A SimpliC compiler in Scala and Kiama

Project in Computer Science – EDAN70
January 15, 2015

Johan Andersson
D10, Lund Institute of Technology
johan.andersson.734@student.lu.se

Abstract

JastAdd and Kiama are both tools that support working with attribute grammars in compiler construction. JastAdd is a tool that generates Java code, while Kiama is a Scala library.

In this project, a compiler was created for a simple C-like language, using Kiama and Scala. A compiler for this language has previously been implemented, but instead using JastAdd.

These two compilers were then evaluated in terms of code size and performance, to see how well they performed. The goal of this project was to compare Kiama with the more widespread tool JastAdd.

1. Introduction

The goal of this project was to implement a compiler for the SimpliC language, using Scala and the attribute grammar library Kiama [1], and compare it to another compiler for the same language. This other compiler has previously been implemented using the attribute grammar tool JastAdd [2]. The JastAdd compiler was created in the EDAN65 Compilers course at Lund University [3]. The comparison was done in terms of runtime performance and size of the respective code bases.

SimpliC is a simple C-like language, with support for functions, integer variables and constants, while-loops, simple conditionals, but not more advanced features. Booleans, logical 'or', and logical 'and' are for example not part of the language. A SimpliC example program for calculating factorials can be seen in Listing 1.

Listing 1. A SimpliC example program

```
int factorial(int x){
        if(x == 0){
                return 1;
        }else{
                return x * factorial(x - 1);
        }
}

int main(){
        int n = read();
        print(factorial(n));
}
```

A related paper worth mentioning is 'A pure embedding of attribute grammars'[4], which describes the attribute grammar library Kiama. The paper compares Kiama with JastAdd. They find that while Kiama has a briefer syntax, it is also the slower of the two. When evaluating the code specification size in this project, the results were similar; the Kiama compiler needed fewer lines of code. But when doing a performance evaluation, the results were not the same as in previous work. The Kiama compiler seemed to have a better time complexity than the JastAdd compiler when tested on a sample program..

2. Attribute Grammars

Attribute grammars decorates abstract syntax trees with attributes. [7] The values of the attributes are specified with equations. These equations are unordered, and automatically computed by an evaluation engine.

Originally there were two types of attributes: Synthesized attributes and inherited attributes. However, nowadays there are more attributes that are supported by both JastAdd and Kiama. Circular and parameterized attributes are two examples of this. Collection attributes is another type of attribute that is supported by JastAdd, but not by Kiama.

The synthesized attributes have their equations defined in the same node of the abstract syntax tree as the attribute is defined in, while inherited attributes have their equations defined in an ancestor.

An attribute can be circular in its definition, i.e. it depends on itself. The value of the attribute will then be computed using fixed-point iteration. Circular attributes must be explicitly declared circular in both Kiama and JastAdd.

A collection attribute is defined by a set of contributions instead of an equation. A parameterized attribute takes parameters.

After the value of an attribute is computed, it is cached, so that it does not have to be recalculated multiple times. The value of an attribute is always the same and does not change.

3. Scala and Kiama

Scala stands for 'Scalable Language' and is a language that combines object-oriented and functional programming. It is compiled to bytecode that is executed on the Java Virtual Machine (JVM). This means that Scala can use Java libraries, and it should be possible to use JastAdd together with Scala.

3.1 Scanning

A Scala library has been used that parses on a character level, which means that no explicit scanning of the input is required. This is often called "Scannerless parsing" [9]. The JastAdd compiler uses JFlex for scanning. [5]

3.2 Parsing

In this project, the Scala `PackratParser` class was used. It is a part of the Scala standard library. As this is standard Scala parsing, it will not be discussed in detail here, but an example comparison of the parsing used in the JastAdd and Kiama compilers are provided in Listing 2 and 3. The JastAdd compiler uses the parser generator Beaver [6] for parsing.

Listing 2. Simple parsing in Beaver syntax

```
if_stmt    = IF LPAREN expr.a RPAREN LBRACKET
                block.b RBRACKET else_clause.c
                {: return new IfStmt(a, b, c); :}
```

Listing 3. Simple parsing in Scala syntax

```
lazy val ifStmt : PackratParser[IfStmt]
= "if" ~> ("(" ~> expr <~ ")") ~ ("{" ~> block <~ "}")
        (elseClause?) ^^ { case a ~ b ~ c =>
                IfStmt(a, b, c) }
```

3.3 Abstract Syntax Trees and Attributes

In JastAdd, attributes are specified using a special language in .jrag-files, and then JastAdd generates Java code from these specifications. The classes representing the abstract syntax tree are also defined using a special language.

In Kiama however, normal scala classes represent the nodes of the abstract syntax tree, and the attributes are defined as normal Scala functions. This means that there is no new syntax added in Kiama. Snippets of abstract syntax tree definitions can be seen in Listing 4 and 5.

Listing 4. AST definition syntax using JastAdd

```
abstract Stmt;
abstract Expr;
AssignStmt : Stmt        ::=     IdUse Expr;
IdUse : Expr             ::=     <ID:String>;
```

Listing 5. AST definition syntax using Scala

```
abstract class Stmt extends LangNode
abstract class Expr extends LangNode
case class AssignStmt (idUse : IdUse, expr : Expr)
                        extends Stmt
case class IdUse (name : String) extends Expr
```

To define an attribute using Kiama, the `attr` function is used. Parameterized attributes and circular attributes are defined in a similar manner using the `paramattr` and `circular` functions. Example implementations of the attribute `isUnknownType` in JastAdd and Kiama can be seen in Listings 6 and 7. `isUnknownType` is an attribute that takes a `Type` (e.g an integer or boolean type) and returns true if the given type is unknown. The unknown type is used for undeclared variables, among other things.

Listing 6. Definition of isUnknownType using Kiama

```
lazy val isUnknownType : Type => Boolean =
        attr {
                case unknownType : UnknownType =>
                        true
                case _ => false
        }
```

Listing 7. Definition of isUnknownType using JastAdd

```
syn boolean Type.isUnknownType() = false;
eq UnknownType.isUnknownType() = true;
```

4. Implementation details

The two language tools were used to implement attributes for semantic analysis, reachability analysis, pretty printing and code generation for SimpliC. An overview of the implemented features of the compilers can be seen in Table 1.

The semantic analysis consisted of name analysis and type analysis. The name analysis in turn consisted of attributes for checking for undeclared variables, multiply declared variables, and functions with the same name as reserved functions among other things.

The type analysis had functionality for checking if a boolean value is assigned to an integer variable, a function parameter is called with the wrong type, there is a non-boolean value in a boolean expression, the return type is correct, and if a variable is incorrectly called as a function.

The reachability analysis consists of checking what functions are reachable from a given function. This entails following the function calls of the function. Because functions can be recursive, this definition is circular, and the corresponding attribute in the compiler is also circular.

This is implemented as the `reachable` attribute, which depends on another attribute, `functionCalls`, that returns all the function calls of the function declaration. In the JastAdd Compiler, `functionCalls` could be implemented using collection attributes. However, as this is not supported in Kiama, the attribute had to be implemented in another way in the corresponding compiler.

Table 1. Implemented compiler features

	JastAdd compiler	**Kiama compiler**
Explicit scanning	●	
Parsing	●	●
Semantic analysis	●	●
Reachability analysis	●	●
Pretty printing	●	●
Visitor interface	●	
Interpreter	●	
Code generation	●	●

Machine code generation was implemented in both compilers, but the target language were not the same. The reason for this was mainly to make as much use of Kiama as possible. Kiama has a class called `AbstractMachine` that can be used to create various machine emulators.

Several of the example projects on the Kiama web page involved code generation, and they all had the same target language. It is a certain kind of RISC, code, defined in the book 'Compiler Construction' by Niklaus Wirth [10]. To run this code, you also need a machine. This was also provided in the Kiama examples, namely a RISC code machine emulator. This was used in the Kiama compiler.

In the JastAdd project, x86 assembly was the target code. It does not matter too much that these two types of code are different from each other, because code generation is not the main focus of this project. Code optimization was outside the scope of this project.

Because a Scala parsing library that has scannerless parsing was used, no explicit parsing had to be implemented for the Kiama compiler.

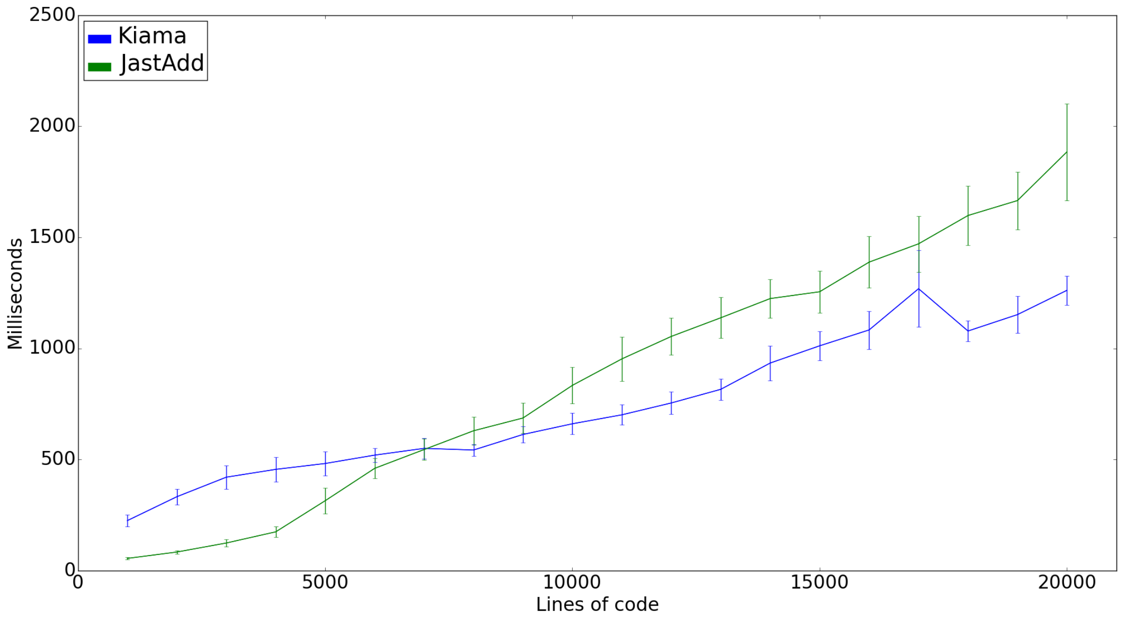
5. Evaluation

The compilers were evaluated by comparing their runtime performance and specification size.

5.1 Performance

To evaluate the runtime performance of the two attribute grammar systems, the `errors`-attribute was measured. The purpose of the

Figure 1. runtime performance comparison



`errors` attribute is to find and collect any errors in the scanned and parsed program. Errors include but are not limited to: undeclared variables, multiply declared variables, declarations of reserved functions (print, read), assignment of a boolean value to an integer variable and function parameter mismatches. Exactly the same analysis and error checking are implemented in the two compilers to ensure a fair comparison.

For each compiler, a number of programs were ran, and the time it took to evaluate the attribute was invoked. 20 programs of different length were evaluated. They were similar in nature, and they all consisted of a single code snippet that was repeated to create a program of the preferred length. The size of the programs ranged from 1000 to 20000 lines of code in increments of 1000 lines of code. The original code snippet was 100 lines long and consisted of a number of functions that were called by a main function. Several errors were present in these functions.

Each data point, i.e. each program, was measured 30 times for each compiler. Only the `errors`-attribute was measured, no scanning, parsing or anything like that was measured. The results can be seen in Figure 1.

Regarding the performance evaluation results; if we look at Figure 1, we see that JastAdd is faster up to an input of 7000 lines of code, and then Kiama is the faster one for input with 8000 lines of code or more.

This is unexpected. Previous work suggests that JastAdd should be significantly faster than Kiama [4]. Why Kiama is faster in this comparison is not entirely clear. It probably has something to do with the implementation of the JastAdd compiler. Either that, or it is caused by the test program used for the evaluation. The test program could be some sort of special case that is better suited for Kiama than JastAdd, but that is not very likely.

5.2 Code size

The two compilers were also evaluated with respect to their number of lines of code. This evaluation was done with the tool 'cloc' [8]. Cloc counts the number of lines of programs, excluding comments and empty lines. It supports many languages out of the box, but not .jrag-files by default. So the JastAdd-related files had their lines of code counted as Java files.

The results are listed in Figure 2. The goal of this project was to compare attribute grammar tools, so it is interesting to look at the total number of lines of code without any scanning or parsing included since they are not a part of attribute grammars.

If we look at the comparison of the code sizes, we can see that the Kiama compiler has less lines of code in total. But reachability analysis and the abstract grammar specification has more lines of code in the Kiama implementation. The abstract grammar contains more boilerplate code here, so naturally it will have a larger source code than in the JastAdd compiler, which has a very brief corresponding implementation.

As for the reachability analysis; the results here probably depend on the fact that the JastAdd implementation uses collection attributes. As this is not supported in Kiama, this analysis had to be implemented in another way.

However, Kiama has much more terse pretty printing than JastAdd in this project. A contributing factor to this might be that the Kiama compiler have built-in functions represented in another way. The functions are created in an attribute, `predefinedFunctions`, while in the JastAdd compiler they are defined as separate classes and thus require slightly more code. The main factor here is probably the Scala syntax though.

These results show some of the positive aspects of working with Scala and Kiama, namely the syntax. Scala has less boilerplate code than Java, and Kiama does not introduce much new syntax, so it is very easy to work with. However, the way that the abstract grammar is specified using JastAdd uses even less code than Scala uses, which makes it easier to get a good overview over the abstract grammar.

Table 2. Source code size comparison

	JastAdd	Kiama
Scanning	86	-
Parsing	52	-
canning + Parsing	-	75
Abstract grammar	37	54
Semantic analysis	251	216
Reachability analysis	19	54
Pretty printing	143	46
Tree dumping	24	31
Code generation	282	295
Total (parsing/scanning included)	894	771
Total (parsing/scanning excluded)	756	696

6. Conclusion

In this project, a compiler for a simple C-like language was implemented using the Scala attribute grammar library Kiama. It was then compared to another compiler for the same language that had been implemented using JastAdd.

When compared to each other, it was found that the Kiama compiler had the smallest specification size, and had a better runtime performance after a certain input size. The performance evaluation results contradict the results of previous work, and the cause for this is unclear. Maybe a future project could find the reason why. Another suggestion for future work is to use more test programs to compare the two compilers in this project, and to compare them in more detail.

References

[1] AM Sloane, Lightweight Language Processing in Kiama, Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III, 2009

[2] T Ekman, G Hedin, The JastAdd system modular extensible compiler construction, Science of Computer Programming 69(1-3), 2007

[3] EDAN65 Compilers course, http://cs.lth.se/edan65

[4] AM Sloane, LCL Kats, E Visser: A pure embedding of attribute grammars. Science of Computer Programming 78 (10), 2011 http://wiki.kiama.googlecode.com/hg-history/v1.2.0/papers/SCP11.pdf

[5] G Klein, S Rowe, R Dcamps, Jflex-the fast scanner generator for java, online source, 2001, http://jflex.de/

[6] Beaver a LALR Parser Generator, http://beaver.sourceforge.net

[7] G Hedin, An Introductory Tutorial on JastAdd Attribute Grammars, Springer Berlin Heidelberg, 2011

[8] A Danial, CLOC—Count lines of code, 2009, http://cloc.sourceforge.net/

[9] E Visser, Scannerless generalized-LR parsing, Technical Report P9707, in: Programming Research Group, University of Amsterdam, 1997

[10] N Wirth, Compiler Construction. Addison-Wesley, 1996. http://www.cs.inf.ethz.ch/ wirth/books/CompilerConstruction/