X)$.

3. There are serious situations where it is difficult to eliminate type 3 circularities.

   Since Arranged orderly AGs = l-ordered AGs [EF82], the problem of eliminating type 3 circularities is NP-complete. Also, during the development of the MAGE2 editor [SK90][GISK93][HGIK97] we had to try many combinations of augmenting dependencies to eliminate all type 3 circularities (see Sect. 5). Considering these two points, we reached the conclusion that we need some mechanisms that relieve us from the problem of eliminating type 3 circularities. As a solution to this problem, we propose OAG* in Sect. 4.

4. OAG*

In this section, we introduce OAG* as a new class of AGs that makes it easier to deal with type 3 circularities. The basic idea of OAG* is relatively simple; OAG* uses a global dependency graph GDS as a good hint to avoid type 3 circularities in polynomial time. GDS projects all dependencies among attribute occurrences in DP into those among attributes. Thus, GDS includes all possible attribute dependencies. First, we give the definition of OAG*. Then, we show the desirable properties of OAG*.

- OAG* retains the positive characteristics of OAGs.

  Especially, the problem of determining if an AG is OAG* is computed in polynomial time.

- l-ordered(=arranged orderly AGs)$\supset$ OAG* $\supset$ OAG.

  OAG* produces less type 3 circularities than OAG. OAG* does not produce the typical type 3 circularities due to the independent threads of attribute dependencies that appear in OAGs.

  For example, G_3 and G_4 in Fig.4 as well as G_1 have independent threads of attribute dependencies, which, for OAGs, result in type 3 circularities. By contrast, OAG* circularity test does not produce type 3 circularities for G_1, G_3 and G_4.

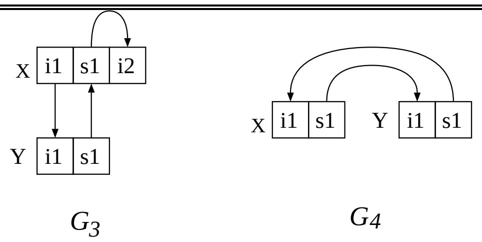


Figure 4: $G3, G4$: simple AGs that have a type 3 circularity

4.1. Definition

The definition of OAG* is given by the following algorithm that decides whether a given AG G is OAG* or not.

Step 1 DP, IDP, IDS, DS

The computing method of DP, IDP, IDS and DS is exactly the same as that of OAG. If it results in type 1 or 2 circularity, G is not OAG*.

Step 2 GDS: global attribute dependencies

$$GDS = \{(X.a, Y.b) | (X_i.a, Y_j.b) \in DP\}$$

Step 3 GA_k: a global attribute partition

GDS is decomposed into its strongly-connected components and then topologically-sorted into $GA_1, \ldots, GA_k$ to satisfy the following relation:

$$\begin{array}{l}
\forall X.a, \forall X.b : \\
((X.a, X.b) \in GDS^+ \wedge (X.b, X.a) \in GDS^+ \Rightarrow \exists m : \{X.a, X.b\} \subseteq GA_m) \\
\wedge((X.a, X.b) \in GDS^+ \wedge (X.b, X.a) \notin GDS^+ \Rightarrow \exists m, \exists n : m < n \wedge X.a \in GA_m \wedge X.b \in GA_n) \\
\wedge((X.a, X.b) \notin GDS^+ \wedge (X.b, X.a) \notin GDS^+ \Rightarrow \exists m, \exists n : m \neq n \wedge X.a \in GA_m \wedge X.b \in GA_n)
\end{array}$$

Step 3.1 initializing; $Finished := \phi, i := 1$

Step 3.2 computing an i-th partition GA_i

$$GA_i = \{X.b | (X.a, X.b) \in GDS^+ \Rightarrow (X.a \in Finished \cup S(X.b, GDS)\}$$

where $S(X.b, GDS)$ means the set of strongly-connected components with $X.b$ in GDS.

Step 3.3 $Finished := Finished \cup GA_i, i := i + 1$

Go to step 3.2 until $Finished = A$.

Note that for a given GDS, global attribute partition GA_k is not determined uniquely. But this ambiguity does not affect the result whether a given AG is OAG* or not (see Theorem 4.5).

Step 4 DS': a completion of IDS with GDS and DS

DS' is constructed:

- by adding dependencies $(X.a, X.b) \in DS$ if $X.a$ and $X.b$ are strongly-connected in GDS,
- otherwise, by adding dependencies in GDS.

$$\begin{aligned} DS' \;=\; & \{(X.a, X.b) | (\{X.a, X.b\} \subseteq GA_m \wedge (X.a, X.b) \in DS^+) \\ & \wedge (m < n \wedge X.a \in GA_m \wedge X.b \in GA_n \wedge \\ & (X.a \in Inh(X) \wedge X.b \in Syn(X) \vee X.a \in Syn(X) \wedge X.b \in Inh(X)))\} \end{aligned}$$

Step 5 EDP': extended DP (with DS')

$$EDP' = DP + \{(p : X_i.a, p : X_i.b) | (X.a, X.b) \in DS' \wedge \{p : X_i.a, p : X_i.b\} \subseteq AO(p)\}$$

If EDP' is cyclic, G is not OAG*. We also call this (OAG*'s) type 3 circularities. If EDP' is cyclic free, G is OAG*.

If G is OAG*, then an actual partition $A'_{X,k}$ used for constructing visit-sequences is computed from DS' (just in the same way as computing OAG's partition $A_{X,k}$ from IDS):

$$\begin{aligned} & A'_{X,1} = Syn(X) - \{X.a | (X.a, X.b) \in DS'^+\} \\ & A'_{X,2n} = \{X.a | X.a \in Inh(X) \wedge (\forall X.b : (X.a, X.b) \in DS'^+ \Rightarrow \exists m < 2n : X.b \in A'_{X,m})\} - \textstyle\bigcup_{k=1}^{2n-1} A'_{X,k} \\ & A'_{X,2n+1} = \{X.a | X.a \in Syn(X) \wedge (\forall X.b : (X.a, X.b) \in DS'^+ \Rightarrow \exists m < 2n+1 : X.b \in A'_{X,m})\} - \textstyle\bigcup_{k=1}^{2n} A'_{X,k} \end{aligned}$$

4.2. OAG* Examples

G_1 given in Sect. 2.2 is an example of OAG* but not of OAG, since EDP of G_1 is cyclic as shown in Fig. 2, but EDP' of G_1 is cyclic free as shown in Fig. 5. Fig. 5 shows GDS, GA, DS' and EDP' of G_1.

G_3 and G_4 given in Fig. 4 are also examples of OAG* but not of OAG. G_6 (Fig.6) is an example of l-ordered AGs (=arranged orderly AGs) but not of OAG*.

4.3. Basic Idea of OAG*

The key idea of OAG* is the global dependency graph GDS, which is useful since OAG* produce less type 3 circularities in polynomial time than OAG does. This section provides the basic idea of OAG*.

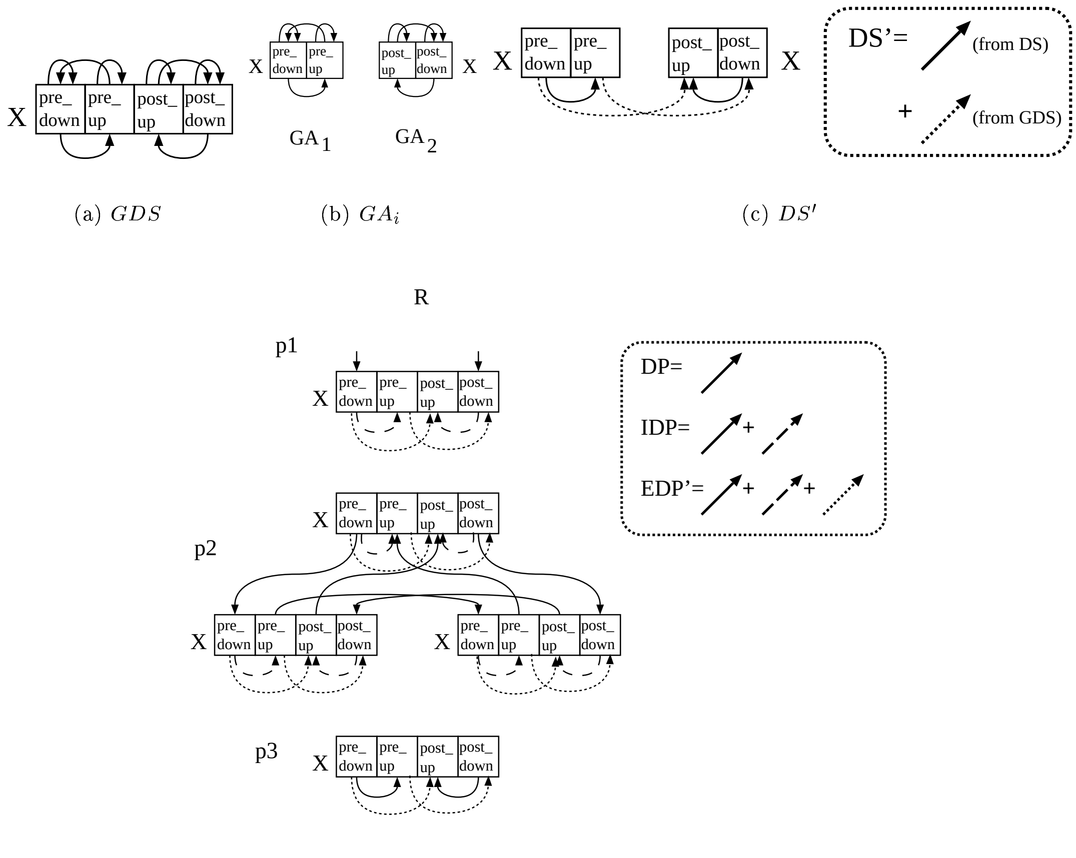
If $(X.b, X.a) \notin GDS^+$ then we can evaluate $X.a$ first and then $X.b$ in any context, whether $(X.a, X.b) \in GDS^+$ or $(X.a, X.b) \notin GDS^+$, because $(X.b, X.a) \notin GDS^+$ implies there is no dependency path from $p : X_i.b$ to $p : X_i.a$ in any attributed trees by the definition of GDS. Thus, $(X.a, X.b)$ from GDS is a *correct* order in the sense that $(X.a, X.b)$ introduces no type 3 circularities. If GDS has no strongly-connected components, we can obtain DS' by topologically-sorting GDS. Unfortunately, GDS has strongly-connected components in general. Therefore, we use DS instead of GDS for strongly-connected components in GDS as the second best approximation of attribute dependencies.

Note that we can not accept full-searching of all possibilities here since the problem becomes NP-complete. And also note that attribute dependencies from DS may be *incorrect* since they may introduce a type 3 circularity into EDP'.

To summarize the above, OAG* first tries to use as many correct dependencies from GDS as possible, and then, for strongly-connected components in GDS, OAG* reluctantly use (possibly incorrect) dependencies from DS instead of GDS.

4.4. OAG* Properties

Lemma 4.1 $(p : X_i.a, p' : X_j.b) \in EDP' \Rightarrow m \leq n \wedge X.a \in GA_m \wedge X.b \in GA_n$.



(a) GDS (b) GA_i (c) DS'

(d) EDP'

Figure 5: GDS, GA, DS' and EDP' of G_1

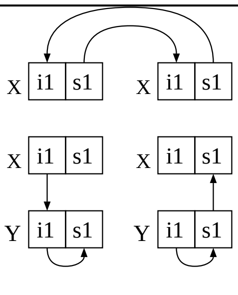


Figure 6: G_6: an example of l-ordered AGs (=arranged orderly AGs) but not OAG*

Proof $(p : X_i.a, p' : X_j.b) \in EDP'$ implies $(p : X_i.a, p' : X_j.b) \in DP$ or $(X.a, X.b) \in DS'$ by Def. of EDP'. If $(p : X_i.a, p' : X_j.b) \in DP$ then $(X.a, X.b) \in GDS$ (by Def. of GDS) and $m \leq n \wedge X.a \in GA_m \wedge X.b \in GA_n$ (by Def. of GA_i). If $(X.a, X.b) \in DS'$ then $m \leq n \wedge X.a \in GA_m \wedge X.b \in GA_n$ (by Def. of DS'). Thus Lemma 4.1 holds.

□

Theorem 4.2 All attributes whose occurrences are cyclic in EDP' (i.e. type 3 circularity) are included in the same global partition GA_m. Furthermore, they are also cyclic (i.e. strongly-connected components) in GDS.

Proof Let $p : X_i.a$ and $p' : X_j.b$ be in the type 3 circularity, that is, $(p : X_i.a, p' : X_j.b) \in EDP'^+ \wedge (p' : X_j.b, p : X_i.a) \in EDP'^+$. From Lemma 4.1, both $m \leq n \wedge X.a \in GA_m \wedge X.b \in GA_n$ and $n \leq m \wedge X.a \in GA_n \wedge X.b \in GA_m$ hold. This implies $m = n \wedge \{X.a, X.b\} \in GA_m$. So the first half holds. All strongly-connected components are introduced from circularities in GDS by Def. of GA_k as topological sorting introduces no circularities. Thus $X.a$ and $X.b$ are clearly strongly-connected components in GDS. So the second half holds and Theorem 4.2 holds.

□

Theorem 4.3 l-ordered AG (=arranged-orderly AG) $\supset$ OAG* $\supset$ OAG.

Proof (l-ordered AG$\supset$ OAG*) If a given AG is OAG*, there exists DS' where $DP + DS'$ is cyclic free. Let $LO(X)$ be obtained by topologically-sorting $\{(X.a, X.b) | (X.a, X.b) \in DS'\}$ for all $X \in N$.

Then, by Def. of OAG*, $DP \cup \{(p : X_i.a, p : X_i.b) | (X.a, X.b) \in \bigcup_{X \in N} LO(X)\}$ is cyclic free. This shows the given AG is l-ordered. Here OAG* is a proper subset of l-ordered AGs since G_6 as defined in Fig. 6 is l-ordered but not OAG*.

(OAG* $\supset$ OAG) First, if OAG* has a type 1 or 2 circularity then OAG also has a type 1 or 2 circularity, since the OAG* method of computing DP and IDP is exactly the same as OAG.

Next, we show that if OAG* has a type 3 circularity then OAG also has a type 3 circularity. All attribute occurrences in a type 3 circularity belong to the same partition G_m by Theorem 4.2. Let DS_{G_m} be $\{(p : X_i.a, p : X_i.b) | (X.a, X.b) \in DS^+ \wedge \{X.a, X.b\} \subset G_m\}$. Here $DP + DS_{G_m}$ is cyclic, since G_m causes the type 3 circularity. $EDP = DP + DS$ is also cyclic, because $DS_{G_m} \subseteq DS$ holds. This shows OAG has a type 3 circularity if OAG* has a type 3 circularity. Thus, if a given AG G is not OAG*, then G is also not OAG. This shows OAG* $\supset$ OAG. Here OAG is a proper subset of OAG* since G_1 is OAG* but not OAG.

□

Theorem 4.4 The problem whether an AG is OAG* is decided in polynomial time.

Proof

Step 1 DP, IDP, IDS and DS is computed in polynomial time as shown in [Kas80].

Step 2 GDS global attribute dependency graph is computed in $O(|E_{DP}|)$ where E_{DP} is the set of edges in DP.

Step 3 GA_k global attribute partition is computed in $O(|A|^2)$ because all strongly-connected components are computed in $O(|E_{GDS}| + |V_{GDS}|)$ by using Tarjan's algorithm [Tar72], and $|E_{GDS}| + |V_{GDS}| \leq |A|^2 + |A|$.

Step 4 DS' is computed in $O(|A|^4)$.
The dominant cost of computing DS' is testing if $e \in DS^+$ or $e \notin DS^+$ for each $e \in A \times A$, which is computed in $O(|A|^2)$. Thus, for all $e \in A \times A$, it takes $O(|A|^2) \times O(|A|^2) = O(|A|^4)$.

Step 5 The cost of constructing EDP' is $O(|E_{DP}| + |SO||A|^2)$ because the size of $\{(p : X_i.a, p : X_i.b) | (X.a, X.b) \in DS' \wedge \{p : X_i.a, p : X_i.b\} \subseteq AO(p)\}$ is $O(|SO||A|^2)$, where SO is the set of symbol occurrences.
The cost of testing if EDP' is cyclic is $O(|E_{EDP'}| + |V_{EDP'}|)$. Here, we have $|E_{EDP'}| + |V_{EDP'}| \leq |AO|^2 + |AO|$, where AO is the set of attribute occurrences.

Clearly, the above steps 1 through 5 show that OAG* is computed in polynomial time for the size of a given AG.

□

Theorem 4.5 OAG* step 3 may create several different global partitions because of topological sorting, but all of them are the same in the sense that the difference among them does not affect whether an AG is OAG* or not.

Proof It is sufficient to show that if EDP' for a partition $G_i (1 \leq i \leq k)$ has a type 3 circularity, then any other partition $G'_i (1 \leq i \leq k)$ also causes type 3 circularities.

Let G_m be the partition including all attribute occurrences in the type 3 circularity for the partition $G_i (1 \leq i \leq k)$. Then $DP + G_m$ is cyclic, since EDP' under the partition $G_i (1 \leq i \leq k)$ has a type 3 circularity. And there exists m such that $G_m = G'_{m'}$, since topological sorting does not change strongly-connected components. Therefore $DP + G'_{m'}$ is also cyclic. This means the partition $G'_i (1 \leq i \leq k)$ also causes a type 3 circularity. Thus Theorem 4.5 holds.

□

Theorem 4.6 If two paths of attribute dependencies are not strongly-connected in GDS and each of them causes no type 3 circularity, OAG* does not produce type 3 circularities due to the two paths.

Proof As shown in Theorem 4.2, all attribute occurrences in any type 3 circularity is also cyclic in GDS. The two paths are not strongly-connected components, and topological sorting does not introduce new strongly-connected components, so if each path has no type 3 circularity, the whole of two paths has also no type 3 circularity. Thus Theorem 4.6 holds.

□

Theorem 4.6 shows that OAG* does not produce type 3 circularities due to independent threads of attribute dependencies if they are not cyclic in GDS such as for G_1, G_3 and G_4. For an intuitive understanding of Theorem 4.6, let's consider this theorem in the case of G_4. As the result of topological sorting of G_4's GDS, one of the following sets is added into DS' by Def. of GA_i and DS'.

$$\{(X.s1, X.i1), (Y.s1, Y.i1)\}$$
$$\{(X.s1, X.i1), (Y.i1, Y.s1)\}$$
$$\{(X.i1, X.s1), (Y.s1, Y.i1)\}$$

But $\{(X.i1, X.s1), (Y.i1, Y.s1)\}$ is not added into DS', because

$$\{(X.i1, X.s1), (Y.i1, Y.s1)\} \cup GDS (= \{(X.s1, Y.i1), (Y.s1, X.i1)\}$$

is cyclic and topological sorting never produces a new cycle. This is the reason why OAG* does not introduce a type 3 circularity for G_4.

Theorem 4.3 and Theorem 4.6 demonstrate that OAG* produce less type 3 circularities than OAG.

5. An Experimental Implementation

We implemented OAG* experimentally on the Synthesizer Generator™ 4.2 (SG for short) [3], which uses incremental attribute evaluation scheme based on OAGs and we obtained good results (see also Table. 1).

1. G_1, G_3, G_4 and G_5 (all of them are OAG* but not OAG)

   The SG with OAG* accepted all of them and generated attribute evaluators successfully, while the original SG based on OAG reported type 3 circularities for them.

2. G_2 (not l-ordered) and G_6 (l-ordered but not OAG*)

   The SG with OAG* failed to generate attribute evaluators and reported type 3 circularities.

3. the MAGE2 editor

   Using the SG, we developed the MAGE2 editor for Object Oriented Attribute Grammars (OOAG) [SK90] [GISK93][HGIK97]. The MAGE2 editor includes a compiler and a static error checking including type checking, and detecting used but not declared variables. It also provides means for communication with other tools such as objects and class browsers.

   The attribute grammar specifying the MAGE2 editor is relatively large; $|N| = 57, |P| = 141, |A| = 729$ and 9193 lines in SSL[4]. We were troubled with eliminating many type 3 circularities when developing the MAGE2 editor, although the SG is very useful to develop such tools.

   55 augmenting dependencies were added to eliminate all type 3 circularities (some of them may not be necessary); without them the original SG reported 10 type 3 circularities.

   The SG with OAG* accepted it without 55 augmenting dependencies and generated an attribute evaluator successfully. This shows that OAG* is useful in this case to eliminate type 3 circularities.

6. Related Works – Eli/Liga system –

GAG system [Kas84] and Liga system integrated in Eli [Eli][Kas89] (Eli/Liga for short) have an algorithm to avoid type 3 circularities. Eli/Liga applies the same algorithm as GAG does on abstraction level. So we refer only to the Eli/Liga's algorithm in the rest of this section. The algorithm is different from ours.

This section compares OAG* and Eli/Liga system. First, the outline of Eli/Liga's algorithm is given. Then, we compare OAG* and Eli/Liga using the examples G_1 through G_8 (Table. 1). From the results, two AG classes defined by OAG* and Eli/Liga are not in the inclusion order each other. Needless to say, they include OAG properly. Both algorithms seem to be effective with some differences, although it requires further research to identify both properties.

6.1. Eli/Liga's Algorithm

This section outlines the Eli/Liga's algorithm to avoid type 3 circularities. Please note the algorithm is given here by partially using mathematical notation, since we would like to focus briefly on how to

[3]Since version 5.0, the Synthesizer Generator is shipped without its source code. Therefore we do not use the SG now for this purpose.

[4]SSL is the specification language of the SG.

"feedback" a partition to IDS. The actual Eli/Liga algorithm is defined as fully iterative and more efficient one.

Roughly speaking, every time Eli/Liga computes the partition $A_{X,k}$ for *one* symbol X, Eli/Liga adds new edges computed from the partition as augmenting dependencies, then performs a closure computation again like Step 2 and 3 of Sect.3.

To be more precise, we need to use two variables $TDP(p)$ and $TDS(X)$ after the Sec. 7 of [Kas80]. We abbreviate $\bigcup_{p\in P} TDP(p)$ and $\bigcup_{X\in V} TDS(X)$ as TDP and TDS, respectively.

- $TDP(p)$: a dependency graph variable for rule p
- $TDS(X)$: a dependency graph variable for symbol X

We also introduces two procedures Propagate() and Partition(X).

procedure Propagate() { // *a closure computation*
  loop forall $p \in P$ **do**
    $TDP(p) := TDP(p) \cup \{(p : X_i.a, p : X_i.b) | ((p' : X_j.a, p' : X_j.b) \in TDP^+$
      $\vee (X.a, X.b) \in TDS(X)) \wedge \{p : X_i.a, p : X_i.b\} \subseteq AO(p)\};$ **od**
  loop forall $X \in V$ **do**
    $TDS(X) := \{(X.a, X.b) | (p : X_i.a, p : X_i.b) \in TDP\};$ **od**
}
procedure Partition(X) { // *partitioning and updating* $TDS(X)$
  Computing a partition $A_{X,k}$ *using* $TDS(X)$ *just in the same way described in Step 4 of Sect.3*;
  $TDS(X) := TDS(X) \cup \{(X.a, X.b) | \exists m, \exists n : m > n \wedge X.a \in A_{X,m} \wedge X.b \in A_{X,n}\};$
}

Using the above, we can now describe both algorithms: the original OAG's and the Eli/Liga's. We assume here $DP(p)$ includes all direct dependencies in p.

procedure OAG() {
  loop forall $p \in P$ **do** $TDP(p) := DP(p)$; **od**
  loop forall $X \in V$ **do** $TDS(X) := \{\}$; **od**
  Propagate();
  loop forall $X \in V$ **do** Partition(X); **od** // *partitioning independently among symbols*
  Propagate(); // *to check for type 3 circularities*
}

procedure EliLiga() {
  loop forall $p \in P$ **do** $TDP(p) := DP(p)$; **od**
  loop forall $X \in V$ **do** $TDS(X) := \{\}$; **od**
  Propagate();
  loop forall $X \in V$ **do**
    Partition(X); // (1) *partitioning one symbol* and feedbacking the partition into $TDS(X)$
    Propagate(); **od** // (2) *then performing the closure computation*
}

To borrow Kastens's words, the above algorithm EliLiga() is summarized as follows:

> As a consequence of the above (1) and (2), further dependencies may be added to $TDS(Y)$ for some other $Y \in N$. They influence the partitioning decisions made for the next symbols, and avoid conflicts which may have occurred if the partitions were computed independently.

In the case of G_4 (Fig. 7), for example, Eli/Liga's algorithm feedbacks $\{(X.i1, X.i2)\}$ as an augmenting dependency to $TDS(X)$ before the partition of Y is computed. As a consequence, $(Y.s1, Y.i1)$ is added to $TDS(Y)$. Thus the type 3 circularity is avoided.

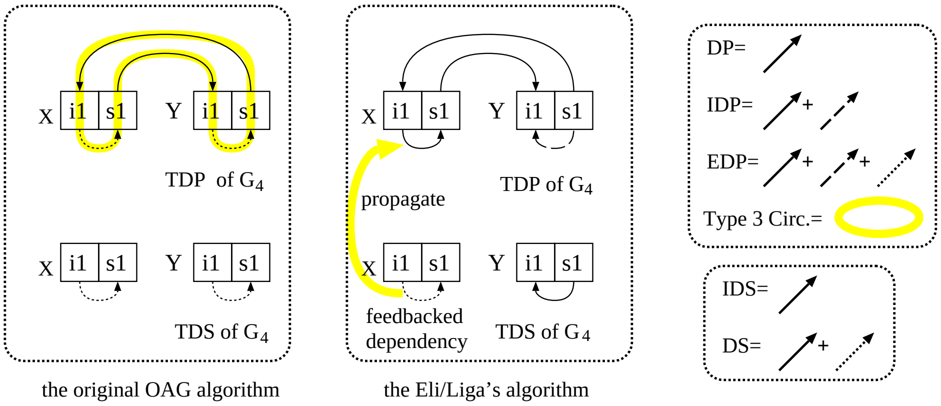


Figure 7: an example of feedback in G_4 by the Eli/Liga's algorithm

6.2. Comparing Eli/Liga and OAG*

Table 1 compares OAG* and Eli/Liga using the example AGs G_1 through G_8. G_7 (Fig.8) is not OAG* but Eli/Liga accepts, while G_8 (Fig.9) is OAG* but Eli/Liga does not accept. Thus G_7 and G_8 shows that two AG classes defined by OAG* and Eli/Liga are not in the inclusion relation each other. Of course, both of the classes include OAG properly.

If a given AG has so many semantic rules that all attributes in GDS are strongly-connected (ex. G_7), OAG* may not be useful. If there is no appropriate augment dependency that can be feedbacked (ex. G_8), Eli/Liga may not be useful, although we do not know how often such situations happen in practical applications (especially that have scattered-semantics).

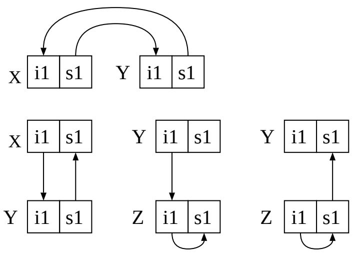


Figure 8: G_7: an example that is not OAG* but Eli/Liga accepts

7. Conclusion

In this paper, we proposed OAG* a new class of AG to reduce the difficulty of eliminating type 3 circularities. To eliminate type 3 circularities, OAG* has a different algorithm from the existing

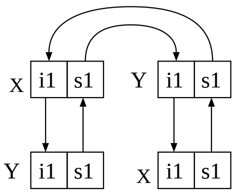


Figure 9: G_8: an example that is OAG* but Eli/Liga does not accept

	G_3, G_4, G_5	G_1, G_8	G_7	G_6	G_2
l-ordered	A	A	A	A	NA
OAG*	A	A	NA	NA	NA
Eli/Liga	A	NA	A	NA	NA
OAG	NA	NA	NA	NA	NA

A=accepted, NA=not accepted

Table 1: Comparative Table: OAG* and Eli/Liga

systems: GAG [Kas84] and Eli/Liga system [Eli][Kas89]. We showed that OAG* has several attractive characteristics:

- The problem if an AG is OAG* is decided in polynomial time.
- l-ordered(=arranged orderly AG) $\supset$ OAG* $\supset$ OAG
- OAG* produces less type 3 circularities than OAG. Especially, OAG* does not produce type 3 circularities for independent threads of attribute dependencies that are not cyclic in GDS, which are typical type 3 circularities in OAG.

We obtained a good result for our MAGE2 editor on our experimental OAG* implementation based on the Synthesizer Generator^{TM} [Gra96]. This result showed that OAG* is useful. We compared OAG* and Eli/Liga using the examples given in the paper. From the results, two AG classes defined by OAG* and Eli/Liga are not in the inclusion order each other. Needless to say, they include OAG properly. Both algorithms seem to be effective with some differences, although it requires further research to identify both properties.

One of our future works is to study a more precise classification of attribute grammars; especially we would like to know the relationship between OAG* and OAG(i)(page 16 of [DJL88]), although we expect that OAG* is one of OAG(i). Another important work is to research how efficient OAG*, Eli/Liga system and l-ordered AGs are in pragmatic use. We could possibly use l-ordered AG scheme to reduce programmer's burden even if the problem of determining if an AG is l-ordered is NP-complete.

Acknowledgements

The authors would like to thank Adel Cherif for reading the draft and making a number of helpful suggestions. The authors would also like to thank anonymous referees for valuable comments that have improved this paper. Finally, my special thanks are due to Uwe Kastens for helpful explanation and valuable comments about the algorithm used in the Eli/Liga system and for permission to quote from his e-mail.

Bibliography

[Boc76] G.V. Bochmann. Semantic evaluation from left to right. *CACM*, 19(2):55–62, 1976.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems, and Bibliography*, volume 323 of *Lec. Notes in Comp. Sci.* Springer-Verlag, 1988.

[EF82] Joost Engelfriet and Gilberto File. Simple multi-visit attribute grammars. *Journal of Computer and System Sciences*, 24(3):283–314, 1982.

[Eli] Eli's English Home Page. http://www.cs.colorado.edu/~eliuser

[GISK93] Katsuhiko Gondow, Takashi Imaizumi, Yoichi Shinoda, and Takuya Katayama. Change management and consistency maintenance in software development environments using object oriented attribute grammars. In *Object Technologies for Advanced Software (Proc. 1st JSSST Int. Sympo.)*, volume 742 of *Lec. Notes in Comp. Sci.*, pages 77–94. Springer Verlag, 1993.

[Gra96] GrammaTech, Inc., Ithaca, NY., http://www.grammatech.com/, *The Synthesizer Generator Reference Manual*, fifth edition, 1996.

[HGIK97] Takeshi Hagiwara, Katsuhiko Gondow, Takashi Imaizumi, and Takuya Katayama. A Tool for Constructing Software Objectbases from Language Structures. In Yahiko Kambayashi, Yoshifumi Masunaga, Makoto Takizawa, and Yuichiro Anzai, editors, *Information Systems and Technologies for Network Society*, pages 71–78, PO Box 128, Farrer Road, Singapore 912805, 1997. IPSJ, World Scientific Publishing Co. Pte. Ltd.

[Kas80] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[Kas84] U. Kastens. The GAG-System–A Tool for Compiler Construction, pages 165–182. Cambridge University Press, 1984.

[Kas89] U. Kastens. LIGA: A Language Independent Generator for Attribute Evaluators, Universitat-GH Paderborn, Bericht der Reihe Informatik Nr. 63, 1989.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu71] D.E. Knuth. Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[RT84] T. Reps and T. Teitelbaum. *Generating Language Based Environments.* MIT Press, 1984.

[SK90] Yoichi Shinoda and Takuya Katayama. Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm. In *Proc. Int. Workshop on Attribute Grammars and their Applications,* Lec. Note in Comp. Sci. *Vol. 461*, pages 177–191. Springer-Verlag, 1990.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

8. Appendix: Another Example AG G_5

This section provides another example of AG called G_5, its EDP and EDP' to illustrate how both methods OAG and OAG* work for a little larger AG than those defined in the previous examples considered in this paper.

The syntax of G_5 has only block-structures, **goto** statements and labels in procedural languages. The following is an example of G_5.

```
{
   label2:
   {
      goto label1;
   }
   label1:
   goto label2;
}
```

G_5 computes the value of a root attribute Program.err. The value of Program.err is **false** if G has no errors or **true** otherwise. We consider the two following types of errors:

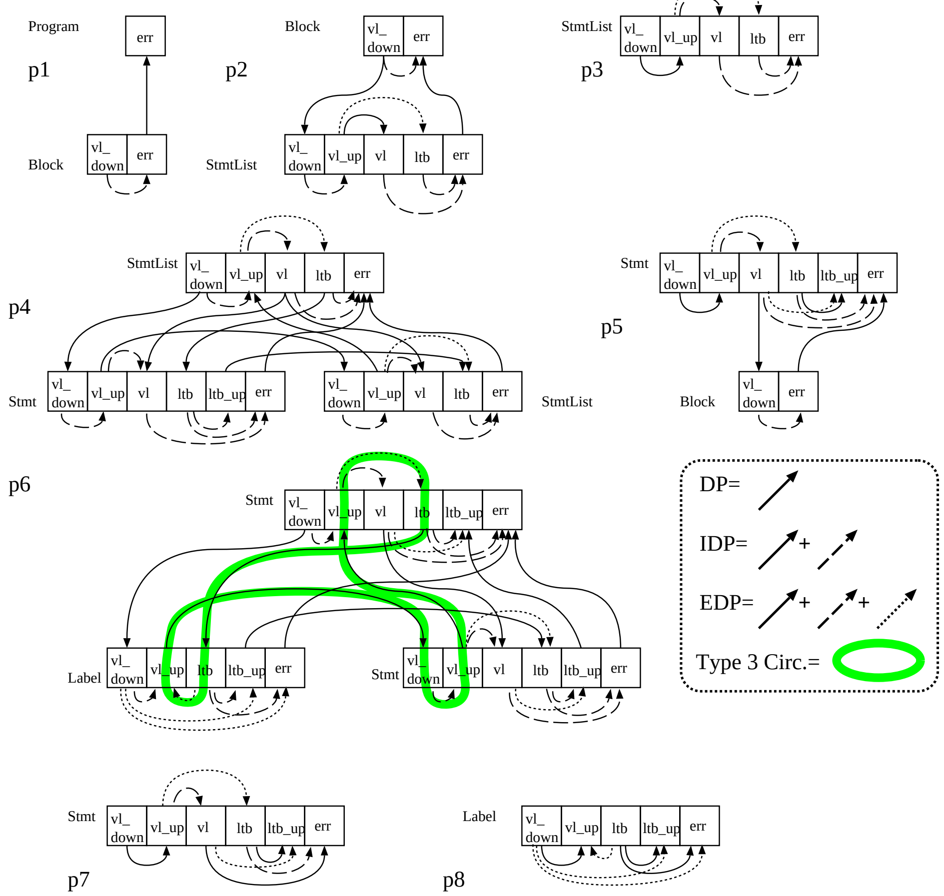
- the same label is already defined before in the same block
- the label in **goto** statement is not defined (we assume here that **goto** statements can not go to the inner blocks.)

The above example of G_5 has no error, so the value of Program.err is **false**. Fig. 10 shows the definition of G_5 where the notation for an attribute occurrence $p : X_i.a$ is abbreviated as $X_i.a$ for simplicity. The identifier *ltb* stands for "labels in the table" and *vl* stands for "visible labels".

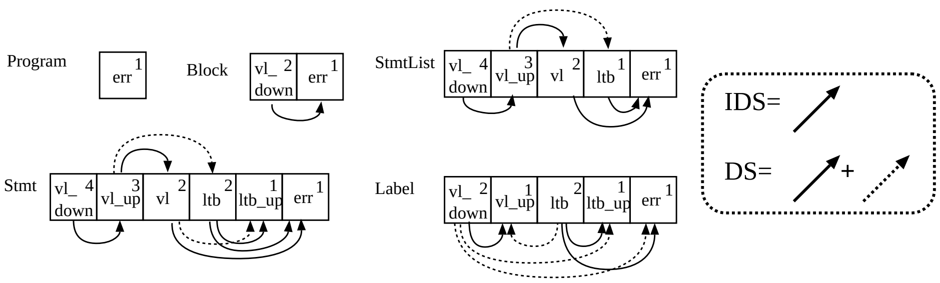
G_5 is not OAG but OAG*. As shown in Fig. 11, G_5 has an OAG's type 3 circularity. If you change the attribute equation appearing in Fig. 10: "p6: Label.ltb = Stmt_1.ltb" to "p6: Label.ltb = **if**(**true**, Stmt_1.ltb, Label.vl_up)", then G_5 becomes OAG. By contrast, EDP' of G_5 without any changes is cyclic free (see, Fig. 12).

p1: Program → Block
Program.err = Block.err
Block.vl_down = ϕ
p2: Block → "{" StmtList "}"
Block.err = StmtList.err
StmtList.vl_down = Block.vl_down
StmtList.vl = StmtList.vl_up
StmtList.ltb = ϕ
p3: StmtList → ϵ
StmtList.vl_up = StmtList.vl_down
StmtList.err = **false**
p4: StmtList → Stmt StmtList
StmtList_1.err = StmtList_2.err **or** Stmt.err
StmtList_1.vl_up = StmtList_2.vl_up
Stmt.vl_down = StmtList_1.vl_down
Stmt.vl = StmtList_1.vl
Stmt.ltb = StmtList_1.ltb
StmtList_2.vl = StmtList_1.vl
StmtList_2.ltb = Stmt.ltb_up
StmtList_2.vl_down = Stmt.vl_up
p5: Stmt → Block
Block.vl_down = Stmt.vl
Stmt.err = Block.err
Stmt.vl_up = Stmt.vl_down
Stmt.ltb_up = Stmt.ltb
p6: Stmt → Label Stmt ";"
Label.vl_down = Stmt_1.vl_down
Label.ltb = Stmt_1.ltb
// Label.ltb = **if**(**true**, Stmt_1.ltb, Label.vl_up)
Stmt_2.vl_down = Label.vl_up
Stmt_2.ltb = Label.ltb_up
Stmt_2.vl = Stmt_1.vl
Stmt_1.err = Label.err **or** Stmt_2.err
Stmt_1.vl_up = Stmt_2.vl_up
Stmt_1.ltb_up = Stmt_2.ltb_up
p7: Stmt → **goto** ID ";"
Stmt.err = ID $\notin$ Stmt.vl
Stmt.vl_up = Stmt.vl_down
Stmt.ltb_up = Stmt.ltb
p8: Label → ID ":"
Label.vl_up = {ID}$\cup$ Label.vl_down
Label.err = ID $\in$ Label.ltb
Label.ltb_up = {ID}$\cup$ Label.ltb

Figure 10: the description of AG G_5

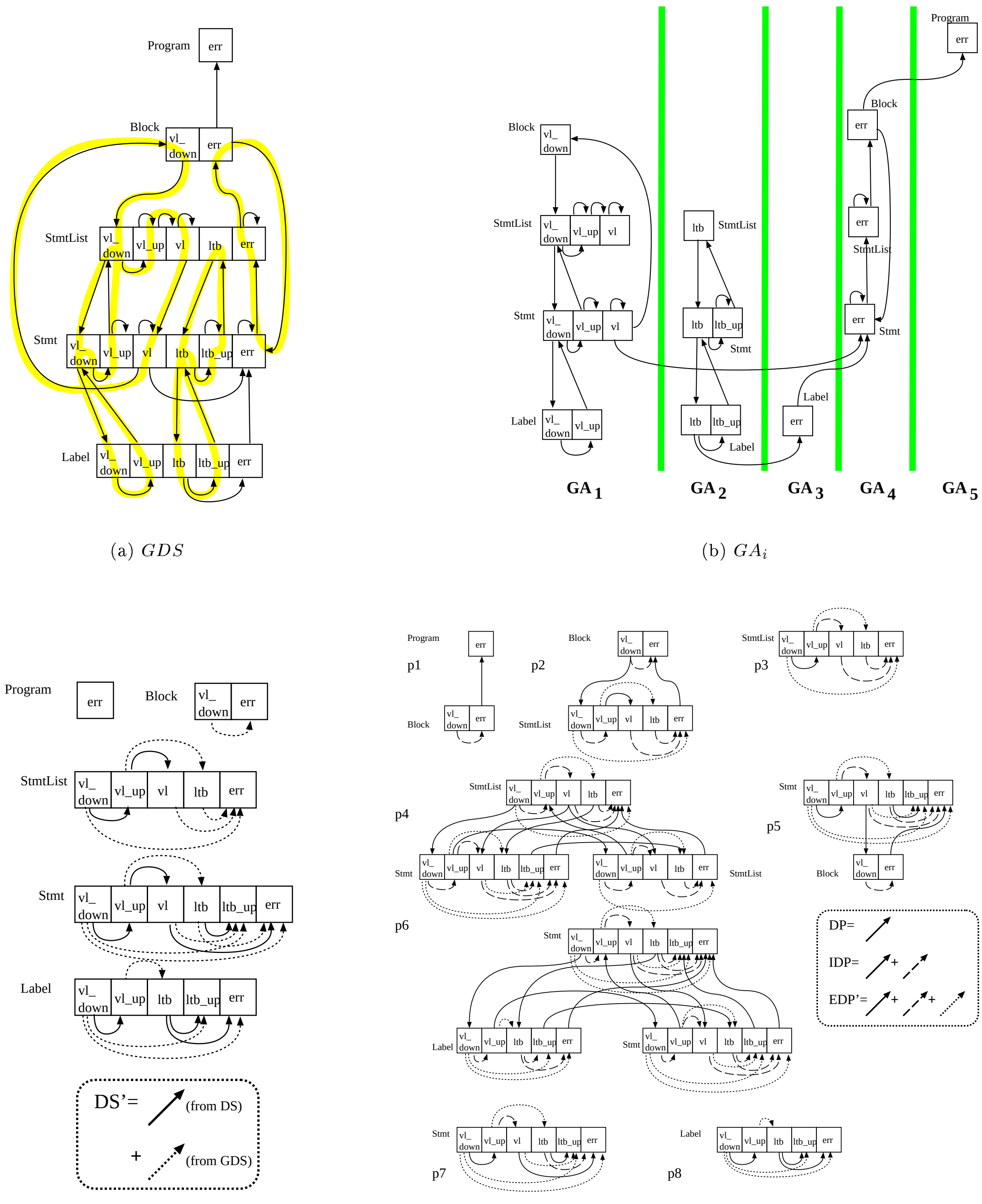


(a) DP, IDP and EDP



(b) IDS and DS

Figure 11: OAG's DP, IDP, IDS, DS and EDP for G_5



(a) GDS (b) GA_i

(c) DS' (d) DP, IDP and EDP'

Figure 12: OAG*'s GDS, GA, DS' and EDP' for G_5