Tool demo: DrAST

An inspection tool for attributed syntax trees

Joel Lindholm Johan Thorsberg Görel Hedin
Department of Computer Science, Lund University, Sweden
joel.lindholm@gmail.com, johanthorsberg@gmail.com, gorel.hedin@cs.lth.se

Abstract

When implementing a language by means of attribute grammars, it is often useful to study example programs and their attributed trees, to understand the compiler structure, or for debugging. DrAST is a tool that allows interactive inspection of attributed trees. It is interfaced to the JastAdd metacompiler, and supports all its attribution mechanisms, such as demand evaluation, reference attributes (graph edges), and nonterminal attributes. A challenge in visualizing attributed trees is that they are large, even for small programs. To allow the user to focus on the aspects of interest, DrAST supports the interactive definition of filtered versions of the tree through a domain-specific language which allows conditional filtering based on the attributes themselves. We have used DrAST on a variety of language implementations, from tiny compilers used in teaching to a complete Java compiler.

Keywords abstract syntax trees, attribute grammars, debugging tools

1. Introduction

Attribute Grammars (AGs) extend context-free grammars with attributes defined declaratively using directed equations (Knuth 1968). The attributes are used for defining context-dependent static properties of syntax nodes, for example, name bindings, types, error messages, and code for a target machine.

Knuth's AGs supported attributes defined by equations either in the node itself (*synthesized* attributes), or in the parent (*inherited* attributes). Current AG tools, e.g., JastAdd (Ekman and Hedin 2007a), Silver (Wyk et al. 2010), and Kiama (Sloane et al. 2010), typically use a number of additional attribute mechanisms. For example, *reference* attributes turn the tree into a graph (Hedin 2000) and *higher-order* attributes can themselves be attributed trees (Vogt et al. 1989). AGs with these extended mechanisms can be used for generating production quality compilers. Examples include ExtendJ (previously known as JastAddJ) which is an extensible Java compiler (Ekman and Hedin 2007b), and JModelica.org which is a Modelica platform developed by the company Modelon AB (Åkesson et al. 2010). Both are developed using the JastAdd tool.

The core data structure in an AG-based compiler is an attributed abstract syntax tree. Each syntax node of a certain kind has a particular set of attributes, and their values are defined using directed equations whose right hand sides are functions that depend on the values of other attributes.

To understand an AG, it is useful to look at example programs and their corresponding attributed trees. This is important in many situations: when learning how an existing AG works, when debugging an AG (for example, if the equations are not correct), and not the least in teaching, when students learn about abstract syntax and AG concepts. We have developed the tool *DrAST* to support these situations for JastAdd AGs. DrAST is an interactive tool, supporting interactive exploration of attributed abstract syntax trees. It is open source, and available at `https://bitbucket.org/jastadd/drast`.

A challenge in visualizing the attributed tree is that it is large. Even for a small example program, there can be hundreds of tree nodes and thousands of attributes. Showing all the information would just be overwhelming and not very useful. Typically, the user is only interested in some parts of the attributed tree, like the nodes of certain types, and only some of their attributes. For example, the user might be interested in viewing all variable uses and their name binding attributes. The interesting part could also involve nodes with certain properties, for example, all undeclared uses of names. In general, we can think of these interesting parts as a *filtered* version of the attributed tree, where only the interesting parts remain.

Typically, the user may be interested in viewing several different programs through the same filter, or use the same filter for several consecutive versions of a compiler, so there is a need for storing the filter definition between runs.

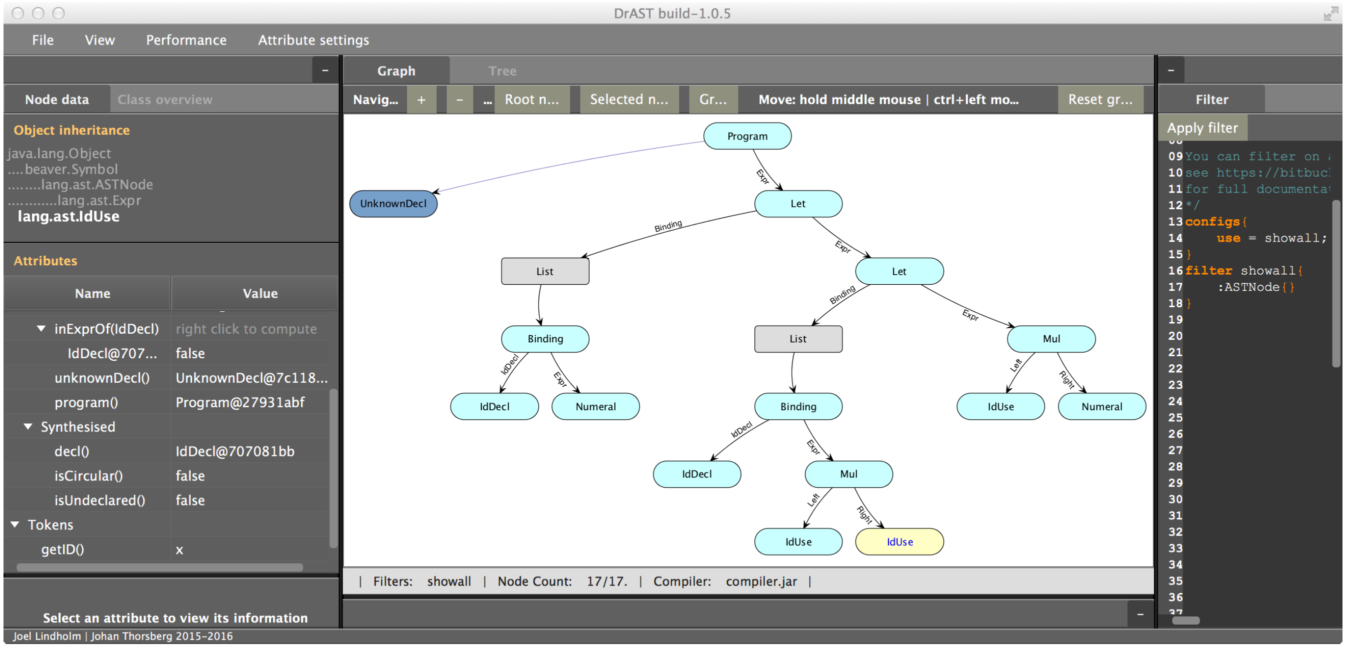

Figure 1. Overview of the DrAST tool

DrAST solves these challenges by making use of a domain-specific language for defining filters. The full attributed tree can be explored interactively to find out what nodes and attributes there are, and the filter can be interactively applied and modified. The filter can use attributes in the implemented language to support arbitrary filtering of nodes. We give an introductory example of the tool in Section 2, then discuss filtering trees in Section 3, and interactive exploration in Section 4.

DrAST is implemented in Java, using reflection and a model-view architecture. We have used it on several different compilers, including ExtendJ and JModelica.org. We discuss the implementation of DrAST and report on some measurements in Section 5. Finally, we discuss related work in Section 6, and then conclude in Section 7.

DrAST was implemented by Joel Lindholm and Johan Thorsberg, as a master's thesis project (Lindholm and Thorsberg 2016).

2. An introductory example

Consider the following program in a calculator expression language with `let`-bindings:

```
let x = 2.0 in
  let z = y * x in
    z * 4.0
  end
end
```

A compiler for this language has been implemented in JastAdd, and Figure 1 shows an overview of the DrAST tool run on this compiler for the above program. The abstract syntax tree is shown in the middle pane. The left pane shows attribute values for a selected node. The right pane shows a filter program that defines what parts of the tree to display. In this case the complete tree is displayed, by including all nodes of type `ASTNode` (a common supertype of all tree node classes).

3. Filtering trees and displaying attributes

Suppose that the user is interested only in viewing some of the nodes, for example, the declarations and uses of names, and let constructs (`IdDecl`, `IdUse`, and `Let` nodes). Furthermore, suppose the user would like to display the identifier tokens (modeled by a string attribute `getID`), and the bindings from uses to declarations (modeled by a reference attribute `decl`). This can be done by changing the filter to select only the node types of interest, and by showing desired attributes, as follows:[1]:

```
filter include{
  :Let{}
  :IdDecl{show{:getID;}}
  :IdUse{show{:getID;decl;}}
}
```

Figure 2 shows the resulting filtered attributed tree. Suppressed nodes are collapsed into *clusters* in such a way that the tree relations between the visible nodes is retained. Each cluster node is labelled with the number of nodes it represents. Attributes of primitive types, like `getID`, are shown directly in the nodes. Reference attributes, like `decl`, are shown as graph edges. The dark blue node of type

[1] Node types and attribute names are preceded by a colon to not conflict with the keywords in the filter language.

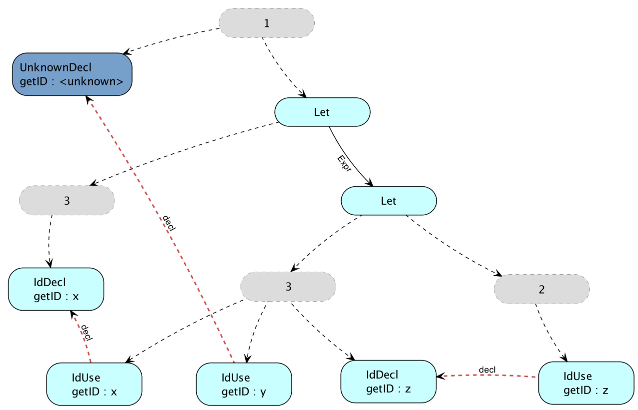

Figure 2. Filtered attributed tree. Each cluster node is labelled by the number of nodes it represents. Reference attributes are shown as graph edges.

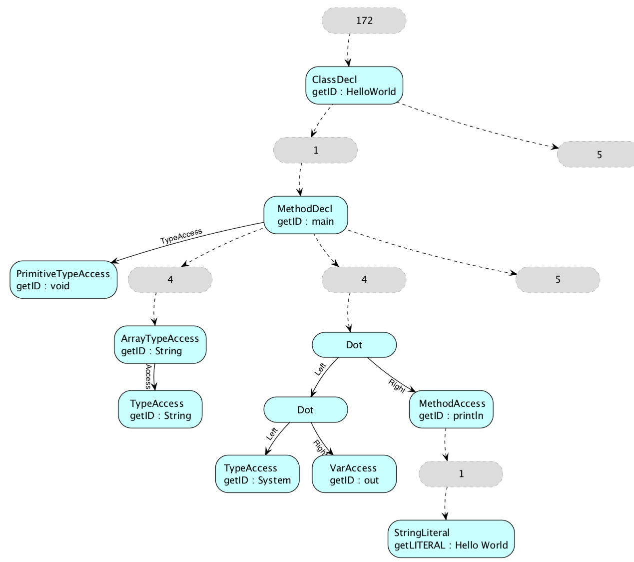

Figure 3. Filtered attributed tree for `HelloWorld.java` compiled by ExtendJ

`UnknownDecl` is a nonterminal attribute, i.e., an attribute that is itself a tree node.

Conditional filtering can be done using attributes from the AG. For example, if the user would like to view only the undeclared `IdUse` nodes, a `when` clause could be added:

```
:IdUse{when{:undeclared;} show{:getID;:decl;}}
```

where `undeclared` is a boolean attribute of `IdUse` that is defined by the AG.

4. Interactive exploration

The user can pan and zoom the tree, interactively collapse subtrees into clusters, and select nodes to interactively explore their attribute values. The values are shown in the left pane, and when clicking on a reference attribute, its graph edge is displayed in the tree. To see the effects of changing the displayed program, or updating the compiler, the user can recompile directly from DrAST.

Cached attributes

DrAST is attached to the syntax tree at a certain point in time, normally after compilation. The attribute evaluation in JastAdd is done on demand, and evaluated attributes are cached (memoized) for performance. The tree displayed by DrAST will thus have a number of cached attributes, whereas other attributes are not evaluated. It is normal in JastAdd compilers that there are many unevaluated attributes after compilation. This is because many attributes are needed only in particular situations, for example, accessed in conditional expression branches in the equations. In DrAST, the user can see which attributes are evaluated (cached) and which are not, and for those not cached, the user can interactively select and evaluate them, or select to evaluate all attributes.

Parameterized attributes

DrAST also supports JastAdd's *parameterized attributes* which are attributes that take on one or more arguments, and they can be thought of as virtual cached function calls (Hedin 2000). The cached calls are shown in the left pane, and additional calls can be made interactively. For example, the `IdUse` in our example compiler has a parameterized attribute `lookup(String)`, to look up declarations of a given name. For an `IdUse` node with the name `x`, there will be a memoized value for `lookup("x")` that was used in the evaluation to find the declaration of `x`. In DrAST, the user can also interactively do a call, e.g., `lookup("y")` to see what declaration a variable named `y` would have been bound to at that location in the tree.

Using DrAST on ExtendJ

The Java compiler ExtendJ uses nonterminal attributes heavily: to reify implicit concepts like the `Object` type, to represent imported libraries, and to handle compile-time expansion of parametric types. Thus, even a small program, like *hello world*, will have many nodes. Figure 3 shows the attributed tree for `HelloWorld.java`, where the filtering mechanism has been used to show only declarations and accesses.

5. Implementation

The major part of a JastAdd-based compiler is the attributed syntax tree, whose objects are instances of Java classes generated from the attribute grammar. To construct a complete compiler, the user writes a main program in Java that parses input files into an syntax tree, and then uses attributes to print error messages and output code (which in turn access other attributes to do name analysis, type analysis, etc.). All attributes are represented by methods, and evaluation is done on demand when attributes are accessed the first time. To give DrAST access to the attributed tree, the main program needs only define a static field `DrAST_root_node` of type `Object`, pointing to the root of the syntax tree. The user can then tell DrAST to run the compiler (which should be in the form of a jar file) on a specific input file, and DrAST will then access the resulting attributed tree and display it. The compiler and/or input program can be updated and run again, without having to restart DrAST.

DrAST is implemented using reflection. The Java classes generated by JastAdd have annotations that allow DrAST to understand the attributed tree structure, for example, which methods that correspond to attributes. The name DrAST is an acronym for Display Reflected Attributed Syntax Tree.

The internal structure of DrAST uses the Model-View pattern. This allows alternative views of the attributed tree to be implemented. For example, there is a view that can produce a `png` file from the filtered attributed tree.

We have run DrAST not only on small compilers suitable for teaching, but also on production compilers like the Java compiler ExtendJ[2] and the Modelica compiler JModelica.org[3]. It can handle large programs and large attribute grammars without any performance problems. For example, for a Java program of around 2000 lines of code, the total number of syntax nodes was around 170,000 (including all libraries needed for compiling the program), and most nodes have between 30-60 attributes each. Building the initial reflected model took only around 1.2 seconds, and rendering a filtered view, or recomputing it after interactions or changes to the filter, takes only a fraction of a second, and is not noticeable by the user.

6. Related work

Other attribute-grammar tools for debugging attributed syntax trees include Aki (Sasaki and Sassa 2003) and LISA (Henriques et al. 2005). *Aki* is a visual debugger for attribute grammars implemented in Smalltalk. It can visualize the abstract syntax tree, but has no specific mechanisms for filtering and interactive exploration of the attribution. The focus is instead on algorithmic and slicing-based debugging. These debugging mechanisms follow the attribute dependencies and query the user about correct attribute values to help pinpoint the source of a bug in the specification. Similar mechanisms would be interesting to include in future versions of DrAST.

LISA is a compiler generator system based on attribute grammars. It includes support for animating the evaluation of attributes, and following the execution of the evaluation by single-stepping and setting breakpoints. LISA also includes a system *Alma* for generating animations from attributed syntax trees, but which is primarily aimed at visualizing a running program in the target language, rather than to allow the user to interactively explore the attributed tree.

There are other tools that support visualization of syntax trees, but that do not support attribute grammars. Examples include ANTLRWorks (Bovet and Parr 2008) and VAST (Almeida-Martínez et al. 2008). *ANTLRWorks* can display parse trees, but the focus is on supporting debugging of the parser and showing connections between the parse tree, the parsed input, and the parsing grammar. No filtering or interactive exploration of attributes is supported. *VAST* is a tool that supports the visualization of abstract syntax trees independently from the parser technology used. The idea is to instrument the parser to generate visualization data in XML-form, and then read this data into the VAST tool which visualizes the tree. To deal with huge trees, VAST provides both a global view, for overview, and a zoomed view for details. In contrast to DrAST, no filtering or support for attribute grammars is provided.

Another kind of related work is that of visualizing a memory graph (a heap snapshot), in the form of a graph of objects (Zimmermann and Zeller 2002), and examples of tools include Heapviz (Kelley et al. 2013) and Fox (Potanin et al. 2004). *Heapviz* can summarize nodes based on their types, so that many objects of the same type are represented by a single node. *Fox* supports a two-step filtering, where the first criterion is based on type or depth in the graph, and the second is based on user-provided queries that can fetch data from the snapshot. The attributed tree in DrAST is similar to a heap snapshot, but visualized in a particular way, taking advantage of the fact that there is a spanning tree (the abstract syntax tree), and that certain methods correspond to attributes. However, DrAST works on a live heap, allowing attributes to be interactively evaluated, whereas heap snapshot tools typically work on a dumped file.

7. Conclusions

We have presented DrAST, an interactive tool for exploring attributed syntax trees, and that scales to large programs through the use of powerful filtering techniques. We see three main uses of the tool. One use is to support teaching, by allowing students to explore the results of attribute computations on example programs. A second kind of use is to support learning an existing attribute grammar, by exploring the syntax tree structure and what attributes there are in different kinds of nodes. Third, to support attribute grammar debugging, by allowing attribute values to be inspected to discover errors in the equations. Future work includes improving the debugging support by taking attribute dependencies into account. For example, it would be very useful to be able to filter out all nodes and attributes that a particular attribute depends on, and view the involved equations. JastAdd has experimental support for incremental evaluation, and we plan on using this information to implement such support (Söderberg and Hedin 2012).

[2] `http://extendj.org`

[3] `http://jmodelica.org`

References

J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749, Nov. 2010.

F. J. Almeida-Martínez, J. Urquiza-Fuentes, and J. Á. Velázquez-Iturbide. VAST: Visualization of abstract syntax trees within language processors courses. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 209–210. ACM, 2008.

J. Bovet and T. Parr. ANTLRWorks: an ANTLR grammar development environment. *Software: Practice and Experience*, 38(12): 1305–1332, 2008.

T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Sci. Comput. Progr.*, 69:14–26, 2007a.

T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *OOPSLA 2007*, pages 1–18. ACM, 2007b.

G. Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

P. R. Henriques, M. V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu. Automatic generation of language-based tools using the LISA system. *IEE Proceedings-Software*, 152(2):54–69, 2005.

S. Kelley, E. Aftandilian, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. *Information Visualization*, 12(2): 163–177, 2013.

D. E. Knuth. Semantics of Context-free Languages. *Math. Sys. Theory*, 2(2):127–145, 1968.

J. Lindholm and J. Thorsberg. DrAST-An attribute debugger for JastAdd. M.Sc. thesis, LU-CS-EX 2016-10, Lund University, Sweden, 2016.

A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 251–259. IEEE, 2004.

A. Sasaki and M. Sassa. Generalized systematic debugging for attribute grammars. *arXiv preprint cs/0310026*, 2003.

A. M. Sloane, L. C. L. Kats, and E. Visser. A pure object-oriented embedding of attribute grammars. *Electr. Notes Theor. Comput. Sci.*, 253(7):205–219, 2010.

E. Söderberg and G. Hedin. Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report 98, Lund University, April 2012. LU-CS-TR:2012-249, ISSN 1404-1200.

H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *PLDI*, pages 131–145, 1989.

E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39–54, 2010.

T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Software Visualization*, pages 191–204. Springer, 2002.