Generalized Systematic Debugging for Attribute Grammars

Akira Sasaki* and Masataka Sassa†

July 10, 2003

Abstract

Attribute grammars (AGs) are known to be a good formalism for semantic analysis and translation. However, debugging AGs is not simple due to the specific difficulty of AGs, such as recursive grammar structure and attribute dependency. We propose a debugging method which lets programmers easily debug AGs without being perplexed with such complicated dependency.

Formerly, two debugging methods of AGs have been proposed, one of which based on algorithmic debugging, and the other based on program slicing. Although these methods achieve a semi-automatic scheme for debugging AG descriptions, several problems remain in either methods. Furthermore, they cannot work simultaneously because of the distinction of their underlying theory.

In this paper, a new systematic debugging method of AGs is proposed. Our approach is, in principle, based on the former algorithmic debugging for AGs but is more general. In fact, the two previous methods can be explained by our new method in a unified framework. This leads to a new debugging method that enables the integration of the two methods.

This generalized method has been implemented in Aki, a debugger for AG description. We evaluate our new approach by experiments using Aki, which shows the usability of our debugging method.

1 Introduction

Debugging of attribute grammars (AGs) involves specific hurdles because of the language feature of AGs such as recursion of syntax structure and complex dependency among attributes. In this paper we propose a novel debugging technique of attribute grammars that reduces such difficulties in debugging of AGs.

The proposed method is an application of algorithmic debugging which is originally proposed as an effective debugging method for logic programming languages. The algorithmic debugging identifies one predicate including a bug that causes a wrong behavior of a program. This is realized by iteratively applying the following: the debugger shows the result of a local execution of a predicate, and asks the programmer whether the result of the execution conforms to the intention of the programmer.

Our group has developed an AG debugging method by applying the algorithmic debugging [10]. In AGs, there seems to be no structure corresponding to "predicate" in logic languages. But computation of AGs involves the same structure as predicates called synth-functions [6]. We have employed synth-functions to get the aspect of predicates.

*The Institute of Medical Science, The University of Tokyo

†Department of Mathematical and Computing Sciences, Tokyo Institute of Technology

Our group has also developed a slice based debugging method for AGs [4], which works complementally with the algorithmic debugging based method. This method also interactively asks the user a question about the behavior of execution and narrows the location of a bug by inference from information given by the user.

In the method in [10] questions to the user are about the correctness of *relation* of attribute values for some subtree. The method is advantageous because the user can concentrate on the question given by the debugger without annoying a big parse tree. On the other hand, in the slice based debugging [4], a question to the user is about the correctness of an attribute *value*. If the user can intuitively tell the correctness of the attribute values, the method has an advantage that it is easier for the user to answer it than to answer about correctness of the relation of the values.

Although these previous methods are effective for debugging of AGs, some limitations still exist. For example, the user may have to answer questions about a huge tree, depending on the location of the bug. Other limitation is that the user has no way to directly give information to the debugger other than by answering the question from the debugger. For example, the user cannot tell the debugger about the correctness of a value even if the user intuitively can understand its correctness. The most obvious problem of previous methods is that they work independently from each other, that is, the user could not switch the debugging methods once the user has started debugging in one method.

In this paper, we propose an AG debugging method that solves the above mentioned problems. This method is based on algorithmic debugging as in [10], but the queries are not limited to questions about the behavior of synth-functions. In that sense, we can say that the proposed method is a generalization of [10]. The method also unifies the slice based method. The best advantage of this method is integration of various query based methods, including the previous two methods, in a single framework. This solves the problem of debugging with a single method and enables effective debugging.

We implemented the new debugging method into our debugger. We will show the effectiveness of the proposed method through experiments.

2 Preliminaries

2.1 Attribute Grammars

Attribute grammar is a formalization that integrates both syntax and semantics of languages. It is widely used as description of language semantics and as specification of analysis or translation in programming language processors.

Figure 1 is a simple example of attribute grammar description. This description calculates the value of a number in binary notation. In this figure, we intentionally include wrong description in part (bug).

An attribute grammar is defined by syntax and semantics. The syntax is defined by a context free grammar and the semantics is defined by semantic rules each of which accompanies a syntax rule. A semantic rule is described by an *attribute* and its *attribute evaluation rule*. The attribute is associated with a nonterminal symbol of a grammar, which represents a variable that expresses semantics of the symbol. Attribute evaluation rules define how each attribute is calculated.

Attribute evaluation is a process that calculates semantics according to the attribute grammar. In other words, this process calculates values of all the attributes on the given parse tree corresponding to the input. The computation is done according to attribute dependency

$$
\begin{array}{lll}
\mathrm{F} & ::= & .\ \ \mathrm{L} \\
& & \{\ \mathrm{L.pos} = 1; \qquad \text{(a)} \\
& & \ \ \mathrm{F.val} = \mathrm{L.val}\ \} \\
\mathrm{L}_0 & ::= & \mathrm{B}\ \ \mathrm{L}_1 \\
& & \{\ \mathrm{L}_1.\mathrm{pos} = \mathrm{L}_0.\mathrm{pos} + 1; \\
& & \ \ \mathrm{B.pos} = \mathrm{L}_0.\mathrm{pos} + 1; \qquad \text{(bug)} \\
& & \ \ \mathrm{L}_0.\mathrm{val} = \mathrm{B.val} + \mathrm{L}_1.\mathrm{val}\ \} \\
& & \ \ |\ \ \mathrm{B} \\
& & \{\ \mathrm{B.pos} = \mathrm{L}_0.\mathrm{pos}; \\
& & \ \ \mathrm{L}_0.\mathrm{val} = \mathrm{B.val}\ \} \\
\mathrm{B} & ::= & 1 \\
& & \{\ \mathrm{B.val} = 2^{-\mathrm{B.pos}}\ \} \\
& & \ \ |\ \ 0 \\
& & \{\ \mathrm{B.val} = 0\ \}
\end{array}
$$

Figure 1: Attribute Grammar $G1$

in semantic rules.

Example 1 For $G1$, the evaluation of ".101" is performed as follows:

1. to construct the parse tree for input ".101"

2. to compute the value of each attribute according to attribute dependency

By this process, an attributed parse tree shown in Figure 2 is made. *F.val* is the result of the evaluation. Since the description includes a bug, the result is not the correct value 5/8. ∎

Nodes in the attributed parse tree are nonterminal symbols like "L", but in this figure we use "$L1$","$L2$" in order to distinguish each instance of nodes.

An attribute associated with a node on the parse tree is called an *attribute instance* (e.g. $L1.pos$), which is distinguished from attributes (e.g. L.pos) in the description.

Here, when an attribute is defined from attributes associated with parent or brother nodes, e.g. L.pos, the attribute is called an *inherited attribute.* When an attribute is defined from attributes associated with their children nodes, e.g. L.val, the attribute is called a *synthesized attribute.*

2.2 Algorithmic Debugging of AGs

Algorithmic debugging [12] is a systematic debugging method originally proposed for logic programming languages and later applied to functional languages. The algorithmic debugging method is formalized by the notion of *computation trees.* A computation tree represents a trace of program execution which corresponds to a proof tree for logic languages, or a call graph for functional languages. On the computation tree, a debugger that employs this method asks a question which is called *query.* A query means a question that is put to the user by the debugger. It asks the user the correctness of a predicate of a logical expression, or a relation between input and output values (arguments and return values) of a function call. By iteratively performing queries, the debugger can finally isolate a predicate or a function that includes a bug.

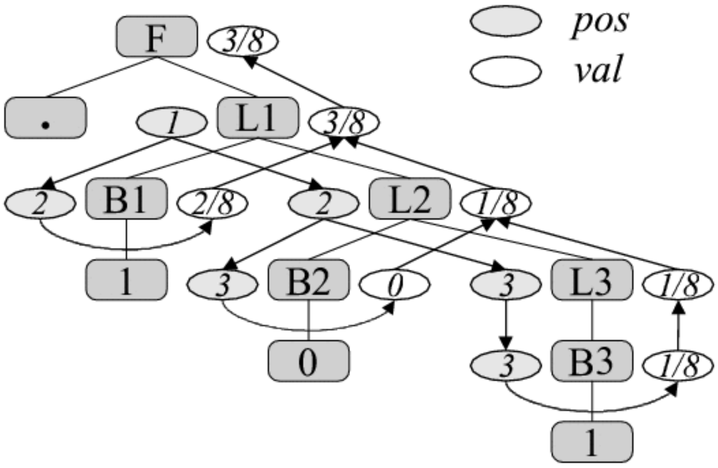

Figure 2: Attributed parse tree

In order to apply this method to AGs, we need a structure equivalent to the computation tree or a recursive structure which can model an execution in AG i.e. attribute evaluation. Sassa et. al [10] realized this by using a structure called *synth-functions*[6] for applying algorithmic debugging to AGs. A synth-function is a fictitious function that can represent computation of attributes. More precisely, a synth-function is a function that computes the value of a synthesized attribute, taking a parse subtree and inherited attributes as arguments. The value of any attribute instance can be represented by applying recursive calls of synth-functions, which in turn enable to construct an equivalent of the computation tree. Figure 3 represents the computation tree made by the attribute evaluation in Example 1. Each node of the computation tree consists of a triple of a function name (of a synth-function), arguments and the result. For example, node (a) represents the computation that the value of $L2.val$ is $1/8$ when $L2.pos = 2$ and the parse tree rooted at $L2$ (substring "01" of the input) are given. The user can answer that this computation (relation) is correct and the debugger can reason that a bug should not be included in the part of the computation tree rooted at (a).

2.3 Systematic Debugging Methods of AGs with Program Slicing

Program slicing [14, 5] is a method that extracts the part of computation (called slice) that affects a given computation. This method has been applied in various area e.g. software engineering, program analysis. Program slicing is quite useful for debugging of programs. The method presented here is a systematic debugging method of AGs, which the debugging method of procedural languages based on "partition of slice" [13] is applied to [4].

We can consider computation of each attribute instance, that is, "a computation that applies a semantic rule and assigns its result to an attribute instance" as unit of computation in AGs. We call such a unit *attribute computation*. Therefore, "A program slice for AGs is a set (or a sequence) of attribute computations that affects an attribute computation."

Figure 4 illustrates computation of Example 1. In this figure, circles, triangles and edges in the left graph represent attribute instances, attribute computations, and flow of attribute values, respectively. Each statement in the right is an attribute computation, and the sequence

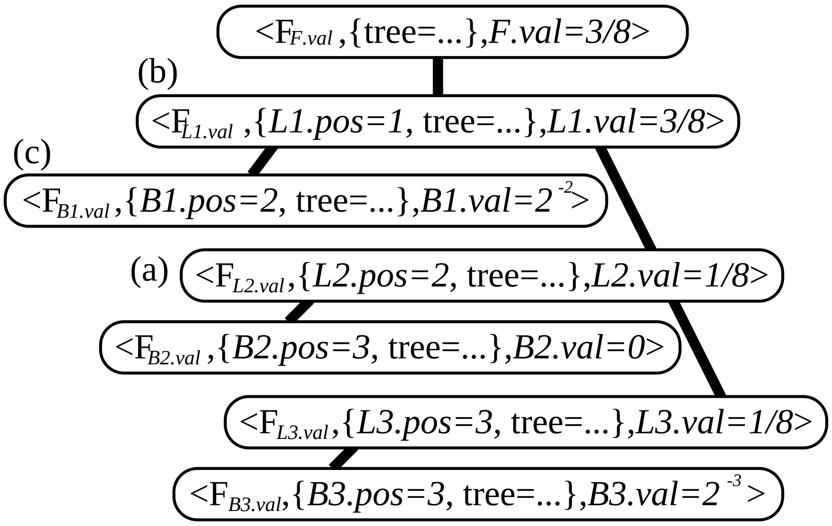

Figure 3: Computation tree

of them represents one possible process of computation, or attribute evaluation. Inside the parentheses after each attribute computation is the result of the computation. (Symbols and broken lines in the graph will be used in a later explanation.)

We illustrate the systematic debugging method of attribute grammars using slice by this example. The debugger starts with separating the sequence of computation to two part s1 and s2 at a boundary. The user judges the correctness of the values of attributes that pass through this boundary (i.e. the values of $L1.pos$ and $L2.val$). If all of such values are correct, part s2 should include at least one bug. If some value is judged as incorrect, the slice of the attribute with the incorrect value includes a bug. By applying partition and query of such sequence successively, candidates of erroneous computation can be localized, and finally one wrong computation will be isolated.

3 Problems of the previous methods and outline of the proposed method

This section points out problems of the previous methods, and then shows the outline of the new method.

3.1 Problems of algorithmic debugging of AGs

There are shortcomings in the previous algorithmic debugging on AGs [10], since in that method, Shapiro's method [12] has been naively applied to AGs.

- It is hard for the user to answer a question that needs information around the root of the whole computation tree.
- The user cannot give any clues to the debugger other than the correctness of synth-functions.

The first problem means that there are cases that the user should look over a large parse tree. For example, comparing questions for node (a) and node (b) in Figure 3, the question

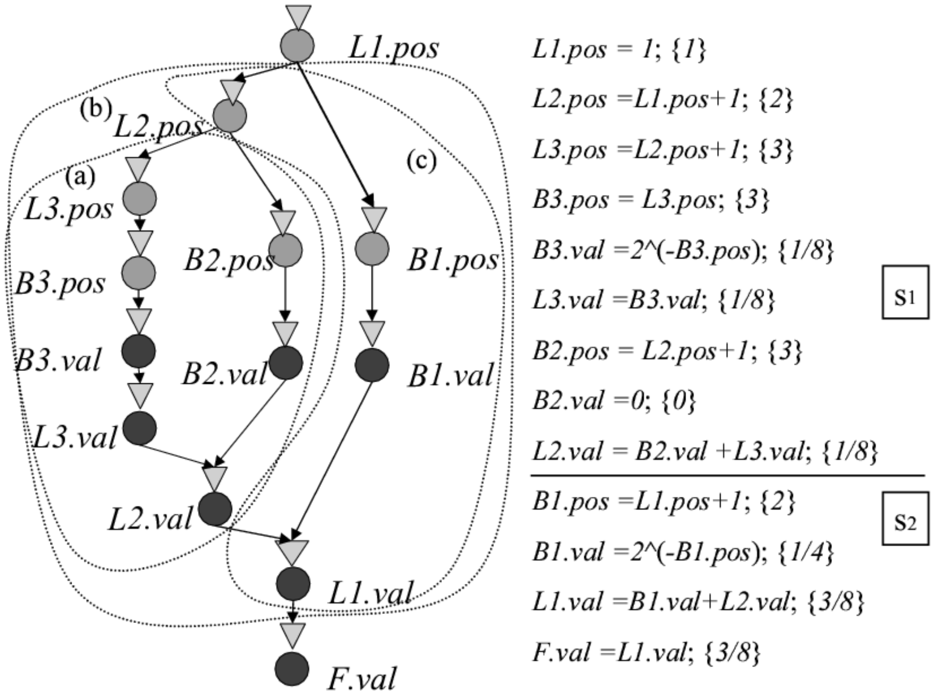

Figure 4: Attribute computation and attribute evaluation

for node (b) is harder to answer because node (b) is nearer to the root and the user should understand a bigger tree (Figure 5). The computation tree consists of nodes representing synth-function calls, each of which takes a subtree as an argument and may recursively call other synth-functions in order to compute the semantics of children of the subtree. This leads that the subtree for an argument of a synth-function may become large if the node corresponding to that synth-function resides near the root of the computation tree. The debugger enforces the user to understand a large tree especially in case that the bug is located in computation of an attribute near the root node.

The second problem is limitation on flexible debugging. For example, when the user tries to answer a question, he may notice that the premise of the question should be wrong, or the value of an inherited attribute for the synth-function should be wrong according to the parse tree as a whole. In this case the user could point out that an attribute value is wrong. Program slice with respect to that attribute should include a bug and it could cut off irrelevant computation from the search space. But after the user has started algorithmic debugging, the debugging method using program slicing cannot be applied in general (and vice versa). For example, suppose the user has narrowed the search space with slice-based debugging and then wants to apply algorithmic debugging in the narrowed search space. This could be done only when the narrowed search space can be mapped to some form of the computation tree, but unfortunately this case may rarely occur.

Other problems that are not mentioned above exist. For example, the previous algorithmic debugging is hard to apply to the program that leads to run-time errors, and the algorithmic debugging generally may indicate a plurality of semantic rules as candidates of bug.

Therefore, the sole use of the previous algorithmic debugging of AGs is not effective.

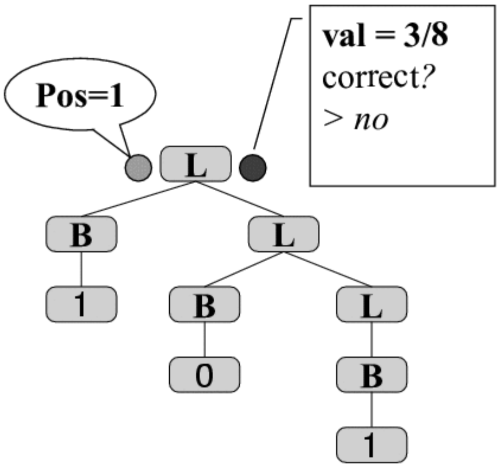

Figure 5: Query with a large subtree

3.2 Outline of the proposed method

The above mentioned problems arise from the fact that every question is based on only one aspect, for example synth-function. By merely using synth-functions, questions always enforce the user to understand the complete form of a subtree.

But the way to ask relation among a subtree and attribute values is not necessarily limited to the form of a synth-function as shown in Figure 6.

In a question for a synth-function, the subtree taken as its argument should be always complete in the sense that the subtree should include all nodes down to leafs in the parse tree. In contrast, the query in this figure is a question about an incomplete subtree where some subtrees belonging to it are removed. The user can answer the question ignoring the part of the removed subtree. From the view point of the computation tree (Figure 3), this question intuitively corresponds to the question about the correctness of "computation for (b) except that for (a)," in place of the question about the synth-function for (b).

The method proposed in this paper is an extended algorithmic debugging that allows a more general form of questions than one in the previous method. By this extension a question in which the user need to look over a large tree, is divided into smaller questions. This method can also use information from the user indicating incorrect values. In the previous algorithmic debugging, once the method is switched to the slice-based method, the user cannot return to the algorithmic debugging. The proposed method is a generalized method, and thus it naturally integrates two methods in one scheme.

The questions about computation in this method are not limited to the form of synth-functions. This comes from the fact that a node in computation tree in the proposed method is not necessary a synth-function. In Example 1 the synth-function for $L2.val$ computes the value of $L2.val$ from the subtree rooted at $L2$ and the value of $L2.pos$. In the graph of attribute evaluation (Figure 4), this computation corresponds to the part surrounded by border (a). In the same manner, the synth-function for $L1.val$ corresponds to (b). The computation in (c), which is not a synth function, can be also considered to represent a function in the proposed method. The question in Figure 6 corresponds to the computation in (c)[1].

[1]The function yields two values i.e. for $L2.pos$ and $L1.val$. So the complete question about computation

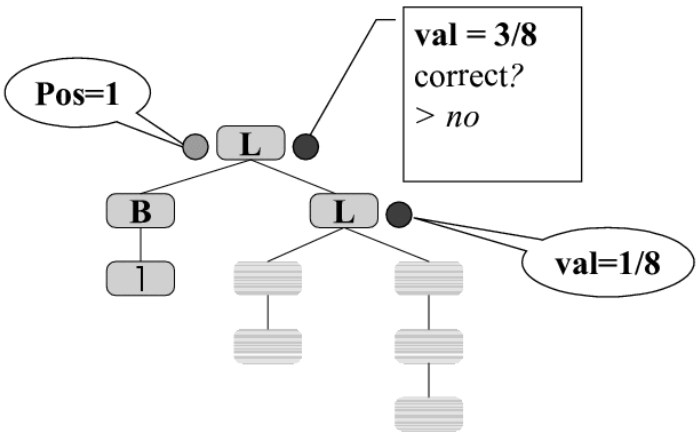

Figure 6: Query with an incomplete subtree

Section 4 shows the way to make such a generalized function that corresponds to some computation in attribute evaluation, and also describes the detail of the extended version of the algorithmic debugging for AGs.

4 Generalization of the algorithmic debugging for AGs

4.1 Attribute Computation

We first recall the model of attribute evaluation, which is roughly divided in the following two steps.

- to construct the parse tree for a given input, and
- to compute the value of each attribute instance according to an evaluation order consistent with the attribute dependency

An *attribute computation* is to compute the value of an attribute instance, as we have mentioned before. We give the definition of an attribute computation as follows.

Definition 1 [Attribute Computation] An *attribute computation* is a computation to determine the value for attribute instance o. The attribute computation for o is denoted as $\langle I \rightarrow o \rangle$, where I is the set of attributes on which o directly depends. An attribute computation $\langle \rightarrow o \rangle$ represents o is independent from other attribute values. ∎

Example 2 In Example 1,

$$\langle (B1.val, L2.val) \rightarrow L1.val \rangle \ , \ \langle \rightarrow L1.pos \rangle$$

represent attribute computations $L1.val = B1.val + L2.val$ and $L1.pos = 1$, respectively. ∎

When we use an example in the rest of this section, we refer to Example 1.

(c) needs the question about $L2.pos$ in addition to the question in Figure 6.

Definition 2 **[Behavior of Attribute Computation]** A *behavior* of attribute computation $\langle I \rightarrow o \rangle$ is denoted as $\langle \langle I \rightarrow o \rangle, X, y \rangle$ which means that the values of $I = (i_1, \cdots, i_n)$ are $X = (x_1, \cdots, x_n)$. ∎

Example 3 The behavior of attribute computation $\langle (B1.val, L2.val) \rightarrow L1.val \rangle$ is the following:

$$\langle \langle (B1.val, L2.val) \rightarrow L1.val \rangle, (1/4, 1/8), 3/8 \rangle$$

∎

4.2 Attribute Computation Composition

From the view point of attribute computations, a synth-function for the previous algorithmic debugging method is considered as

> a function that is constructed by the composition of "an attribute computations from a set of inherited attributes $N.I$ to synthesized attribute $N.s$."

Such a composition of attribute computations is called *attribute computation composition*. Attribute computation compositions can be made from any combination of attribute computations on which attribute instances directly or indirectly depend.

In Example 1, an attribute computation composition from $L1.pos$ to $B3.pos$ is the composition of the sequence of attribute computations as follows (Figure 4):

$$\langle L1.pos \rightarrow L2.pos \rangle$$
$$\langle L2.pos \rightarrow L3.pos \rangle$$
$$\langle L3.pos \rightarrow B3.pos \rangle,$$

which results in computation $B3.pos = L1.pos + 2$. If the user can intuitively tell the correctness of such a composition of attribute computations, a bug of a program can be localized efficiently.

The definition of an attribute computation composition is as follows.

Definition 3 **[Attribute Computation Composition]** Provided that attribute instances $I = (i_1, \cdots, i_n)$ and $O = (o_1, \cdots, o_m)$ satisfy the following:

> For $i \in I$, there exists $o \in O$ such that o depends on i directly or indirectly.

Then, the *attribute computation composition* from I to O, denoted as $\langle I \Rightarrow O \rangle$, is the computation to determine the values of O when all of the values of I are given. $\langle \Rightarrow O \rangle$ represents the case for $I = \{\}$. For $\langle I \Rightarrow O \rangle$, I and O are called the *input* and the *output* for the attribute computation composition, respectively.

The set of attribute computations included in the computation from I to O is denoted as $Set(\langle I \Rightarrow O \rangle)$ and is defined as follows.

$$
\begin{array}{ll}
(1) & \text{For any } o \in O \\
 & \quad \langle X \rightarrow o \rangle \in Set(\langle I \Rightarrow O \rangle) \\
(2) & \langle X \rightarrow y \rangle \in Set(\langle I \Rightarrow O \rangle) \\
 & \Rightarrow \text{For any } x \in X \wedge x \notin I \\
 & \qquad \langle X' \rightarrow x \rangle \in Set(\langle I \Rightarrow O \rangle)
\end{array}
$$

Intuitively, $Set(\langle I \Rightarrow O \rangle)$ is the attribute computations that can be collected by reversely traversing dependency edges from each $o \in O$ to $i \in I$. Note that there is a case that reverse traversing results in no input attributes $i \in I$, which means $\langle \rightarrow o \rangle$.

∎

Example 4 In Example 1 (Figure 4),

$$\begin{aligned} & Set(\langle L2.pos \Rightarrow (L3.pos, B1.val) \rangle) \\ & \quad = \{ \quad \langle L2.pos \rightarrow L3.pos \rangle, \langle \rightarrow L1.pos \rangle, \\ & \qquad\quad \langle L1.pos \rightarrow B1.pos \rangle, \langle B1.pos \rightarrow B1.val \rangle \} \end{aligned}$$

∎

If the given parse tree is T, and the synthesized attributes of T are $(s_1, \cdots, s_n)$, the whole attribute evaluation is represented by $\langle \Rightarrow (T.s_1, \cdots, T.s_n) \rangle$.

Definition 4 [Behavior of Attribute Computation Composition] Behavior of attribute computation composition $\langle I \Rightarrow O \rangle$ can be denoted by the similar notation as one of behavior of an attribute computation. That is, if the values of attribute instances I, O are X , Y respectively, $\langle \langle I \Rightarrow O \rangle, X, Y \rangle$ is the behavior of $\langle I \Rightarrow O \rangle$. ∎

Note that Definitions 3 and 4 tell that the output of an attribute computation composition is determined only from the values of the corresponding inputs. Generally, the value of an attribute instance also depends on the parse subtree. However, we assume that the parse tree is an a-priori constant until Section 4.5.

4.3 Properties of Attribute Computation Composition

This section describes some properties of attribute computation compositions.

Inclusion $Set(\langle I_1 \Rightarrow O_1 \rangle) \supset Set(\langle I_2 \Rightarrow O_2 \rangle)$ means that $\langle I_2 \Rightarrow O_2 \rangle$ is a nested or partial computation in $\langle I_1 \Rightarrow O_1 \rangle$.

Example 5 In Example 1 (Figure 4),

$$\begin{aligned} & Set(\langle L1.pos \Rightarrow L1.val \rangle) \\ & \qquad \supset Set(\langle L3.pos \Rightarrow L3.val \rangle) \end{aligned}$$

∎

In the computation tree of the algorithmic debugging for AGs (Figure 3), the synth-function for $L3.val$ is a descendant of the one for $L1.val$. Thus the descendant or ancestor relation in a computation tree corresponds to the inclusion relation in attribute computation compositions.

Example 6 In Example 1 (Figure 4(b) and (c)),

$$\begin{aligned} & Set(\langle L1.pos \Rightarrow L1.val \rangle) \\ & \qquad \supset Set(\langle (L1.pos, L2.val) \Rightarrow (L1.val, L2.pos) \rangle) \end{aligned}$$

∎

As in Example 6, the inclusion relation holds even when two attribute computation compositions do not correspond to synth-functions.

Disjoint When $Set(\langle I_1 \Rightarrow O_1 \rangle) \cap Set(\langle I_2 \Rightarrow O_2 \rangle) = \phi$, or the two sets of attribute computation compositions are disjoint, it means that each computation involved in $\langle I_1 \Rightarrow O_1 \rangle$ or $\langle I_2 \Rightarrow O_2 \rangle$ does not affect each other.

Example 7 In Example 1,

$$\begin{aligned}&Set(\langle L2.pos \Rightarrow L2.val \rangle)\\ &\quad \cap Set(\langle B1.pos \Rightarrow B1.val \rangle) = \phi\end{aligned}$$

∎

In a computation tree of the algorithmic debugging for AGs, when two synth-functions is not in the ancestor relation, they are in the disjoint relation.

Example 8 In Example 1 (Figure 4(a) and (c)),

$$\begin{aligned}&Set(\langle L2.pos \Rightarrow L2.val \rangle)\\ &\quad \cap Set(\langle (L1.pos, L2.val) \Rightarrow (L1.val, L2.pos) \rangle)\\ &\quad = \phi\end{aligned}$$

∎

This example suggests that the disjoint relation may hold between any attribute computation compositions, not limited to synth-functions.

4.4 Closed Attribute Computation Composition

Concerning debugging, if the user can judge the relation between the input and the output values of the behavior of $\langle I \Rightarrow O \rangle$ to be false, $Set(\langle I \Rightarrow O \rangle)$ includes at least one bug. On the other hand, it cannot be generally concluded that $Set(\langle I \Rightarrow O \rangle)$ does not include any bugs from the judgment of a behavior of $\langle I \Rightarrow O \rangle$ being true. A closed attribute computation composition is an attribute computation composition that can be ensured that $Set(\langle I \Rightarrow O \rangle)$ does not include bugs when the user judges the behavior of $\langle I \Rightarrow O \rangle$ to be true [2].

Definition 5 [Completely Closed Attribute Computation Composition] We define AS as $\mathsf{AS} = Set(\langle I \Rightarrow O \rangle) - \{\langle X \rightarrow o \rangle \mid o \in O\}$. The attribute computation composition $\langle I \Rightarrow O \rangle$ is *completely closed* iff all the computation in AS cannot reach computations outside $Set(\langle I \Rightarrow O \rangle)$, i.e. $Set(\mathsf{AE}) - Set(\langle I \Rightarrow O \rangle)$, without passing through one of the computations in $\{\langle X \rightarrow o \rangle \mid o \in O\}$. Here, AE represents the attribute computation composition for the whole of the attribute evaluation. ∎

The definition is intuitively illustrated in Figure 7. $\langle I_2 \Rightarrow O_2 \rangle$ is *not* completely closed because attribute computations (a) and (b) included in $\mathsf{AS}_2 = Set(\langle I_2 \Rightarrow O_2 \rangle)$ can affect computations (c) and (d) which are outside the $Set(\langle I_2 \Rightarrow O_2 \rangle)$ without passing through computations for O_2.

[2] Precisely, this ensures that when an attribute computation composition with incorrect behavior includes $\langle I \Rightarrow O \rangle$, $Set(\langle I \Rightarrow O \rangle)$ never includes the source of the incorrect behavior.

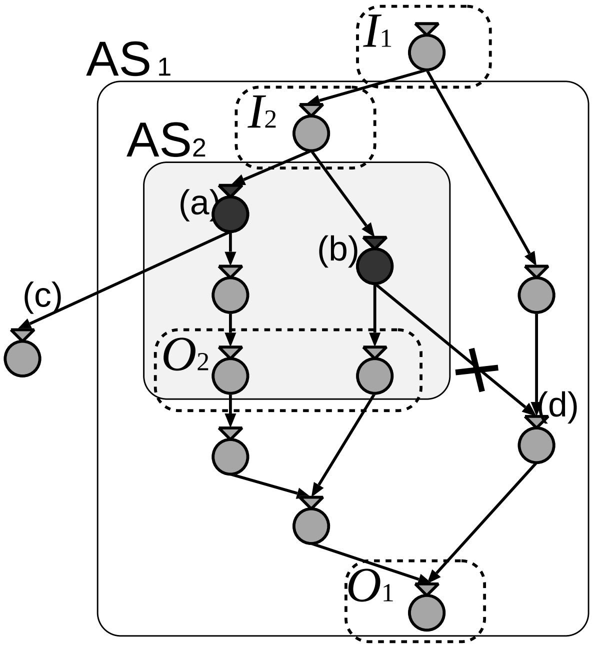

Figure 7: Definition of completely closed attribute computation composition

Example 9 In Example 1, $\langle\Rightarrow B1.pos\rangle$ is not completely closed (Figure 8(a)). In this example,

$$\begin{aligned}&Set(\langle\Rightarrow B1.pos\rangle)\\&\quad=\{\langle\rightarrow L1.pos\rangle,\langle L1.pos\rightarrow B1.pos\rangle\}.\end{aligned}$$

holds. The set corresponding to AS in Definition 5 is $\{\langle\rightarrow L1.pos\rangle\}$. But $\langle\rightarrow L1.pos\rangle$ affects computations outside $Set(\langle\Rightarrow B1.pos\rangle)$ (e.g. the computation for $L2.pos$) without passing through the computation for $B1.pos$. Consequently, $\langle\Rightarrow B1.pos\rangle$ is not completely closed. ∎

Example 10 In Example 1 $\langle\Rightarrow(B1.pos, L2.pos)\rangle$ is completely closed (Figure 8(b)), since every computation in

$$Set(\langle\Rightarrow(B1.pos, L2.pos)\rangle)$$

cannot reach outside their computations without passing through the computations for $B1.pos$ or $L2.pos$. ∎

Note that any attribute computation compositions that are equivalent to (or that correspond to) synth-functions are completely closed.

In the above definition, the terminology "completely" means that the criterion of "computations outside" is the whole of attribute evaluation AE. On the other hand, the following definition is a generalized version in the sense that the criterion can be taken as any attribute computation ACC that satisfies $Set(\langle I\Rightarrow O\rangle)\subset Set(\mathsf{ACC})$.

Definition 6 [Closed Attribute Computation Composition] For two attribute computation compositions $\langle I_1\Rightarrow O_1\rangle$ and $\langle I_2\Rightarrow O_2\rangle$, let $\mathsf{AS}_1 = Set(\langle I_1\Rightarrow O_1\rangle)$, $\mathsf{AS}_2 = Set(\langle I_2\Rightarrow O_2\rangle)$ and suppose $\mathsf{AS}_1\supset\mathsf{AS}_2$. Let $\mathsf{AS}_{O_2}=\{\langle X\rightarrow o\rangle \mid o\in O_2\}$. Then, $\langle I_2\Rightarrow O_2\rangle$ is *closed with respect to* $\langle I_1\Rightarrow O_1\rangle$ iff every computation in $\mathsf{AS}_2-\mathsf{AS}_{O_2}$ does not affect any computations in $\mathsf{AS}_1-\mathsf{AS}_2$

without passing through computations in AS_{O_2} [3]. This is denoted by $Closed(\langle I_2 \Rightarrow O_2\rangle, \langle I_1 \Rightarrow O_1\rangle)$. Precisely, the condition is that for the set

$$Y = \{y \mid \langle X \rightarrow y\rangle \in \mathsf{AS}_2 - \mathsf{AS}_{O_2}\},$$

the following holds:

$$\{\langle X' \rightarrow y'\rangle \mid \text{arguments } X' \text{ includes an element of } Y\} \\ \cap (\mathsf{AS}_1 - \mathsf{AS}_2) = \phi$$

In other words, no attribute computation in $\mathsf{AS}_1 - \mathsf{AS}_2$ depends on the results of computations in $\mathsf{AS}_2 - \mathsf{AS}_{O_2}$ unless it passes through the computations for O_2. ∎

The definition is illustrated by Figure 7 where each symbol in the figure corresponds to that of the definition. In this figure, let us focus on $\langle I_2 \Rightarrow O_2\rangle$. $\langle I_2 \Rightarrow O_2\rangle$ is not closed (w.r.t AS_1) because attribute computation (b) affects computations in $\mathsf{AS}_1 - \mathsf{AS}_2$ (e.g. (d)) without passing through computations in O_2. On the other hand, attribute computation (a) affects attribute computation (c) but does not affect the computations in $\mathsf{AS}_1 - \mathsf{AS}_2$. So if we temporarily suppose that (d) does not directly depend on (b) in the figure, $Closed(\langle I_2 \Rightarrow O_2\rangle, \langle I_1 \Rightarrow O_1\rangle)$ will be satisfied.

Example 11 In Example 1, $\langle \Rightarrow B1.pos\rangle$ is closed with respect to $\langle \Rightarrow B1.val\rangle$ (Figure 8(a) and (c)), that is,

$$Closed(\langle \Rightarrow B1.pos\rangle, \langle \Rightarrow B1.val\rangle)$$

.

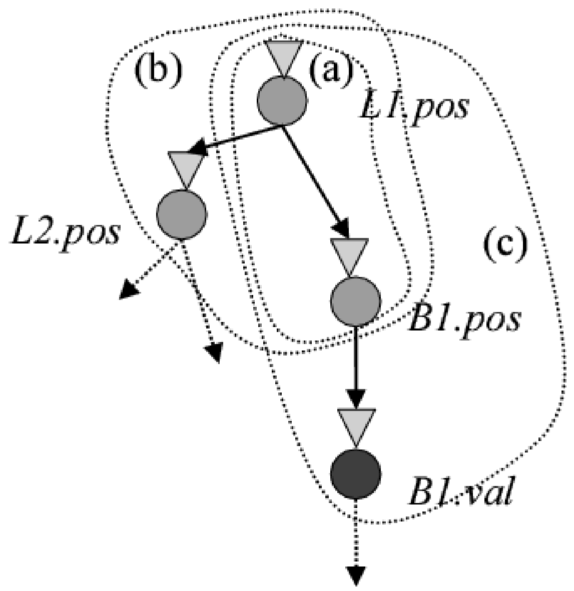

Figure 8: Closed attribute computation composition

Every attribute computation in $\langle \Rightarrow B1.pos\rangle$ never affects the value of $B1.val$ without passing through the computation for $B1.pos$. $B1.val$ does not depend on the attribute computations $\langle \Rightarrow B1.pos\rangle$ without passing through $B1.pos$. ∎

[3]The criterion for the outside computation is $\langle I_1 \Rightarrow O_1\rangle$ in this case.

Note that attribute computation composition $\langle \Rightarrow B1.pos \rangle$ is closed with respect to $\langle \Rightarrow B1.val \rangle$, but is not completely closed, since $F.val$ depends on the computation for $L1.pos$ (inside $Set(\langle \Rightarrow B1.pos \rangle)$) without passing through the computation for $B1.pos$ (e.g. by passing through the computation of $L2.pos$).

4.5 Generalized Algorithmic Debugging

Definition 7 [correct/incorrect behavior] For a behavior of an attribute computation composition $\langle \langle I \Rightarrow O \rangle, X, Y \rangle$, the behavior is said to be *correct/incorrect* if the relation between the values of X and Y is correct/incorrect, according to the user's intention or knowledge on $\langle I \Rightarrow O \rangle$. This is denoted as follows.

$$\begin{aligned} \text{Query}(\langle I \Rightarrow O \rangle) &= \text{correct} \\ \text{Query}(\langle I \Rightarrow O \rangle) &= \text{incorrect} \end{aligned}$$

∎

Subtree to be considered

Generally, the user should investigate the whole of the parse tree to judge whether a behavior is correct or incorrect. But actually, the user can judge the correctness of the behavior with looking only a part of the tree, rather than the whole tree.

Attribute computation composition $\langle I \Rightarrow O \rangle$ is determined uniquely from $Set(\langle I \Rightarrow O \rangle)$, if a parse tree is given. The part of the tree that affects $\langle I \Rightarrow O \rangle$ can be determined by appropriately applying the productions (production rules) which define the attributes whose computations are in $Set(\langle I \Rightarrow O \rangle)$.

In Example 1, to judge the correctness of the behavior of $\langle \Rightarrow B1.val \rangle$, the user need to know only a part of the given tree. The following are the productions that define the attributes whose computations are included in $Set(\langle \Rightarrow B1.val \rangle)$:

Attribute Computation	Production
$\langle \rightarrow L1.pos \rangle$	F ::= . L
$\langle L1.pos \rightarrow B1.pos \rangle$	L_0 ::= B L_1
$\langle B1.pos \rightarrow B1.val \rangle$	B ::= 1

Therefore, the relevant subtree can be determined by applying productions {F ::= . L, L_0 ::=B L_1, B ::= 1}. This is actually the tree after removing the subtree of $L2$ from the whole tree in Figure 2.

Definition 8 [Bug] Let an attribute computation composition be $\langle I \Rightarrow (o) \rangle$. Suppose that $Set(\langle I \Rightarrow (o) \rangle)$ consists of the single element $\langle I \rightarrow o \rangle$. An attribute computation $\langle I \rightarrow o \rangle$ is called a *bug* when the behavior of $\langle I \Rightarrow (o) \rangle$ is incorrect [4]. ∎

Theorem 1 At least one attribute computation in $Set(\langle I \Rightarrow O \rangle)$ has a bug if Query($\langle I \Rightarrow O \rangle$) = incorrect. ∎

The proof is almost obvious and is omitted.

[4]Although it is more natural to say "the *semantic rule* for o includes a bug", we consider that a"bug" is included in one of the *attribute computations*.

Theorem 2 Suppose that Query($\langle I_1 \Rightarrow O_1 \rangle$) = incorrect. If Query($\langle I_2 \Rightarrow O_2 \rangle$) = correct for any closed $\langle I_2 \Rightarrow O_2 \rangle$ with respect to $\langle I_1 \Rightarrow O_1 \rangle$, $Set(\langle I_1 \Rightarrow O_1 \rangle) - Set(\langle I_2 \Rightarrow O_2 \rangle)$ includes at least one bug. ∎

Informal proof is shown in [8]. Theorem 2 does not hold if the condition that $\langle I_2 \Rightarrow O_2 \rangle$ is closed with respect to $\langle I_1 \Rightarrow O_1 \rangle$ is not satisfied. In other words, without the condition of closeness, there is a case that the user cannot find bugs.

Corollary 1 Suppose Query($\langle I \Rightarrow O \rangle$)=incorrect. If a set of attribute computation compositions $\{\mathsf{ACC}_k\}(1 \le k \le n)$ satisfies the following condition (Figure 9):

- $Closed(\mathsf{ACC}_k, \langle I \Rightarrow O \rangle)$
- Query(ACC_k)=correct
- $Set(\mathsf{ACC}_i)$ and $Set(\mathsf{ACC}_j)$ are disjoint ($i \neq j,\ 1 \le i \le n,\ 1 \le j \le n$)

then

$$Set(\langle I \Rightarrow O \rangle) - \bigcup_{k=1}^{n} Set(\mathsf{ACC}_k)$$

includes at least one bug. ∎

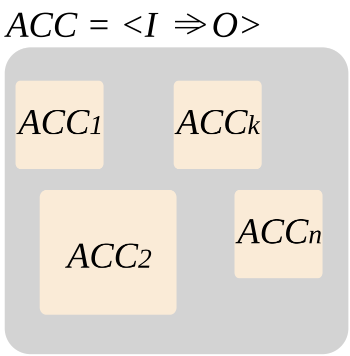

Figure 9: Illustration of Corollary 1

Corollary 1 does not always hold without the condition that ACC_k's satisfy the disjointness relation. To show this, let ACC_1 and ACC_2 be attribute computation compositions that satisfy the condition of Corollary 1 but are not disjoint. It follows that there is an attribute computation composition ACC' that satisfies the following:

- $Closed(\mathsf{ACC}', \mathsf{ACC})$
- $Set(\mathsf{ACC}') = Set(\mathsf{ACC}_1) \cup Set(\mathsf{ACC}_2)$

From the premises the behavior of both ACC_1 and ACC_2 are correct. But as the following example will show, this does not ensure the correctness of the behavior of ACC' . Thus it is not guaranteed that $Set(\mathsf{ACC}) - Set(\mathsf{ACC}')$ should always include bugs.

Corollary 2 Let ACC be $\langle I \Rightarrow O \rangle$. Suppose that Query($\mathsf{ACC}$)=incorrect and that the set of attribute computation compositions $\{\mathsf{ACC}_k\}(1 \le k \le n)$ satisfies the same condition as in the premises on $\{\mathsf{ACC}_k\}$ in Corollary 1.

Now consider an attribute computation composition is ACC' that satisfies the following condition (Figure 10).

- $Closed(\mathsf{ACC}', \mathsf{ACC})$
- $Set(\mathsf{ACC}_k) \subset Set(\mathsf{ACC}') \qquad (1 \leq k \leq m)$
- $Set(\mathsf{ACC}_k)$ and $Set(\mathsf{ACC}')$ are disjoint $(m < k \leq n)$.

If Query(ACC')=correct, then

$$Set(\mathsf{ACC}) - (Set(\mathsf{ACC}') \cup \bigcup_{k=m+1}^{n} Set(\mathsf{ACC}_k))$$

[5] includes at least one bug. If Query(ACC')= incorrect, then

$$Set(\mathsf{ACC}') - \bigcup_{k=1}^{m} \mathsf{ACC}_k$$

includes at least one bug. ∎

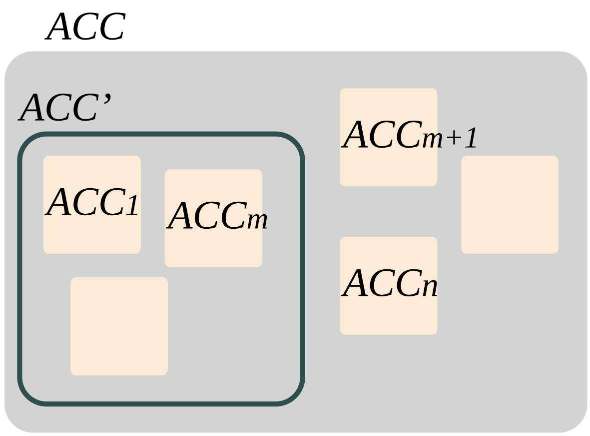

Figure 10: Illustration of Corollary 2

4.6 Debugging Algorithm

The theorem in the previous section leads to a generalized version of algorithmic debugging for AGs, GAD, which has a simple form as in Figure 11.

This algorithm localizes a bug by recursively applying Corollary 2. GAD_{init} is the function that gives an initial condition, and GAD_{main} is the actual function that performs the recursive applications of Corollary 2. The argument ACC of GAD_{main} represents an attribute computation composition whose behavior is incorrect, that is, Query(ACC) = incorrect. The second argument $\{\mathsf{ACC}_1, \cdots, \mathsf{ACC}_n\}$ represents a set of attribute computation compositions that satisfies the premise in Corollary 1.

Function $getNextACC$ selects an ACC' and m[6] that satisfies Corollary 2 for the next query, given arguments ACC and $\{\mathsf{ACC}_1, \cdots, \mathsf{ACC}_n\}$ that satisfy the premise of Corollary 1.

[5] This expression is equivalent to

$$Set(\mathsf{ACC}) - (Set(\mathsf{ACC}') \cup \bigcup_{k=1}^{n} Set(\mathsf{ACC}_k))$$

[6] m is actually determined from the way to take ACC'.

```
GAD_init(ACC){
  return GAD_main(ACC, {});
}

GAD_main(ACC, {ACC_1, ..., ACC_n}){
  /* Corollary 1 */
  bugACs = Set(ACC) − ⋃_{1≤k≤n} Set(ACC_k);
  if size(bugACs) ≤ ε
    /* report the candidates of a bug */
    return bugACs;
  /* select ACC′ and m that satisfy Corollary 2 */
  ACC′ = getNextACC(ACC, {ACC_1, ..., ACC_n}) ;
  if Query(ACC′) == correct
    /* if the behavior is correct, then add ACC′
      to correct ACC set */
    return
      GAD_main(ACC, {ACC′, ACC_{m+1}, ..., ACC_n});
  else
    /* narrow the search space to ACC′ */
    return GAD_main(ACC′, {ACC_1, ..., ACC_m});
}
```

Figure 11: Algorithm GAD

This algorithm reduces the number of attribute computations that are candidates of a bug to some small ϵ in a finite number of queries, provided a correct implementation of $getNextACC$. This fact can be verified by (1) in the moment that a GAD_{main} is called, ACC and $\{\mathsf{ACC}_1, \cdots, \mathsf{ACC}_n\}$ satisfy Corollary 1, and (2) the recursive call applied in GAD_{main} always narrows the search space.

This algorithm covers the previous methods described before and is a generalization of them. Various debugging methods can be induced from this algorithm by changing the realization of function $getNextACC$, which determines the form of query to the user.

The previous algorithmic debugging by synth-functions can be realized in this algorithm as follows: let the function $getNextACC$ select an ACC' that satisfies Corollary 2 from the set of erroneous attribute computation compositions equivalent to synth-functions.

The slice based debugging can be also realized as follows: let the function select a candidate of ACC' that satisfies Corollary 2 from the set of attribute computation compositions having the form $\langle \Rightarrow O \rangle$, that is, erroneous attribute computation compositions whose inputs are empty. In both methods, binary search can realize an efficient localization of bugs. The query is selected so that the numbers of candidates of attribute computation compositions are almost the same in either cases that Query(ACC') is correct or incorrect.

5 Debugger Aki

In this section debugger Aki, an experimental debugger for AGs, is introduced. The debugger Aki is implemented as a part of our compiler development environment with AGs [9]. In this

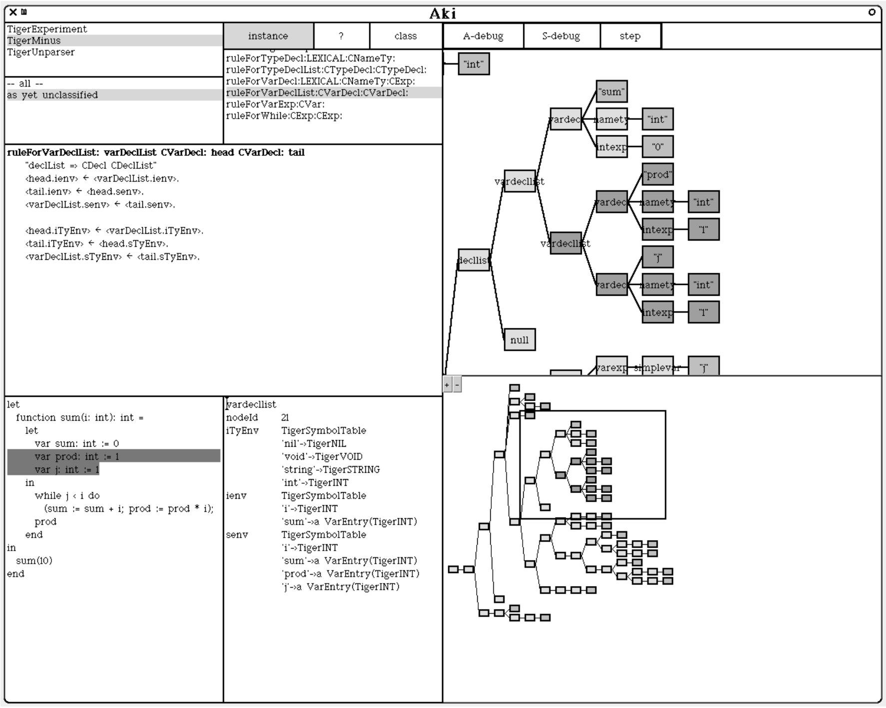

Figure 12: Debugger Aki

development environment, each phase of a compiler is described in AG. Attribute evaluator generators Rie and Jun generate front-ends and back-ends for the compiler respectively. The debugger Aki assists debugging of descriptions of compiler phases written in AGs.

Aki provided two debugging methods: the previous algorithmic debugging (in Section 2.2) and the method based on partition of slices [4]. In addition, the generalized method proposed by this paper in Section 4.6 is implemented for this research.

Aki is designed so that it incorporates with any attribute grammar systems. But the current implementation assumes to work with attribute evaluator generator Jun. The debugging of a description by Aki needs the corresponding attribute evaluator generated by Jun. This evaluator is extended so as to enable attribute instance values to be traced for debugging purposes.

Aki performs systematic debugging by using information of attribute dependency that can be obtained from the AG description, an input parse tree, and the trace of evaluation of attribute instance values.

Figure 12 shows a screen shot of debugging of an AG description using Aki. The panes display the AG description, the source program, attribute values and the input tree. Queries are given by a dialog window as shown in Figure 13. When Aki detects the semantic rule that includes a bug, Aki highlights the rule as in Figure 14.

6 Experimental Result and Discussion

In this section, we discuss the efficiency of the proposed debugging method, including user tests.

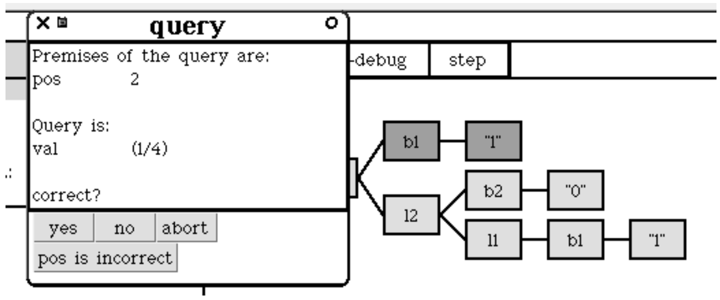

Figure 13: Query

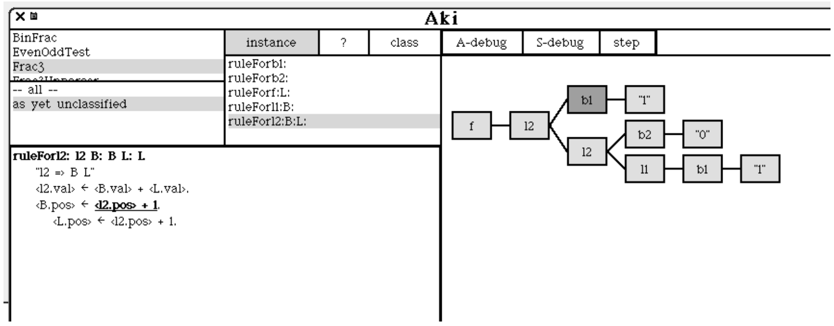

Figure 14: An erroneous rule inferred by Aki

6.1 User Tests

We made user tests of the debugger Aki presented in the previous section. The user tests have been done by three users. The users tried to find erroneous positions in six programs using three methods i.e. slice-based, pure algorithmic and our generalized method. The main purpose of this experiment is to compare easiness of answering questions in each method.

The example attribute grammar is a description of static semantics checking of "Tiger language" from a textbook of compiler construction [1]. The description is composed of 25 productions and 105 semantic rules.

Table 1: Comparison of the number of queries

test case	# attrs	# nodes	Slice	AD	GAD
A	78	52	7(1)	8(3)	4(1)[6(1)]
B	146	103	6(1)	9(4)	5(1)
C	60	43	4(1)	8(3)	4(1)
D	56	34	9(1)	6(4)	7(1)
E	45	32	5(1)	5(2)	7(2)
F	104	69	7(1)	6(2)	2(2)

Table 1 shows the number of the queries that the users needed to identify an erroneous portion in each description. In this table, "# attrs" means the number of attribute instances, but in the case of run time errors, it is the number of attribute instances that are evaluated in the attribute evaluation. "# nodes" is the number of nodes in each parse tree. "Slice","AD" and "GAD" represent slice-based method, pure algorithmic debugging and the generalized method proposed in the previous section, respectively. The numbers in parentheses are the numbers of candidates of semantic rules identified as bugs.

In this experiment, we cannot conclude that a method with fewer queries is more efficient, because some of questions are hard to answer and some are easy. We will discuss this fact in the next section. In the example, the numbers in the bracket of raw A means that one user had a different result from other users. This is because users can freely indicate erroneous values of inherited attributes in GAD.

6.2 Discussion

In this section, we discuss advantages of the proposed debugging method based on the result given by the above user tests and our experiment using Aki.

Easy question to answer

In the method based on synth-functions (AD in Table 1) and the slice based debugging (Slice), to answer the question is the only way that the user can give debuggers information for bug localization. On the other hand, in the proposed method (GAD), the user can indicate inherited attribute values that are found to be wrong. Although this method may increase the number of queries, the focus of the search tends to go near the user's interest, which leads to efficient debugging.

Runtime error

On the previous algorithmic debugging, it is hard to debug descriptions which causes runtime error while evaluation. In such a case, undefined synthesized attributes may be included in the query (in the case of E and F using AD in Table 1). In this situation, the user should answer such a question like, "is it correct that a synthesized attribute should be undefined for this premise?", which is difficult.

On the slice based debugging, it is possible to start debugging from where the runtime error have occurred. But the user needs to indicate correctness based on a careful investigation of the whole parse tree.

In the proposed method, the debugger never asks a question including undefined synthesized attributes and unlike the slice based method, the user only needs to look at a small part of the parse tree. This can be realized by the integration of both of the pure algorithmic debugging and the slice based debugging.

Reduce questions on big trees

This is relevant to the previous item. In the previous synth-function based method, a subtree that is a premise of a query should be a complete subtree with all descendant nodes. On the other hand, in the proposed method, trees in a query may be incomplete, in the sense that some part of the subtrees may be pruned.

Identification of bug

When employing the previous algorithmic debugging, semantic rules identified as erroneous are not a single rule in general. In the example attribute grammar of this experiment, there is a case that four semantic rules remain as candidates of a bug. On the other hand, the proposed method can identify just one semantic rule as erroneous.

These are advantages of our debugging method. Some items can be realized in the previous algorithmic debugging by ad-hoc extension. But our method is advantageous because these features could be realized in a single framework.

User Interface

It is important for the query based system to give comprehensive questions to the user. Aki implements several mechanisms for helping the user understand questions. For example, for questions about values with a big data structure, Aki can display the difference of the values, which helps the user intuitively understand the value. In the example AG, if a question is about relationship of two symbol table values, different part is highlighted.

7 Concluding Remarks

This paper shows a systematic debugging method of AGs. We generalized the algorithmic debugging of AGs, which enables to allow various forms of questions and unifies the previous two methods. We also developed a new debugging method using the proposed framework. This method is implemented in debugger Aki and experimental result is shown.

We point out some researches related to ours in the following.

- Declarative debugging scheme [7] is a scheme that generalizes the algorithmic debugging. This scheme is applicable to various language paradigm and can treat several classes of bug. This means the scheme includes our proposed method. But that paper does not show which kind of computation can be the subject of questions to the user and cannot be applied directly to the automated debugging of attribute grammars. On the other hand, we clarified that the attribute computation composition can be the base for questions, and it easily led to the new debugging method.

- Interprocedural slicing [3] is a technique to compute a slice considering interprocedural dependencies. GADT [11] is a system that applies algorithmic debugging to a procedural programming language and exploits the interprocedural slicing. The debugging algorithm shown in Section 4.6 can be thought as an application of this technique in some sense that when the user indicates a wrong inherited attribute of a synth-function the program slicing with respect to that value is computed interprocedurally.

- "Assertion" is a feature to be satisfied in a program and it is used for effective localization of implementation faults. In a debugging point of view, when an assertion is given for a function, a behavior that satisfies the assertion is ensured to be correct. Thus, we can omit the query for this behavior and the number of queries can be reduced. EGADT [2] is a system that incorporates assertion into the algorithmic debugging of a programming language. Assertion, as an extension of algorithmic debugging, aims to reduce the number of queries. On the other hand, our method aims at understandable question by enabling to extract functional behavior in attribute evaluation in various forms.

As future work, we should develop a more effective method using the generalized algorithmic debugging proposed in this paper. We will also investigate combination of the proposed method and other debugging methods such as assertion as well as the user interface of the debugger for easier questions to answer.

References

[1] Appel, A. W.: *Modern Compiler Implementation in Java (1st ed.)*, Cambridge University Press (1998).

[2] Fritzson, P., Auguston, M. and Shahmehri, N.: Using Assertions in Declarative and Operational Models for Automated Debugging, *Journal of Systems and Software*, No. 25, pp. 223–239 (1994).

[3] Horwitz, S., Reps, T. and Binkley, D.: Interprocedural slicing using dependence, *ACM Trans. Prog. Lang. Syst.*, Vol. 12, No. 1, pp. 26–60 (1990).

[4] Ikezoe, Y., Sasaki, A., Ohshima, Y., Wakita, K. and Sassa, M.: Systematic debugging of Attribute Grammars, *Proceedings of the Fourth International Workshop on Automated Debugging AADEBUG2000*, pp. 235–240 (2000).

[5] Korel, B. and Laski, J.: Dynamic Slicing of Computer Programs, *Journal of Systems and Software*, Vol. 13, pp. 187–195 (1990).

[6] Mayoh, B. H.: Attribute Grammars and Mathematical Semantics, *SIAM Journal on Computing*, Vol. 10, No. 3, pp. 503–518 (1981).

[7] Naish, L.: A Declarative Debugging Scheme, *Journal of Functional and Logic Programming*, Vol. 1997, No. 3 (1997).

[8] Sasaki, A., Ikezoe, Y. and Sassa, M.: Systematic Debugging Method for Attribute Grammars, *Transactions of Information Processing Society of Japan "Programming"* (in Japanese), Vol. 23, No. SIG3(PRO 14), pp. 1–15 (2002).

[9] Sassa, M.: Rie and Jun: Towards the Generation of all Compiler Phases, *Proceedings of the 3rd International Workshop on Compiler Compilers, LNCS 477*, Springer-Verlag, pp. 56–70 (1991).

[10] Sassa, M. and Ookubo, T.: Systematic debugging method for attribute grammar description, *Information Processing Letters*, Vol. 62, pp. 305–313 (1997).

[11] Shahmehri, N.: *Generalized algorithmic debugging*, PhD Thesis, Linköping University (1991).

[12] Shapiro, E. Y.: *Algorithmic Program Debugging*, The MIT Press (1982).

[13] Shimomura, T.: Critical Slice-Based Fault Localization for Any Type of Error, *IEICE Transactions on Information and Systems*, Vol. E76-D, No. 6, pp. 656–667 (1993).

[14] Weiser, M.: Program slicing, *IEEE Transactions on Software Engeneering*, Vol. 10, No. 4, pp. 352–357 (1984).