Debugger for Attribute Grammar Specification Language, Silver

Yogesh Mali
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455, USA
yomali@cs.umn.edu

1 Abstract

Despite the growth of information technology, it has not provided satisfactory results in certain areas. This is because our current approach has been limited by time consuming processes of software testing and software development. Programming, specially declarative and functional, can improve this methodology by making programmers free from following certain sequence of expression calculation. Particularly, they also provide the better safety than the current programming language in use. With advancement of new languages, the tools used for developing languages has remained unattended mostly. Attribute grammars are one of those tools. Attribute grammars has provided a new paradigm to programming languages for their design and development. Current research in attribute grammars has focused not just on utilizing them for language development but also to use the underlying architecture efficiently [3]. Unfortunately, the complexity of attribute grammars has deterred many users. One of the reasons is non-existence of a good debugger which generally helps programmer to understand what went wrong and where. The proposed research addresses the development of a debugger for attribute grammar specification language, Silver using algorithm based debugging which will involve user interaction. The results will facilitate novice users to use Silver effectively and experienced users to use it efficiently.

2 Introduction

Debugging: Debugging facilitates program testing and minimizes the possibilities of errors in a program. There are two problems in functional programming 1) how to turn programs into efficient computations 2) how to trace program errors. During 80-90s, the research in the domain of programming languages was concentrated in developing new languages, their optimizations, evaluation of different paradigms of programming and debugging in general purpose languages. Debuggers for functional programming, and in particular attribute grammars haven’t got adequate attention. So, in general the research has remained limited to certain programming communities. The lack of debugging tool has
kept the scope of attribute grammars restricted. The major problems in attribute grammar area are the circular attribute grammars, the large size of syntax trees which ultimately depends on the size of grammar and the complexity of attribute dependencies. These problems make the tracing of computation difficult.

**Solution:** Therefore, I propose a debugger for attribute grammar specification functional language, Silver. The aim of this research will be the development of a limited prototype tool for silver debugging. It will solve some of the problems, to name a few, tracing of computations of syntax trees decorated with attributes, detecting circular grammars using tracing and easing software testing process. This will allow users to concentrate on the intended semantics of functions rather than operational issues such as evaluation order.

3 Proposed Research

3.1 Background

Despite the development of Silver, the language for specifying attribute grammars which help to generate compilers, its scope has remained limited. The same can be said with ANTLR group, so this is the general notion with attribute grammars. The complexity and difficulty in understanding specification has hindered its progress among developers. Actually there should not be any need for debugger, in general. Many errors which are hard to find at run-time type checking, are caught by static type checking. However, not all errors can be caught statically. Silver poses the same problem. At times Silver does not provide very clever error messages, specially while using its advance constructs. While using pattern matching in Silver, if the programmer does not use it efficiently and ignore the *don’t care* case of pattern for error catching, he gets the run time error ‘*Error: Pattern matching failed*’ in case there is some problem with grammar. The error message is not sensible in the sense that it does not provide line number information or any other information which will aid programmer to remove the problem. In such scenario, programmer has to go through all code and have to search where he used pattern matching. In short, this can lead to a time consumption if the grammar specification is huge in size.

3.2 Attribute Grammars

To get the sense, Figure 1 presents the famous binary to decimal conversion grammar [9]. Attribute grammar helps to integrate both syntax and semantics of the language. Semantics of languages can be defined using context-free grammars. The figure 2 represents specification of the grammar in figure 1 in Silver.
3.3 Silver

The specification written in Silver, generally consists of declarations for non-terminals, terminals, productions and attributes. Specification of the grammar may be spread across several files. First line of any file generally indicates the grammar name. Figure 2 shows the specification for binary to decimal conversion. The grammar name is `edu:umn:cs:melt:sandbox:knuth`. This is followed by nonterminal, terminal and attribute declarations. Productions are specified with production name and actions on the attributes defined over non-terminals. In the grammar shown, `Root, B, N, L` are nonterminals and `0, 1, ,` are terminals. The concrete productions are used to generate the input specifications for a parser generator. The first production `root` is having `Root` as its left hand side which goes to `N`. The attribute definition has been specified in the curly braces `{,} `. For this simple specification, I have mainly defined concrete productions, but in more complex grammars, concrete and abstract syntax can be separated so that the attributes in concrete productions can be used to build the AST. Abstract productions can be used to error checking and semantic analysis in the language since abstract syntax is free of ambiguities. In general, silver demonstrates attribute computation.

Attribute grammars help to calculate the semantics by evaluating the attributes used in the language grammar. Silver supports various advance features. `forwarding` is used by the productions which define new language extensions to specify the semantically equivalent construct in the host language that they will translate to. `collection` attributes helps to build extension modules that just does not require local transformation, but also requires global transformations. Operator overloading is the one example where this is particularly used. `pattern matching` is the new feature which helps to examine higher order attribute grammars data from outside.

To demonstrate why we need a debugger for silver, we will discuss a problem one might face with the grammar specified above in figure 1. The figure 2 represents the correct silver specification for the binary to decimal conversion grammar. On compiling this grammar and using the executable generated, one can get the decimal value of any binary number. For example, for input 111, the executable will give an output of 7.0. In the specification, concrete syntax is used for generating scanner and parser. Abstract syntax then parses the input
grammar edu:umn:cs:melt:sandbox:knuth;
nonterminal Root,B,N,L;
terminal '0', '1', '.';
synthesized attribute value::Float occurs on B,N,L;
synthesized attribute len::Integer occurs L;
inherited attribute scale::Integer occurs on L,B;

concrete production root
top::Root ::= n::N
{ top.value = n.value;}
concrete production decimal
n::N ::= l1::L '.' l2::L
{ n.value = l1.value + l2.value; l1.scale = 0;
  l2.scale = -l2.len; }
concrete production multiplication
l::L ::= l1::L b1::B
{ l1.value + b1.value; b1.scale = l1.scale;
  11.scale = 11.scale + 1; 11.len = 11.len + 1; }
concrete production nTOl
n::N ::= l::L
{ n.value = l.value; l.scale = 0;}
concrete production lTOb
l::L ::= b::B
{ l.value = b.value;
  b.scale = l.scale; l.len = 1; }
concrete production bZero
b::B ::= '0'
{ b.value = 0.0; }
concrete production bOne
b::B ::= '1'
{ b.value = if (b.scale < 0) then 1.0/poworOf(-b.scale) else poworOf(b.scale); }
abstract production main
m::Main ::= args::String
{ m.ioOut = print(‘
’ + toString(parse(args).value) + ‘
’, m.ioIn); }
function poworOf
Float ::= n::Integer
{ return if (n==0) then 1.0 else 2.0 * poworOf(n-1); }

Fig. 2. Attribute Grammar for binary to decimal conversion in Silver
string for which parse tree is developed. Attribute evaluation process happens according to attribute dependencies. Figure 4 represents the attribute evaluation process.

Now replace the production multiplication in the specification as per figure 3. Silver compiler will compile in normal way and there will not be any errors. Even while using the executable, there will not be any errors. But the executable now will give output as 5.0 for input 111 instead of 7.0 which is not correct.

```concrete production multiplication
l::L ::= l1::L b1::B
{l.value = l1.value + b1.value; b1.scale = l.scale + 1;
 l1.scale = l.scale; l.len = l1.len + 1; }
```

Fig. 3. Same code as in figure 2 with bug

Here we will see how this production makes the overall value of input string wrong. Algorithmic debugging will help us to find out this bug. So with respect to specification, there is nothing wrong. But the bug lies in attribute computation which needs to be identified. scale is an inherited attribute. In the production multiplication, scale of B is calculated by adding scale of its parent node and 1 which is not correct, as obvious. The algorithmic debugging presented in next section will try to find out the bug in this production.

### 3.4 proposed approach

To find out the bug, we can use the approach of algorithmic debugging [13], and a tracing technique. Query based algorithmic debugging can help us to find out the bug easily in the grammar. This approach has been borrowed from logic programming.

Basically a new algorithm is to be proposed where for each subtree, the computed attribute values will be presented. The debugging method will use this recursive algorithm to which whole attribute evaluation tree will be an input. For base case, the algorithm will just return an empty set of rules. But at each recursion step, algorithm will inquire about accuracy of calculated attribute values to user. Since Silver generates the compiler, it is very hard to have an interactive session. Here is the proposed idea for implementation:

1. Use the Haskell translation code generated by Silver Compiler.
2. Compile the Haskell translation using haskell compiler and use the tracing technique to generate the trace for attribute values.
3. User interface will use this trace as its input to interact with user. User will answer “Yes” or “No” based on the correctness of computed attribute values.
(A) \( \text{top.value} = \text{n.value}\{7\} \) \{\text{Root ::= N}\}

(B) \( \text{n.value} = \text{l.value}\{7\} \) \{\text{N ::= L}\}
\( \text{l.scale} = 0 \)

(C) \( \text{l.value} = \text{l1.value} + \text{b1.value}\{7\}\) \{\text{production L ::= L B}\}
\( \text{l1.scale} = \text{l.scale} + 1\{ 1 \} \)
\( \text{b1.scale} = \text{l.scale}\{ 0 \} \)
\( \text{l.len} = \text{l1.len} + 1 \)

(D) \( \text{b1.value} = \text{powerof(b1.scale)}\{ 1 \}\) \{\text{production B ::= 1}\}

(E) \( \text{l2.value} = \text{l1.value} + \text{b3.value}\{4 + 2 = 6\}\) \{\text{production L ::= L B}\}
\( \text{l1.scale} = \text{l.scale} + 1\{ 2 \} \)
\( \text{b3.scale} = \text{l1.scale}\{ 1 \} \)
\( \text{l2.len} = \text{l1.len} + 1 \)

(F) \( \text{b3.value} = \text{powerof(b3.scale)}\{ 2 \}\) \{\text{production B ::= 1}\}

(G) \( \text{b4.value} = \text{b4.value};\{ 4 \}\) \{\text{production L ::= B}\}
\( \text{b4.scale} = \text{b4.scale};\{ 2 \} \)

(H) \( \text{b4.value} = \text{powerof(b4.scale)}\{ 4 \}\) \{\text{production B ::= 1}\}

Fig. 4. Attribute Evaluation for input string 111
4. User interface will recursively ask the questions to reduce the localization of bug space, eventually leading to the production rule where the bug is located.

To understand how the query based algorithm will work, consider the example presented previously. We have modified the production multiplication by introducing a bug. The attribute rule for \textit{11.scale} has been changed. For explanation purpose, the attribute rules for different productions have been named with capital letters as if they are nodes of a tree. Figure 4, represents the correct evaluation of attributes. Input string is 111. At first node, interface will ask the value of \textit{top.value(5)} which is actually not correct. User will answer “No”. So it means the child value on which top depends, is itself wrong. Eventually, we will come to node C. At this node, we will check the value of all four attributes. 

Value of \textit{1.value} is 5 which is incorrect. The value of \textit{b1.scale} is 1 which is once again incorrect, but debugger will come to know only when we get the value of \textit{b1.value}. This is because of the dependency of \textit{b1.value} on \textit{b1.scale}. At node D, we will get the value as 2 which is clearly wrong and debugger will mark the calculation of \textit{b1.scale} as wrong.

The implementation of user interface will be carried out in TclHaskell. TclHaskell is the library developed by Chris Dornan and is mainly used for writing platform independent graphical user interfaces in Haskell. The main reason for choosing the haskell for implementation is to demonstrate the strength of functional languages and since the input trace to user interface will be in Haskell, there will not be any need of any other translation.

Following are the important points which need to be considered for this research:

– Scope - First we must find out the grammar sizes which the debugger can support to find out a bug in initial prototype. \textit{Research Questions} - Are we going to consider circular attribute grammar in initial prototype? What are the different error messages of Silver which are not intelligent enough to convey programmar where the errors are?

– Performance - Since it is difficult to estimate the syntax tree sizes from just looking at grammar, the performance of the debugger should be critical for later prototypes. \textit{Research Questions} - What are the methods to determine syntax tree sizes?

3.5 Related Work

Sasaki and Sassa implemented debugger for attribute grammar \cite{1}, but hasn’t provided enough details of implementation to justify the scalability of the debugger. Sparud \cite{15} implemented the tracing technique for debugging in haskell. In this debugger, the main idea was structure of program rather than order of evaluation. First phase of the debugger generates the evaluation dependency tree. The second phase helps user to navigate this dependency tree. Tolmach \cite{16} developed a debugger for Standard ML using source code instrumentation technique.
used the concept of observation of intermediate data structures to implement a portable debugger for Haskell. The observation of intermediate data structures is on similar lines of tracing technique where we trace everything from beginning to end. Chitiil, Runciman and Wallace [4] provided the comparison of three debuggers Freja, Hat and Hood for Haskell. Ennals and Jones [6] developed HsDebug, a debugger for Haskell using the conventional approach of “stop, examine and continue”. The detail study of attribute grammars and Silver will be found in [12, 17].

4 Time Allocation

Initial prototype should take around three months for one person depending on his understanding of attribute grammar and languages silver and haskell. The estimation of time detailed here is rough and it may vary accordingly and is estimated depending on the resources available. Here I present the weekwise division of the whole project.

1. Week 1 - Analysis of different grammars and try to find out the different static and runtime errors which actually do not provide very sensible error messages.
2. Week 2 - Study of Haskell translation and tracing techniques.
3. Week 3 - Implementation of tracing in Haskell translation and testing it for simple grammars. Verification that we get needed values of all attributes.
4. Week 4 - Analysis of different user interfaces which are developed using TclHaskell.
5. Week 5 - Small prototype implementation of user interface and its testing.
6. Week 6 - Implementation of query based algorithm in user interface.
7. Week 7 - Implementation will continue.
8. Week 8 - Implementation to be finished.
10. Week 10 - Demonstration of complete debugger with some large examples.
11. Week 11 - Writing technical report
12. Week 12 - Finish technical report and sign off.

5 Challenges

1. Silver is used for generating compilers. Once the compiler is generated, it is used to parse the language file. Extensible compilers basically help to find the static errors which are not possible in underlying main compiler and this is done by extending the host language. So extensible compiler will parse the program with new extension and will verify it for all static errors. If successfully parsed, it will transform the extensible program into host language syntax which will be used by the compiler of host language. During the compilation, silver compiler checks for binding errors, type errors, generate the scanner and parsers and finally generate the haskell translation
which is used by haskell compiler for generating executable. The main challenge here will be making the debugging session interactive with breakpoints or watchpoints. The above suggested approach sounds more feasible at the moment.

2. The problem of circular attribute grammars has remained unresolved. While Silver compiler errors out circular attribute grammars, how the debugger will take care of this, will be one of the challenges.

3. Since the debugger is planned to be implemented in Haskell, the performance and scalability will be other challenges.

6 Resources

The project will be done 20 hours per week by a single student for duration of three months.

References

