Systematic Debugging of Attribute Grammars

Yohei Ikezoe, Akira Sasaki, Yoshiki Ohshima, Ken Wakita, and Masataka Sassa
Tokyo Institute of Technology
{Yohei.Ikezoe, sasaki, ohshima, wakita, sassa}@is.titech.ac.jp

Abstract

Although attribute grammars are commonly used for compiler construction, little investigation has been conducted on debugging attribute grammars. The paper proposes two types of systematic debugging methods, an algorithmic debugging and slice-based debugging, both tailored for attribute grammars. By means of query-based interaction with the developer, our debugging methods effectively narrow the potential bug space in the attribute grammar description and eventually identify the incorrect attribution rule. We have incorporated this technology in our visual debugging tool called Aki.

1 Introduction

The *attribute grammar* (AG) is a formal framework to express both syntax and semantics of programming languages [4]. An attribute grammar description comprises a set of productions (BNF rules) and a set of attribution rules defined over the attributes associated with the grammar.

Attribute grammars are easy to describe and understand because they describe "what the programming language semantics is like" but not "how their attribution rules are actually implemented." An *AG-based compiler-compiler* takes an attribute grammar description of a programming language and generates an efficient compiler for it. Attribute grammar has been successfully used to describe various programming languages and their processors.

On the other hand, debugging an attribute grammar is not simple. Debugging an attribute grammar description using a standard debugger exposes the attribute grammar implementation such as the attribute evaluator, which is usually a program generated from the attribute grammar description, and the runtime representation of attributes and parse trees.

The paper proposes AG-aware debugging techniques for attribute grammars. By "AG-aware" we mean that the debugger is aware of attribute grammars and thus debugging does not necessarily require knowledge about strategy and implementation of attribute evaluation. We have formerly applied an *algorithmic debugging* [10] technique to attribute grammar [9]. This paper is on the same line but further incorporates *slice-based debugging* technique as well. By means of query-based interaction with the compiler developer, both techniques effectively narrow the potential bug space in the attribute grammar description and eventually identify the incorrect attribution rule.

The benefit of our approach, independence from the implementation of particular AG-based compiler-compiler, is twofold: (1) the compiler developer is freed from understanding implementation of the compiler-compiler and (2) the proposed technique can be applied to other AG-based system.

Our hybrid debugging technique has been implemented as a visual debugging environment called Aki. It is written in Squeak Smalltalk [3]. The resulting system has been used in our project that incorporates AG technologies in most phases of compiler construction — including transformation of intermediate code, optimization, and code generation [8].

The rest of this paper is as follows. Section 2 briefly introduces the attribute grammar and attribute evaluation, sections 3 and 4 explain the two debugging techniques, section 5 describes the visual debugger, section 6 discusses the debugger, and section 7 concludes this article.

F ::= . L
 { $L.pos = 1$;
 $F.val = L.val$ }
L_0 ::= B L_1
 { $L_1.pos = L_0.pos + 1$;
 $B.pos = L_0.pos + 1$; (bug)
 $L_0.val = B.val + L_1.val$ }
 | B
 { $B.pos = L_0.pos$;
 $L_0.val = B.val$ }
B ::= 1
 { $B.val = 2^{-B.pos}$ }
 | 0
 { $B.val = 0$ }

Figure 1: An example of attribute grammar

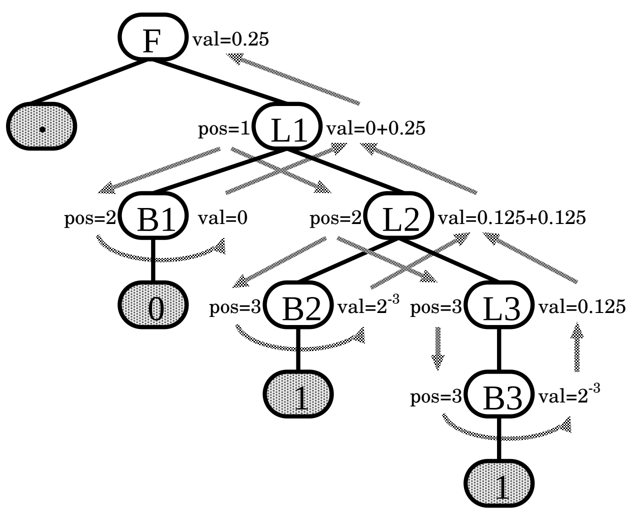

Figure 2: Attributed parse tree

2 Attribute grammars

The attribute grammar [4] is a description that provides both syntax and semantics of an input string. It is one of fundamental system used for formalization and construction of compilers.

Fig. 1 is an example of an AG description that evaluates binary representation of numbers. An attribute evaluator generated from the AG description takes an input string (e.g., ".011") and parses it in accordance with the syntax definition of the grammar. Then the attribute evaluator computes attribute values for each node in the parse tree according to the dependency among attributes and associates them to the respective node. In this manner, an *attributed parse tree* is created from the input string (see Fig. 2). The shaded arrows represent dependency between attributes.

There are two kinds of attributes. One is called *inherited* attribute, whose value is computed from the values of the attributes associated with the ancestor and sibling nodes. The other is called *synthesized* attribute, whose value is computed from the values of the attributes associated with the children nodes.

We have intentionally introduced a bug in the AG description for the sake of discussion in the following sections. Because of this, some of the attribute values in Fig. 2 are wrong.

3 Application of algorithmic debugging to attribute grammar

Algorithmic debugging [10] is a systematic bug locating technique. With programmer's guidance, an algorithmic debugger locates a bug in program execution. Algorithmic debugging formulates execution in terms of *computation tree* which is defined as recursive logical deduction of logic programming, or recursive β-reduction for functional programming. The debugging method has been applied to functional [7] or procedural language [1].

To apply algorithmic debugging to attribute grammar paradigm, we need to formulate attribute evaluation as some form of recursive application. In [9] we have shown that the notion of *Synth function* [6] suits for this purpose.

It is known that for any synthesized attribute s of a node N in the parse tree, its value $N.s$ is uniquely defined using the Synth function $F_{N.s}$:

$$N.s = F_{N.s}(N.I_{N.s}, tree_N).$$

where $tree_N$ stands for the subtree of the parse tree rooted at node N and $N.I_{N.s}$ for a set of inherited attributes of N on which $N.s$ directly or indirectly depends in $tree_N$ [9].

Because the entire attribution for the parse tree can be represented as recursive application of Synth functions, we can use Synth functions as basis for formulating computation tree for attribute grammars.

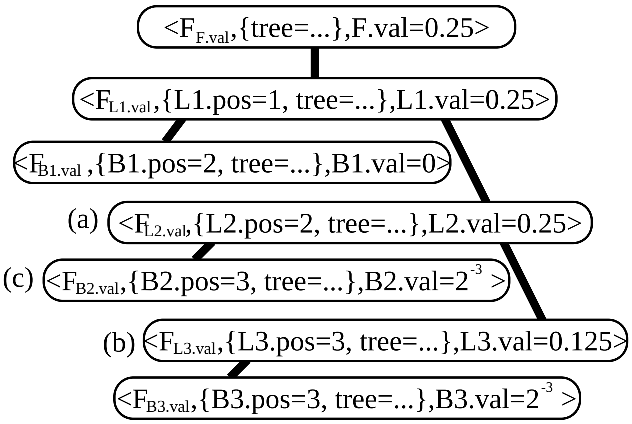

Figure 3: Computation tree

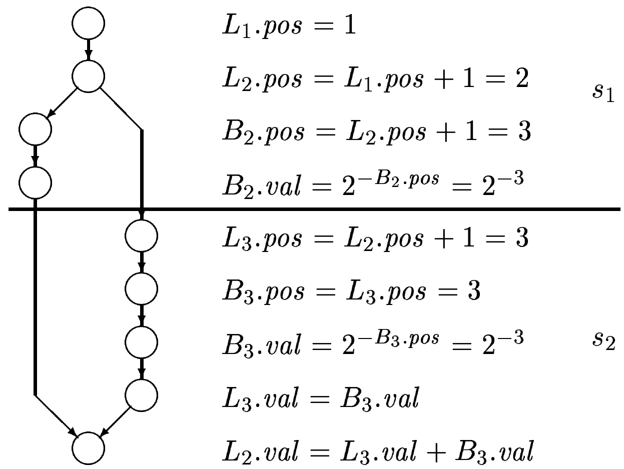

Figure 4: Slice partitioning

We will explain how the bug in Fig. 1 can be detected by the algorithmic debugging. When a string ".011" is given as an input to the description, the computation tree in Fig. 3 is created. First the debugger chooses node (a) in this figure, and queries the user whether $L_2.val = 0.25$ is correct with respect to the argument $L_2.pos$ and the subtree rooted at L_2. This query means whether the binary number "11" from the second decimal place represents 0.25, and the user can respond that this is incorrect because the correct value is 0.375. Given this, the debugger prunes the computation tree above (a) from the search space. Then the debugger queries similar question for node (b). This time the value is correct and the subtree rooted at (b) is pruned. Next the debugger queries similar question for node (c). Finally the debugger locates the attribution rule for node (a) as containing a bug.

As for the applicability of the method to classes of AG, [9] gives how to apply algorithmic debugging to noncircular AG, absolutely noncircular AG, simple multi-visit AG and to its subclasses.

4 Slice-based debugging

Techniques that utilize the notion of *program slices* have been applied to program verification, program testing, version management, and systematic program debugging. Shimomura proposed a systematic debugging method of procedural programming languages using slices [11]. We applied this idea to the attribute grammar framework.

The *dynamic program slice* of an execution of a statement s in a program is a set of all the statements upon which s depends, directly or indirectly [5]. In attribute grammars, attribute evaluation of a given parse tree can be considered to be a sequence of evaluation of node attributes, and we can define dynamic program slices for a given attribute grammar description and its input program. For example, the sequence on the right side in Fig. 4 is a possible attribute evaluation sequence of $L_2.val$ for AG in Fig. 1 and the input ".011." The directed graph on the left side illustrates direct dependency among the attributes. The slice for a given attribute (instance) is a set of attributes upon which the attribute depends.[1] For example, the slice for $L_3.pos$ is a set of attributes upon which $L_3.pos$ directly or indirectly depends, namely, $L_2.pos$ and $L_1.pos$.

Our debugging strategy works as follows. Suppose that we already know that a slice contains an incorrect attribution that is triggered by an unknown bug. The debugger partitions the attribute evaluation sequence into arbitrary two subsequences (e.g., s_1 and s_2 in Fig. 4). Then the debugger queries the user about correctness of evaluation. This is accomplished by asking the correctness of the attribute values crossing the boundary between the two subsequences (e.g., correctness of $B_2.val$ and $L_2.pos$). If one of the crossing at-

[1] In AG, the slice can be given for an attribute instance rather than for evaluation of each attribute, because each attribute is evaluated only once.

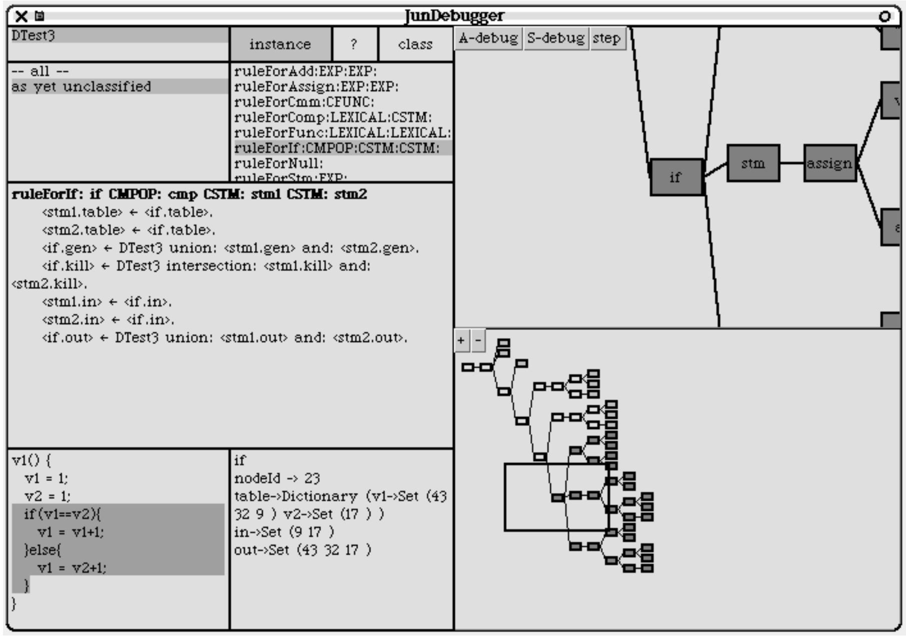

Figure 5: Example of execution of Aki

tribute values turns out to be incorrect, then the debugger identifies that the error was triggered in the subslice of that attribute. Otherwise, when all the crossing attribute values are correct, then the debugger excludes the subsequence s_1 from its bug-locating search space.

For example in Fig. 4, value assignment to $B_2.val$, 2^{-3} is different from its expected value, 2^{-2} ($B_2.val$ stands for the value of binary number $(0.01)_2$). Given this information from the user, the debugger understands that the bug inhabits somewhere in the subslice of $B_2.val$. Next, the debugger divides the subslice of $B_2.val$ and queries whether the value of $L_2.pos = 2$ is correct. The user answers "yes". Then the debugger narrows the search space to $B_2.pos$ and $B_2.val$, and queries whether the value of $B_2.pos = 3$ is correct. Since the user says it is incorrect, the debugger can locate the bug to a single attribute $B_2.pos$, and it identifies that the attribution rule "$B_2.pos = L_2.pos + 1$" contains a bug.

5 Aki: the debugger

The Aki visual debugger is a part of our AG-based compiler construction system that comprises a compiler frontend generator called Rie and backend generator called Jun[8]. Aki is used to locate

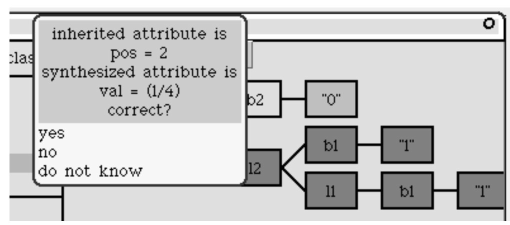

Figure 6: Example of query of algorithmic debugging

bugs in the compiler backend description. The two systematic debugging techniques explained in earlier sections are incorporated in Aki. Because Jun accepts fairly large class of attribute grammars, this approach can be applied to other AG evaluators [9].

When the compiler developer finds that the compiler generates an incorrect code sequence for some source program, then he/she can supply both the compiler backend description and the source program to Aki for debugging. Fig. 5 shows a screen shot of a debug session using Aki. The panes presents attribution rules, input source program supplied to the debugged compiler, values of attributes, and the parse tree of the source program in several forms. These panes work cooperatively: user's interaction with one pane is reflected to others.

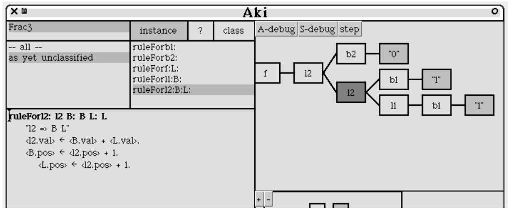

Figure 7: The screen shot when the bug is located

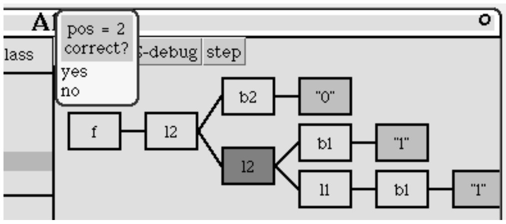

Figure 8: Example of query of slice-based debugging

The user can choose systematic debugging strategies with buttons: A-debug button for algorithmic debugging and S-debug button for slice-based debugging.

The algorithmic debugger chooses an arbitrary node from the computation tree and ask the user about correctness of the respective computation. In Fig. 6, the debugger is asking about correctness of the computed attribute value (“$val = 1/4$”). Aki helps the user answer this question by highlighting the respective subtree of the parse tree in dark-gray and showing inherited attribute values given as inputs to this computation (“$pos = 2$”). The user’s answer to this question is used by the debugger to narrow the search space for erroneous code in the program. This narrowing is repeatedly applied until the debugger eventually locates the erroneous attribution rule (in Fig. 7, Aki successfully located a bug in ruleForl2 rule.).

The user starts slice-based debugger pointing it an incorrect attribute value in the attributed parse tree. The slice-based debugger systematically locates the source of the problem in the attribution rules. The debugger computes the slice for the incorrect attribute and partition the slice into two subslices. Then the debugger queries if incorrect attribute values are passed from one subslice to another. This is accomplished by asking separate questions for these attribute values, respectively. In Fig. 8, Aki is asking about correctness of the value of one of those attributes. Given answers to these questions, the debugger narrows the search space for erroneous code into one of the subslices and eventually locate the incorrect attribution rule.

	algorithmic debugging	slice-based debugging	without Aki
number of queries	9	13	NA
time to locate bugs (min.)	3	3	15

Figure 9: Result of a user test

An advantage of slice-based debugging it can debug a program that failed to complete in the middle of its execution. On the other hand, it is impossible to create a computation tree for the entire computation of such program and hence difficult to apply algorithmic debugging.

6 Experiments and discussion

6.1 Evaluation of Aki

Aki has been used to debug descriptions of several modules in our complier construction project for the C language; examples of modules are SSA-conversion and liveness analysis. Users report it is easier to find bugs by using Aki than by using conventional approach.

We have done user test to evaluate usefulness of Aki. Three experienced compiler programmers are chosen from our team as subjects. They are shown a source program of some compiler module. We have included a typical mistake in the source program. Then the programmer is asked to locate using algorithmic debugging feature of Aki, using slice-based debugging feature of Aki, or without support by Aki. The 123 lines long test program contains 18 attribution rules. Given a sample C program that triggers the compiler’s bug, the parser generates a parse tree of 77 nodes and 171 attribute instances.

Fig. 9 shows how the choice of debugging method affects the time to locate bugs. The numbers imply effectiveness of the systematic debugging approach for AG-based compiler construction, at least for smaller sized modules.

6.2 Discussion

A known problem in algorithmic and automatic debugging is that the user has to understand the question and answer it correctly ([1] [7] [2]).

For example in the algorithmic debugging of AGs, the user may have difficulties in checking whether the behavior of a Synth function or the values of attributes are correct. This is due to the fact that the user has to look at the subtree in the parameter of the Synth function, and furthermore attribute values may be a set and may have many elements in compilers.

These problems have been partly solved from the algorithmic point of view by extending the debugging algorithm in several situations [8]. For example, the extended algorithm can deal with the case when the user find the value of the inherited attribute — given as a premise of a query by the system — is itself wrong, or when the user cannot reply to a query with confidence. Some part of the extended algorithm is implemented in Aki.

On the other hand, from the implementation point of view, we made several efforts in making Aki so that the user can easily grasp the attribute values and the subtree in a query. For example, Aki can show the source code corresponding to any subtree. When a new attribute value is computed by combining several attribute values with some operation, Aki highlights the part of the original attribute values within the new attribute value by a separate color, making the difference of both attributes clear.

However, we think further improvement in the algorithm and the implementation is necessary.

7 Conclusion

This paper has presented two systematic debugging techniques for attribute grammars, one is based on the algorithmic debugging and the other is based on the program slicing. These techniques have been incorporated in our high-level visual debugging tool called Aki.

There remain some issues that require further investigation. Currently the two systematic debugging features are provided as separate technologies. We plan to integrate the two techniques and search for more effective debugging methodology.

References

[1] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Prog. Lang. Syst.*, 1(4):303–322, 1992.

[2] Peter A. Fritzson, editor. *AADEBUG'93, LNCS 749*. Springer-Verlag, 1993.

[3] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future—The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*, pages 318–326, 1997.

[4] D. E. Knuth. Semantics of context-free languages. *Math. Syst. Th.*, 2(2):127–145, 1968. correction: Vol. 5, No. 1, pp.95–96.

[5] B. Korel and J. Laski. Dynamic slicing of computer programs. *J. System Software*, 13:187–195, 1990.

[6] B. H. Mayoh. Attribute grammars and mathematical semantics. *SIAM Journal on Computing*, 10(3):503–518, 1981.

[7] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *J. Functional Programming*, 4(3):337–369, 1994.

[8] M. Sassa. Rie and Jun: Towards the generation of all compiler phases. In *Proceedings of the 3rd International Workshop on Compiler Compilers, LNCS 477*, pages 56–70. Springer-Verlag, 1991.

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

[10] E. Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1983.

[11] T. Shimomura. Critical slice-based fault localization for any type of error. *IEICE Transactions on Information and Systems*, E76-D(6):656–667, 1993.