Declarative Program Analysis and Interpretation for AttoL

Anton Göransson
D14, Lund University, Sweden
dat14ago@student.lu.se

Abstract

I have implemented an interpreter for a small object-oriented language called AttoL. I then experimented with different kinds of program analysis. This was all done using JastAdd, which makes it possible to write declarative code. The idea is that the interpreter will be used for experimenting with declarative program analysis with attribute grammar since it has not been done that much before. I then evaluated my implementation by comparing program execution speed and source lines of code to the original AttoL implementation. The results were positive and I definitely think that JastAdd is a viable option for implementing program analysis. Because of JastAdd's modularity is easy to add new functionality without having to change old one, creating a maintainable and decoupled system.

1 Introduction

In this paper I will describe how I implemented an interpreter and program analysis for AttoL using JastAdd. JastAdd is a java-based system used for constructing compilers and it supports reference attributed grammars (RAG) [4]. RAG is an extension to attribute grammar that allows attributes to be references in the attribute grammar tree [3]. JastAdd has been used before to implement object-oriented languages, an example is the implementation of Oberon-0[2]. It is also a small language with many features, similarly to AttoL.

The goal of the paper is to analyze how program analysis can be written with a declarative approach using attribute grammar (AG) and decide how suitable JastAdd is for doing this. I first created the groundwork (the interpreter) to allow experimenting with program analysis written declaratively using AG. It has not been experimented with much and if I can create groundwork that is stable enough there might be new insights with further experimentation.

I then implemented some different analyses and because of JastAdd's modularity it was simple to add them to the existing program that I had written. The definite assignment analysis was implemented with ca 70 source lines of code and I did not have to modify any existing code. Most of the time was spent on the interpreter to make sure it was stable.

Course paper, EDAN70, Lund University, Sweden
January 23, 2019.

2 Background

In this section I will introduce and desribe the different tools and systems that I used.

2.1 JastAdd

JastAdd is an object-oriented compiler generator system that supports both imperative and declarative approaches. Because of its object oriented focus it is very modular and it is simple to add a new feature in it's own file without modifying the existing system. The pros of supporting declarative attributes are that the order of computations does not need to be defined.

JastAdd supports on-demand evaluation. This is what allows the aspects to be so decoupled. In JastAdd the abstract syntax is modeled by a class hierarchy. For each class Java code is generated and these classes are called AST classes as they model a node in the abstract syntax tree. Some of the key features of JastAdd are described below.

2.1.1 Aspects

In an aspect file you can declare intertype declarations for AST classes. Intertype declarations appear in the aspect file but they belong to an AST class. JastAdd reads the aspect files and interweaves the intertype declarations to the correct AST classes. The intertype declarations can be regular Java methods, Java Fields and attribute grammars constructs such as equations. Since JastAdd caches the return value of equations they are typically used when the result will not change and Java methods are used when the return values can change during run time.

2.1.2 Reference Attribute Grammar (RAG)

JastAdd supports RAG which allows writing of declarative code for computations. Each computation is defined by attributes and equations. Each equation can either be inherited for sending information down the tree or synthesized for sending information up the tree[4]. Contrary to plain AG where all attributes are values RAG allows attributes to references to other nodes. It does not matter where they are in the tree. The reference nodes can be dereferenced to obtain the value of the node referred to. This allows for direct links between nodes[3].

```
coll Set<ErrorMessage> Program.errors()
    [new TreeSet<ErrorMessage>()]
    with add root Program;

IdUse contributes error
    ("symbol '" + getID() + "' is not declared")
    when decl().isUnknown()
    to Program.errors() for program();
```

Figure 1. First a collection attribute is defined on program, declaring a treeset containing errormessages. Then a contribution is declared, contributing to the collection when decl.isUnknown() equals true.

2.1.3 Circular Definitions

JastAdd supports writing circular definitions declaratively and it is not necessary to code when the iteration should take place. Attributes handles this. It allows circular attributes to be freely combined with other modules. The circular attributes are calculated using fixed-point iteration. When defining a circular attribute you explicitly define it as circular and give it a starting value.

2.1.4 Collections

It is possible to define a collection attribute on a ASTNode. Other ASTNodes can then contribute to this collection through contribute statements. JastAdd performs a survey of the AST searching for contributions to the collection. An example used in my program is shown in figure 1.

2.1.5 Nonterminal Attribute

JastAdd allows defining AST nodes by equations. These are called nonterminal attributes (NTA). They are similar to both nodes and attributes since it can have attributes and it is defined by equations. NTA's refer to a new sub tree corresponding to predefined types and methods. I have for example followed the null-object pattern when handling unknown function declarations. The null object pattern is used for handling declarations that might be null or undefined. By encapsulating it in an object, in our case a NTA we avoid direct comparison with null and allows for easier error handling[5].

2.2 AttoL

AttoL is a small object-oriented language[1]. An AttoL program consists of a sequence of statements. AttoL supports a lot of common program features. For example, if and while statements, arrays, variable, class, and function declarations and assignments and dynamic typing.

When a new class is instantiated all the statements inside the body block is ran. A small program, taken from AttoVM webpage, covering many of the language's features such as classes, initialisers, literal arrays, strings, dynamic dispatch, and loops and it's expected output is shown in figure 2.

```
class C(int x) {
    int s = x;
    int size() {
        return s;
    }
}

obj counters = [[], C(1), "foo"];
int i = 0;
while (i < counters.size()) {
    print(counters[i].size());
    i := i + 1;
}
```

Figure 2. Small AttoL example program, showing support for many of the language's features

3 Implementation

In this section I will describe the different parts of my implementation. Starting with the interpreter and continuing with the different analyses.

3.1 Scanner And Parser

To be able to create an interpreter of AttoL I also had to implement a scanner and a parser. The scanner was generated using JFlex[2]. JFlex is a scanner generator for Java, written in Java.

Generating the scanner was made with Beaver[3]. Beaver is a LALR(1) parser generator that takes a context free grammar and converts it into a Java class that implements a parser for that grammar.

Both of these were used in the compiler course and they work nicely together with JastAdd. Therefore I chose to use them for this project.

3.2 Interpreter

The abstract grammar is very similar to that of AttoL's, described in the overview[4]. The grammar is modeled by a hierarchy. I have general abstract classes such as statement and expression which most of the other classes are subclasses of.

The interpreter is in its own JastAdd aspect and uses an activation record to keep track of values. The activation record also has its own aspect with regular Java methods to get and insert a value. It is basically a hash map where each entry consists of a string (the variable's ID) as key and a Value class object that I implemented as the key's value. This

[1]AttoVM Web page, http://sepl.cs.uni-frankfurt.de/teaching/attovm.en.html

[2]JFLex Web page, http://www.jflex.de/

[3]Beaver Web page http://beaver.sourceforge.net/

[4]AttoL Overview http://sepl.cs.uni-frankfurt.de/teaching/overview.pdf/

Value object can contain a string, an int or another activation record used for new instances of a class. When a class is instantiated all its statements are run, the values of these statements are then saved in the value objects activation record. This is similar to a function call where a temporary activation record is created for the duration of it's run time.

Almost all intertype declarations in the interpreter aspect are basic Java methods and fields since their return values change during run time. I have defined a Java method called `eval` which program, all statements and all expressions implements to correctly evaluate an AttoL program.

To be able to get the declaration of a variable that is used I have implemented a function `decl`. The function calls an inherited attribute `lookup(String id)` which all relevant statements define. These are:

- ClassDecl - The variable might be the class name or a formal.
- FuncDecl - The variable might be the function name or a formal.
- VarDecl - The variable might be the variable that VarDecl declares.
- Block and Program - Should only look for declarations before the variable is used.

The name analysis logic is in its own aspect, utilizing the modular approach of JastAdd. In this aspect most of the intertype declarations are equations since the values won't change during run time. RAG also shows it's strengths here, since I don't have to define any order of the `lookup` calls, JastAdd handles this.

3.2.1 Definite assignment analysis

The implementation of definite assignment (DA) analysis is defined by two inherited attributes on the `IdUse` class. These attributes are: `isDAafter(IdUse id)` and `isDAbefore(IdUse id)`. The idea is that an `IdUse` has to be DA before it is used and it is if it is DA after any previous statements in the program. I then collect all uses of an id that were not DA before in a collection attribute. Similarly to the example shown in figure 1.

3.2.2 Circular Inheritance analysis

The circular inheritance analysis is implemented using a circular attribute. This collection is a hash set containing the ids of all superclasses of a class. This means that if there exists three classes: A, B, and C where A inherits from B and B inherits from C. The collection of class A would contain B and C. If the definition is circular the class itself will also be contained in it's own superclass collection and an error is reported. Here the start value is an empty hash set.

	JastAdd	C
Program 1	0.111504, 0.137556	0.220235, 0.243146
Program 2	0.414614, 0.481326	0.196562, 0.205127

Table 1. Confidence intervals after running test program 1 and 2 a 100 times with the different implementations

4 Evaluation

4.1 Proof of concept examples

A working example constructed by my supervisor Alfred Åkesson is shown in figure 3. It showcases a linked list implemented using a class. Null checks are also used making use of the null-object pattern implemented with a NTA.

Another example showcasing the inheritance and dynamic dispatch is shown in figure 4.

4.2 Performance

I compared the implementation of AttoL in C with my implementation in Java, using JastAdd on the following two programs.

I ran both the program shown in figure 5 and the program shown in figure 6, 100 times on each implementation and got the confidence intervals in seconds seen in table 1. The JastAdd results are after warmup.

It is quite surprising that the C implementation does the 1 million loops faster than a simple class instantiation but I don't now exactly how it's implemented there. My implementation handles the class instantiation quite fast and that was expected. It does take longer to execute program 2 but I think that is more about java than my implementation since there are no heavy calculations.

4.3 Definite Assignment analysis

I chose to evaluate my implementation of definite assignment analysis by looking at the source lines of code (sloc) for my implementation and the C implementation. My implementation has 68 sloc and the C implementation has 168 sloc. The C implementation does also use a generic data-flow analysis framework which has 338 sloc. By just looking at these numbers we can see that my implementation has a lot less sloc. This might of course not mean that my solutions is better or more understandable but I still think that it shows some of the pros of declarative programming and RAG.

4.4 Circular Inheritance analysis

I also evaluated the circular inheritance using sloc. It was implemented with 13 sloc, which is very few. I do not have proper error messages though. Therefore it is currently only reporting the line where the circular inheritance is defined. It does not say anything about which classes are involved. This would probably add some more sloc.

```
class ListElement(obj data) {
    obj el = data;
    obj next = NULL;

    obj add(obj d) {
        obj toAdd = ListElement(d);
        obj a = toAdd;
        if (next == NULL) {
            next := toAdd;
            return d;
        }
        obj last = next;
        int i = 0;
        while(last.next != NULL) {
            i := i + 1;
            last := last.next;
        }
        last.next := toAdd;
        return d;
    }

    obj get(int index) {
        if (index == 0)
            return el;
        int i = 1;
        obj cur = next;
        while (i < index) {
           i := 1 + i;
           cur := cur.next;
        }
        return cur.el;
    }
}


obj lista = ListElement(2);

lista.add(6);
lista.add(8);
lista.add(10);
lista.add(12);

print(lista.el);
print(lista.get(0));// Print 2
print(lista.get(1));// Print 6
print(lista.get(2));// Print 8
print(lista.get(3));// Print 8
print(lista.get(4));// Print 8
```

Figure 3. Example program, showing a working example program constructed by my supervisor

```
class Rectangle(int w, int h) extends Shape {
    super(w, h);
}

class Square(int side) extends Shape {
    super(side, side);
}

class Triangle(int b, int h) extends Shape {
    super(b, h);
    int area() {
        return x * y  / 2;
    }
}

class Shape(int width, int height) {
    int y = width;
    int x = height;
    int area() {
        return width * height;
    }
}
obj r = Rectangle(5, 10);
obj s = Square(10);
obj t = Triangle(5, 10);
print(r.area()); // 50
print(s.area()); // 100
print(t.area()); // 25
```

Figure 4. Example program, showcasing inheritance and dynamic dispatch.

```
class Record(obj x, obj y) {
      obj x = x;
      int y = y;
      int both = x * y;
}

obj r = Record(2, 3);
print(r.both);
print(r.x);
print(r.y);
```

Figure 5. Test program 1, showing a very simple program instantiating a class

```
int x = 1;
while (x < 1000000) {
    x := x + 1;
}
print(x);
```

Figure 6. Test program 2, showing a very simple program looping 1000000 times

5 Related work

JastAdd has been used previously when implementing an interpreter for an already existing language[2]. Oberon has some similarities to as it is also a small object-oriented language. They had a very similar structure on their program, with an abstract statement and expression class in their abstract grammar. This is almost unavoidable if you want to have a healthy structure and are implementing a object-oriented language since java handles hierarchy so well. The lookup definition is also very similar to my solution.

In ”Declarative Intraprocedural Flow Analysis of Java Source Code” an implementation of definite assignment analysis using JastAdd is described[6]. I have taken some inspiration and have tried to implement a similar solution.

Some examples of powerful applications of JastAdd are shown in "The JastAdd Extensible Java Compiler" by Torbjörn Ekman and Görel Hedin [1]. They showcase the implementation of name analysis, type analysis and definite assignment using declarative methods.

6 Conclusion

I have implemented an interpreter for a small programming language called AttoL using JastAdd. I then extended the language with inheritance and implemented two different program analyses, also using JastAdd.

Most of the time were spent on creating a solid base to experiment with. But when it was done it was quite simple to add new functionality without modifying old code. The results were positive with less sloc than the original C implementation of AttoL.

7 Future Work

As the intention was to create the groundwork to allow further experimentation with declarative program analysis using AG, my implementation can hopefully be used exactly for that since from my results I definitely think JastAdd is viable.

Acknowledgments

I would like to thank my supervisor Alfred Åkesson for helping me out, solving bugs that I would probably not otherwise have solved.

References

[1] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. *ACM Sigplan Notices* 42, 10 (2007), 1–18.

[2] Niklas Fors and Görel Hedin. 2015. A JastAdd implementation of Oberon-0. *Science of Computer Programming* 114 (2015), 74 – 84. https://doi.org/10.1016/j.scico.2015.02.002 LDTA (Language Descriptions, Tools, and Applications) Tool Challenge.

[3] Görel Hedin. 2000. Reference attributed grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.

[4] Görel Hedin and Eva Magnusson. 2003. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58.

[5] Kevlin Henney. 2002. Null object. In *Proceedings of the Seventh European Conference on Pattern Languages of Programming, EuroPLoP.*

[6] Emma Nilsson-Nyman, Görel Hedin, Eva Magnusson, and Torbjörn Ekman. 2009. Declarative intraprocedural flow analysis of Java source code. *Electronic Notes in Theoretical Computer Science* 238, 5 (2009), 155–171.