Compiler in Scala and Kiama in comparison with JastAdd

Project in Computer Science - EDAN70
January 20, 2015

Andrée Ekroth
D11, Lund University, Sweden
dat11aek@student.lu.se

Felix Mulder
D11, Lund University, Sweden
dat11fmu@student.lu.se

Abstract

This paper examines the main differences in usability and performance between Kiama, a Scala library for language processing, and the combination of tools used in course Compilers EDAN65 offered at Lund Institute of Technology. Specifically the focus of the paper is how well Kiama can perform with the aid of Scala compared to JastAdd which generates Java code. The Scala language is considered a more advanced, and by some an experimental language when compared to Java. As such there will also be a focus on how feasible the usage of Kiama is versus using JastAdd or alternatively a compiler implemented by hand for production.

1. Introduction

This project implements a lab language called SimpliC. The language is used in the course Compilers EDAN65. It is a small subset of C using C89 style declaration of variables and supports simpler constructs from C. The purpose of this re-implementation and project is to compare the JastAdd tool to other compiler generators, namely Kiama. Kiama is a Scala library for language processing. According to its wiki it enables convenient analysis and transformation of structured data. Like JastAdd, it provides support for common language processing paradigms such as: attribute grammars, tree rewriting, abstract state machines and pretty printing[1]. Which it does without the need for external tools. This paper will show that Kiama offers well thought out, yet sometimes limited, features.

As aforementioned, this project will implement SimpliC - the lab language from EDAN65. In said course the students implement a basic compiler with common compiler features including: scanning, Abstract Syntax Tree building, semantic analysis, interpretation, static analysis and code generation (x86_64 assembly).

In EDAN65 the students implemented this language using the following tools:

1. **JastAdd**: a meta-compilation system that supports Reference Attribute Grammars[2].
2. **Beaver**: an LALR parser generator[5].
3. **JFlex**: a flex-like lexer generator for Java with emphasis on speed and full Unicode support[6].

Kiama doesn't need to handle 2 and 3 since Scala handles both parsing and tokenizing.

2. Implementation of SimpliC

SimpliC is best described as a small subset of C89. SimpliC only supports integer variables and these need to be declared first in a block. While this is a limitation, like in C89 you can declare shadowed variables in a nested block. Aside from arithmetic expressions the language supports boolean expressions in the usual control structures like `if`- and `while`-statements. SimpliC does not support `for`-loops. Listing 1 attempts to showcase most of the constructs supported in SimpliC. The example lacks variable declaration, and nested variable declaration.

Listing 1. A showcase of SimpliC language constructs

```
int gcd1(int a, int b) {
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    return a;
}

int gcd2(int a, int b) {
    if (b == 0) {
        return a;
    }

    return gcd2(b, a % b);
}
```

2.1 Parser

Kiama uses Scala's parser combinators and specifically a packrat parser implemented in said library. As the documentation provides an excellent explanation of what packrat parsers are, we quote:

Packrat Parsing is a technique for implementing backtracking, recursive-descent parsers, with the advantage that it guarantees unlimited look-ahead and a linear parse time. Using this technique, left recursive grammars can also be accepted. [7]

Packrat parsing uses memoization and lazy evaluation to keep linear time at the expense of greater memory usage[Parsing Expression Grammars: A Recognition-Based Syntactic Foundation]. Kiama uses Scala's parser combinators to perform this, as shown in listing 2. The use of the keyword `lazy` when declaring parsers is required in order to avoid problems with the order of declaration.

Listing 2. Parsing an if statement to a "stmt"

```
1  lazy val ifStmt: PackratParser[Stmt] =
2    Tokens.If ~> (Tokens.LPar ~> cmp <~ Tokens.RPar) ~
      block (Tokens.Else ~> block?) ^^ If
3
4  lazy val stmt = ifStmt | whileStmt | assign
```

Compared to when using the JastAdd suite, an external tool, like Beaver, is not needed. Scala provides the necessary functionality without needing an intermediate abstraction layer. Scala's parser combinators library performs a lot of heavy lifting in Kiama, in fact all of Kiama's parsing can be done via Scala parser combinators. It should be noted that the parsing can also be handled by an alternate library *Rats!* which is part of *xtc* the *eXTensible Compiler* project[4]. An evaluation of *Rats!* is beyond the scope of this paper.

Parsing maps the input to case classes which can then be used to build the Abstract Syntax Tree (AST). Using case classes in Scala is a way to model algebraic data types, as well as generating `equals` and `hashCode` automatically.

2.2 Abstract Syntax Tree Building

The AST is defined by using Scala `case classes` and by extending a Kiama trait `TreeNode`. This trait in turn inherits from the `Attributable` trait which adds support for attributes and some useful methods. Listing 3 contains a small example of an AST. In order to be able to use all the methods added by Kiama it is required to explicitly initialize the tree using an `initTree` function on the AST. This initialization is required each time a new tree is created as the function will add the attributes to the tree. If an attempt to use the attributes is made without first initializing the tree, a run-time error will be thrown when attempting to use any defined attribute[8].

Listing 3. AST node definitions

```
1 sealed abstract class ASTNode extends TreeNode
2
3 case class Program(decls: Seq[FuncDecl]) extends ASTNode
4 case class FuncDecl(name: Id,
5                     param: Seq[Param],
6                     block: Block) extends ASTNode
7
8 abstract class Stmt extends ASTNode
9 case class Assign(lhs: IdUse, rhs: Exp) extends Stmt
```

2.3 Semantic Analysis

When the AST has been built, the same mechanics as in JastAdd can be used. Meaning reference attribute grammars may be used. Listing 4 shows how to define an attribute for symbol usage.

Listing 4. Adding an attribute to a symbol usage

```
1 val decl: CA[ASTNode, IdDecl] = attr {
2   case n @ IdUse(name) => n->lookup(name)
3   case n @ FuncCall(name, _) => n->lookup(name)
4 }
```

This gives us a cached attribute value `decl` for the `ASTNode` to which this value is applied using the arrow method. Which would have the JastAdd syntax[10] illustrated in listing 5.

Listing 5. The equivalent of listing 4 in JastAdd

```
1 syn IdDecl ASTNode.decl();
2 eq IdUse.decl() = lookup(name);
3 eq FuncCall.decl() = lookup(name);
```

In listing 4, the arrow is a method for passing `n` as the first parameter to the lookup attribute calculation. Kiama uses functions instead of class-bound methods in order to add support for attributes. This can result in odd compilation errors if another function or variable with the same name is in scope.

When using `attr` the match is applied to the supplied ASTNode. When using `childAttr` the outer match is applied to the node, giving the user a context to which it can then match the parent node in the inner match. This is how Kiama handles inherited attributes, depending on the parent node, the child node is given an attribute. Listing 6 illustrate this point.

Listing 6. Using a parent node as context for a child attribute

```
1 val expressionType: CA[ASTNode, Type] = childAttr {
2   case e: Exp => {
3     case _: BinExpr      => IntType
4     case If(`e`, _, _)   => BoolType
5     case _               => UnknownType
6   }
7   case _ =>
8     case _ => UnknownType
9 }
```

If the match is not exhaustive, the compiler will warn the user. If a match fails during run-time, a run-time error will be thrown. In the course EDAN65, the static interpretation of SimpliC was implemented such that if the evaluation reached an abstract type, a run-time error would be thrown - i.e. the same as a failed match.

In listing 6 `expressionType` has the type `CachedAttribute[ASTNode, Type]`. This means that the attribute is applied to `ASTNode` and maps to `Type`.

In some cases the need to pass a parameter to an attribute is needed. Kiama supplies a `paramAttr` attribute. In this case the value would evaluate to a `CachedParamAttribute[T,U,V]` where `T` is the type of the parameter[8].

Broadcasting is supported in Kiama much like in JastAdd and can be added using an attribute, see the first attribute in listing 7. This feature is also achievable in Kiama using its decorators. Decorators are simply functions for common patterns of attribution, see listing 7, the second attribute. The `down` function will pattern match on the current node, if it can not match the node it will propagate the evaluation to the parent.

Listing 7. Broadcasting in Kiama

```
1 val topNode1: CA[ASTNode, ASTNode] = attr {
2   case e: Program => e
3   case n => (n.parent[ASTNode])->topNode1
4 }
5
6 val topNode2: CA[ASTNode, ASTNode] = down {
7   case e: Program => e
8 }
```

In JastAdd a common way to compose multiple errors is to use collection attributes. Collection attributes are not directly supported by Kiama, but can be created using rewriting helper functions and attributes. Kiama's version of JastAdd's specific collection attribute syntax[11] (see listing 8) is to use partial functions, see listing 9. One feature that is missing in the Kiama example is the possibility to specify `N2-ref-exp`, but it is possible using `childAttr`, but one quickly notices that this does not integrate well with `collectAll`. In SimpliC this is used in order to generate semantic errors, such as type- and name errors. Using traits it is possible to stack multiple types errors using Scala's stackable trait pattern[13], see listing 10.

Listing 8. JastAdd collection attributes

```
1 N1 contributes value-exp
2 when cond-exp
3 to N2.a()
4 for N2-ref-exp;
```

Listing 9. Kiama collection attributes

```
1 val a: CA[N2, Seq[Exp]] = attr(collectAll {
2   case N1 if cond-exp => value-exp
3 })
```

Listing 10. Semantic Errors

```
1 trait Errors {
2   def err[T]: T ==> Messages = Map.empty
3   final val errors = attr(collectall(err))
4 }
5
6 trait NameAnalysisErrors extends Errors {
7   self: NameAnalysis =>
8
9   abstract override def err[T]: T ==> Messages = {
10     def errors: T ==> Messages = {
11       case n @ IdUse(name) if (n->decl == UnknownDecl())
         =>
12         message(n, s"$name␣is␣undeclared")
13
14       case n @ FuncCall(name, _) if (n->decl ==
         UnknownDecl()) =>
15         message(n, s"$name␣is␣undeclared")
16
17       case n: IdDecl if (n->isMultiplyDecl) =>
18         message(n, s"$n.name␣is␣multiply␣declared")%$
19     }
20
21     errors orElse super.err
22   }
23 }
```

2.4 Pretty Printing of the AST

Pretty printing in Kiama is based on a paper by Swierstra and Chitil[14], a functional and concise way of turning an AST into a layout. Using pattern matching on the AST and the supplied methods in Kiama the pretty printing on a `while` statement can be done as shown in listing 11. This kind of pretty printing resembles the way John Hughes describes the design of a pretty printing library[12]. Three concatenation operators available in Kiama are used in the implementation; `<@>` separates by newline, `<+>` separates by a space, and `<>` uses no separation.

Listing 11. Pretty printing

```
1   While(cmp, b) => Tokens.While <+> Tokens.LPar <> show(
      cmp) <> Tokens.RPar <+> show(b)
```

2.5 Interpretation using Abstract State Machine

Kiama implements classes to help with the construction of a state machine[16]. The classes provided for this are flexible enough in that they provide state objects and a method of executing code. In this paper, `Code` is defined as a sequence of instructions. These instructions in themselves are case classes that the user has defined. The `Machine` class handles the execution of these instructions.

In listing 12, the crucial parts of the ASM are shown. As can be seen, the machine executes instructions while `halt` is `false`. The function calls inside `execute` handle the execution of each instruction. If the instruction is not arithmetic, then `memory` will try to to handle it and so on. When the machine receives an exit instruction, it will set the `halt` state to `true`.

Listing 12. The Abstract State Machine

```
1 override def main { if (!halt) execute(code(PC)) }
2
3 def execute(instr: Instr) {
4   arithmetic(instr)
5   memory(instr)
6   ...
7 }
8
9 def arithmetic(instr: Instr) = instr match {
10   case MOV(a, im)    => R(a) := im
11   case ADD(a, b, c) => R(a) := R(b) + R(c)
12   ...
13   case _ => //Not an arithmetic instruction
14 }
```

When assigning a value to a state the special operator "`:=`" is used. The values won't be immediately updated, rather all the states assigned a value during the same instruction cycle will be updated at the end of the cycle. This point is illustrated in listing 13.

Listing 13. Illustration of state updating in ASM

```
1 PC := PC + 1 /* from PC to PC+1 at end of cycle */
2 RP := PC + 1 /* PC value hasn't been updated, RP becomes
     old PC + 1 */
```

It is conceivable that an advanced state machine emulating a processor could be built thanks to this behavior. For this paper, a simple stack based RISC was constructed. This RISC uses a couple of registers to ease the implementation. The full implementation is listed in appendix A. The registers store intermediate values for arithmetic operations and the return value from function calls. The stack is used to push and pop expressions.

2.6 Code generation

The code generation phase takes the top ASTNode, i.e. the program node and translates the tree directly to a sequence of instructions using attribute grammars. Initially the implementation didn't use attributes but rather Scala code directly. But, as the implementation became more advanced, the need for attributes became apparent.

2.7 Inlining functions using Rewriting

Kiama features strategy-based rewriting[17], heavily influenced by the Stratego/XT language and library[15][18]. Strategy-based rewriting uses different strategies in order to perform program transformations on the AST. There are multiple ways to create strategies in Kiama, but the `Strategy` class is simply a function which takes a `Term` and returns an `Option[Term]`, where `Term` is an arbitrary type, and `Option[Term]` can either be `Some(term)` (successful) or

None (failed). Rewriting a term is done by calling Kiama's function rewrite, which takes a strategy and a term as its parameters, it returns a possibly transformed Term if the strategy is successful, otherwise it returns the same term. There are three methods in which strategies are used in the SimpliC implementation; Basic strategies or predicates, where the strategy is used to test a specific term against a predicate, or used to obtain another term in the tree. In listing 14 the strategy tests a Stmt against the predicate that all of its children are of type Exp.

Listing 14. Strategy example

```
1 val lonelyStmt = strategy[Stmt] {
2   case stmt => {
3     if (stmt.children forall { _.isInstanceOf[Exp] })
4       Some(stmt)
5     else
6       None
7   }
8 }
```

The second method is Rewriting rules, where the strategy is used to transform a term. listing 15 is an example where each Return statement would be replaced by a Block of Assign and Goto.

Listing 15. Rule example

```
1 val retDecl = VarDecl(...) /* unique variable name */
2 val exitLabel = Label(...) /* unique label name */
3 val fixReturn = rule[Stmt] {
4   case Return(e) => {
5     Block(Nil, Seq(
6       Assign(IdUse(retDecl.name), e),
7       Goto(exitLabel)
8     ))
9   }
10 }
```

The last one is queries, which is when the strategy always returns the same term, and is only called for its side effects. The attribute in listing 16 returns every available FuncCall in the AST, by calling the query on every node, and modifying a variable depending on the type of node.

Listing 16. Query example

```
1 val calls: CA[ASTNode, Seq[FuncCall]] = {
2   val calls = mutable.ListBuffer[FuncCall]()
3   val hasCalls = query[ASTNode] {
4     case f: FuncCall => calls += f
5   }
6   rewrite(everywhere(hasCalls))(n)
7   calls.toList
8 }
```

It should be noted that the last two of these constructs make use of Scala's partial functions, and a fail to match would return None. The everywhere function is used to apply the query on every node in the AST.

Kiama supplies three methods in the Strategy class in order to compose multiple strategies. Sequential composition p <* q first applies p and then q. Deterministic composition p <+ q applies p or q, depending on the result of p. Non-deterministic composition p + q, which would apply either p or q first, but is current implemented as deterministic composition, and is therefore equivalent to <+.

The rewriting support in Kiama also supplies multiple functions in order to traverse the AST in different ways. The functions relevant to this implementation are the following; repeat applies a strategy as long as it succeeds, oncetd applies a strategy at least once in a top-down manner, everywhere applies the strategy on every node, count maps a term to an integer and calculates the sum of all values, and leaves applies the strategy to every node which is a leaf according to a supplied predicate.

The application of rewriting in SimpliC is inlining of function calls. The inlining is one-pass and inlines every found function call, but it cannot inline function calls inside while and if condition expressions. A strategy inline is applied to every leaf in the program, defined as a Stmt containing children only of type Exp (this effectively filters out if and while), as well as requiring that there exists a child of type FuncCall.

Listing 17. Count function calls

```
1  val hasFuncCall = strategy[Stmt] {
2    case stmt => {
3      val q = count {
4        case f: FuncCall => 1
5      }
6
7      if (q(stmt) > 0) Some(stmt) else None
8    }
9  }
10 { ... }
11 val optStrategy = leaves(inline, lonelyStmt <*
     hasFuncCall)
```

Every Stmt satisfying the predicate in listing 17 will use an attribute inlined on FuncCall and rewrite every use of the function call with a variable use created by the inlining. The transformation can be seen in listing 18. Note that index is a method available in Kiama which returns the index of an AST node in the parent's children list.

Listing 18. Rewrite function calls

```
1 val fixCalls = rule[Exp] {
2   case f @ FuncCall(name, _) => IdUse(s"inl_ret_${name}
    _${f.index}_${f.hashCode}")%$
3 }
4
5 val rwStmt = rewrite(repeat(oncetd(fixCalls)))(stmt)
```

The inlining attribute returns a Block of code to be inserted before the statement containing the function call, as well as a sequence of declarations to be used outside the block code. The block is prepended by zero or more assignments to renamed parameter variables. In this step rewriting is used on two occasions, replacing return statements as well as renaming parameter id uses. The rule can be viewed in listing 19.

Listing 19. Rewrite 'returns'

```
1  val retDecl = VarDecl(...) /* unique variable name */
2  val exitLabel = Label(...) /* unique label name */
3  val fixReturn = rule[Stmt] {
4    case Return(e) => {
5      Block(Nil, Seq(
6      Assign(IdUse(retDecl.name), e),
7      Goto(exitLabel)
8      ))
9    }
10 }
11
12 val fixIduse = rule[Exp] {
13   case n @ IdUse(name) if n->decl->isParam => IdUse(s"
     inl_par_${fname}_${f.index}_${name}_${f.hashCode}")
14 }
15
16 val block = rewrite[Block](everywhere(fixIduse) <*
     everywhere(fixReturn))(fblock)
```

This kind of inlining results in the following transformation of a simple function call, see listing 20 and listing 21.

Listing 20. Original

```
1 int f(int i) {
2   i = 22;
3   return i + 2;
4 }
5
6 int main() {
7   return f(4);
8 }
```

Listing 21. Inlining result

```
1  int main() {
2    {
3      int inl_par_f_0_i_14290345;
4      int inl_ret_f_0_14290345;
5      {
6        inl_par_f_0_i_14290345 = 4;
7        {
8          inl_par_f_0_i_14290345 = 22;
9          {
10           inl_ret_f_0_14290345 = inl_par_f_0_i_14290345
     + 2;
11           goto '$n'
12         }
13       }
14       label '$n'
15     }
16     return inl_ret_f_0_14290345;
17   }
18 }
```

3. Evaluation of the Kiama Library

3.1 Parsing

Parser combinators are well known to people who have a background in functional languages. Thanks to Scala's flexible syntax it is easy to emulate an EBNF like parser. Scala's parser combinators work well and are easy to use. Not having to learn a separate syntax for the parser is great. The parsers are also a great tool for uniform implementation. The parser combinators force the user to define the parsing rules in a specific way which would be the same no-matter who implemented the parsing.

3.2 Building the Abstract Syntax Tree in Kiama

Passing the root node to the `initTree` function is simple enough, but it is surprising for Kiama to deviate from the functional way of handling things, and choosing this imperative approach. This, however, has its reasons. Originally the `Attributable` class initialized node properties in the constructor[8]. Said approach proved to be unstable due to the problem of keeping track of when nodes were constructed in relation to their parents and children. As such, the `initTree` method was introduced to make the initialization explicit. An option could be to wrap the context where an attributed Abstract Syntax Tree is needed.

As shown in listing 3, Kiama recommends mixing in the trait `TreeNode` to any class that should be a part of the tree. When looking at the implementation, however, it turns out that `TreeNode` is simply a trait inheriting `Attributable`. In the *Lightweight Language Processing In Kiama* paper the author recommends using the trait `Attributable` for all prospective Abstract Syntax Tree nodes[15].

3.3 Reference Attribute Grammars in Kiama

The way one would go about implementing reference attributes in Kiama is really good. This is largely thanks to how matching works in Scala. Being able to match first on a node and then on the parent node in the `childAttr` attribute is especially handy. This results in very easily read code, it is also easy to see to which nodes the attribute will apply. Depending on how much code is needed for each case, the code can be made very concise.

As aforementioned, in JastAdd the static interpretation had to be designed such that if an abstract type was reached, an exception would be thrown. The match statements in Kiama solve this at no extra charge - there is no need to implement the same construct in Kiama, a failing match will throw an exception.

Using an attribute is as simple as calling `a->b`. This, however, causes problems in nested namespaces, this is explained in listing 22.

Listing 22. Name clash example

```
1 lazy val attribute: CA[ASTNode, Value] = attr { ... }
2 ...
3 def foo(node: ASTNode) = {
4   val attribute = ...
5   node->attribute // this now chooses the val from foo!
6 }
```

The problem with this is that the arrow operator looks like it is being applied to a value inside `node`, but it's not. It simply finds the value with the same identifier closest in the scope and applies this to `node`.

The broadcasting feature in Kiama works well, and the fact that Kiama has decorators of various kinds make working with attributes much easier. The syntax is simple and the use of pattern matching is consistent with other attributes. A similar feature also available in Kiama, but not used in the SimpliC implementation, is forwarding. Forwarding is a way to access attributes residing in other trees. An example would be where a node in the AST needs to access an attribute in the code generation tree. `node->codegen->size`, this however, requires that the exact access path is specified. By specifying an implicit attribute, the `->` method will implicitly convert the node to the code generation node and then access the requested attribute, `node->size`. This way the attribute is implicitly forwarded to the node.

Listing 23. Forwarding in Kiama

```
1 implicit val codegen: CA[ASTNode, Code] = attr { ... }
2
3 val node = ....
4 node->codegen->size
5 <==>
6 node->size
```

Collection attributes are not directly supported, but the functionality that Kiama provides is sufficient in the SimpliC use case. It does integrate well with traits in Scala which makes it easy to compose and modularize the different contributors of messages.

A more comprehensive evaluation of how well Kiama attributes scale is covered in section 3.5, but it leaves out a comparison with JastAdd. In the paper by AM Sloane[9] there is an evaluation of the performance of Kiama versus the performance of JastAdd, which clearly shows that JastAdd performs better. The reason for the large gap is because of the difference in methods used for caching attributes in Kiama and JastAdd. JastAdd stores attributes in fields in the respective classes, since Kiama does not use a generator it cannot store attributes this way and instead uses a type of hash map stored in a separate object. The cost of the look-up in the hash map is a lot more expensive than a simple field access.

3.4 Interpretation using Abstract State Machine

As aforementioned the code generation to the pseudo RISC code is done using attributes, however, the execution inside the Abstract State Machine is done in a linear fashion. I.e. the program is actually executed instruction after instruction. In contrast to JastAdd where the interpretation is done offline. As mentioned the state machine is a concept that can be used to emulate more advanced structures such as a pipelined processor. The implementation proposed in this paper is somewhat naive and doesn't make use of immediate instructions.

There is a lack of support for the visitor pattern in Kiama. This, however, is because of the visitor pattern being an object oriented design pattern. In Scala, it is instead encouraged to use pattern matching. Martin Odersky, the language's creator states in an interview[20] that the concept of visitor pattern is a way to accomplish some of the things that can be done using pattern matching. He also proclaims that the use of the visitor pattern is inefficient when compared to pattern matching using modern virtual machine technology.

3.5 Rewriting in Kiama

The foundation of the rewriting in Kiama is easy to grasp since all features can be reduced back to the simple `Strategy` class. The vast amount of helper functions available in Kiama makes the traversal and rewriting of the tree much easier and the Scala syntax further simplifies the implementation.

It is stated in the Kiama Wiki[17] that one must avoid multiple references to the same node in the AST, since attributes are bound to the instance of nodes. This means that when using rewriting one must explicitly copy a node when multiple references can occur, which means that usability is greatly reduced in these cases. The authors of this paper did stumble upon the problem of applying rewriting to sub trees of an AST. Applying rewrite to sub trees breaks some of the features supplied by `initTree`, such as parent node references and possibly other features. It is unclear whether it is good practice or not to apply nested rewrite, where one rewrite rule rewrites another node using another rule. Nested rewriting does not seem to break any parent-child relationship in the AST, most likely because the outer rewrite will fix parent-child references.

In order to evaluate how well rewriting integrates with attributes a simple inlining benchmark is used, see listing 24. The benchmark recursively calculates the faculty of an integer. Since the benchmark is recursive it means that it can be inlined as many times as needed.

Listing 24. Benchmark

```
1  int f(int i) {
2    if (i == 0) {
3      return 1;
4    } else {
5      return i * f(i - 1);
6    }
7  }
8  int main() {
9    return f(12);
10 }
```

The running time of three phases of the implementation is measured; inlining of the AST, semantic analysis and code generation. The AST is inlined 50 times and each intermediate AST is saved, afterwards the resulting ASTs are used in the other phases, and again measured. The generated code is also being run in the ASM to check that it is still valid, but this is not measured. Two variables are changed in order to create four different setups; reset (or flush) attributes before each phase and/or deep copy the AST after inlining. Flushing attributes makes sure that each phase in the implementation will be independent, and copying will make sure that no attributes are shared between independent ASTs. The size of the generated code grows linearly with a delta of 37 instructions per pass. The measurements can be viewed in figure 1, 2 and 3, and the 95% confidence intervals in figure 4.

One problem with this approach is that code generation in case of no flush nor copying is incorrect. This problem is solved by either copying or flushing, since the origin of the problem is most likely because of attributes and nodes residing in multiple trees. This is very much a usability problem in Kiama since one would expect different instances of a rewritten AST to be unique and independent.

During the inlining phase there is very small difference between no-flush/no-copy and copy/no-flush, most likely because there is a very low amount of nodes shared between the ASTs with attributes that are cached. During the semantic phase there is a larger gap between the different configurations, flushing is the slowest as predicted, but an interesting observation is that the copy is undeniably faster than flush. This can be explained by that different phases are using the same attributes, and therefore some values are cached from the inlining phase. This observation is even more noticeable in the code generation phase, where the same argument applies.

It was attempted to use a larger number of inlining passes in order to achieve a better performance analysis, but stack overflow problems occurred. There is no reason to believe that the stack overflow is caused by Kiama, it is more likely because of the way the performance measurements are structured. By optimizing the performance measurements one might have been able to achieve a larger number of passes.

Another observation regarding these measurement is that flush dominates copy, i.e. there is no noticeable difference between only using flush and using flush and copy. This makes sense since it does not matter if there are shared nodes between the ASTs since the attributes will not be cached in between calculations.

Finally, this shows that Kiama very much can make use of attributes and its performance is dependent on it. It does however come with a few problems, especially regarding copies and rewriting, it is hard to determine if copies are required or not. In this implementation the problem only occurred when performing performance measurements, since this was the only case where intermediate results of multiple inline passes where used, i.e. when nodes where shared. In the general case Kiama has to be able to automatically flush the attributes of nodes that require it.

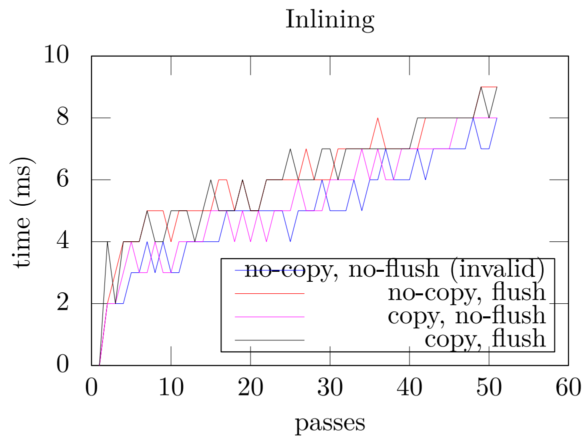

Figure 1. Inlining measurements

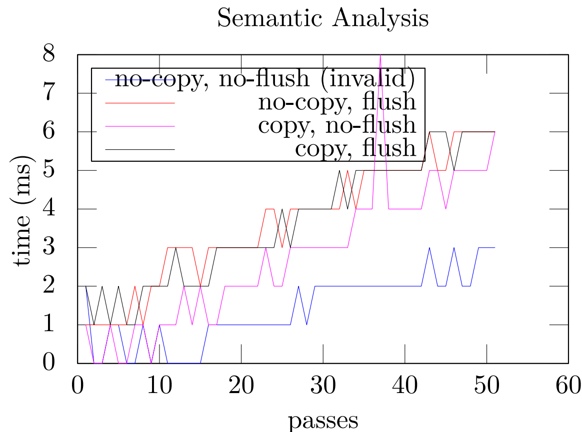

Figure 2. Semantic analysis measurements

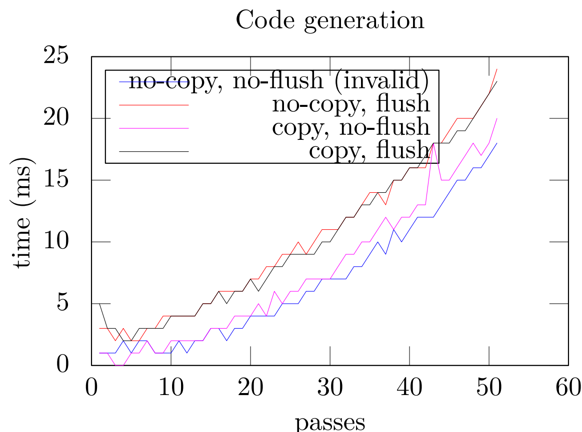

Figure 3. Code generation measurements

Phase	Config		Interval	
	Copy	Flush	Start	End
Inlining	-	-	4.5260 *	5.4740 *
	-	x	5.5790	6.5779
	x	-	4.7618	5.7872
	x	x	5.5425	6.5359
Semantic	-	-	1.1339 *	1.6504 *
	-	x	3.2416	4.1309
	x	-	2.2393	3.2509
	x	x	3.2178	4.1155
Code generation	-	-	5.2505 *	8.1220 *
	-	x	8.5004	12.0094
	x	-	5.8733	9.1463
	x	x	8.3549	11.8020

Figure 4. 95% confidence intervals of measurements. **Invalid result*

4. Conclusion

Parsing in Kiama is convenient if one has a background in functional programming, it should, however, be noted that the parsing is part of Scala - not Kiama.

Using case classes to build the AST is convenient and feels like a built in feature in the language.

Attributes work especially well - thanks to the match construct. Mainly because the compiler will issue a warning if it suspects that the match may fail and a run-time error will be thrown if it actually fails during run-time.

The authors are split in their opinions as to whether the arrow (“`->`”) method is good or bad. The name-clashing issue is avoidable by good naming conventions, but in a bigger project there still exists a possibility that this will be overlooked. Albeit, this will most likely cause compile-time errors.

The ASM feature in Kiama is well thought out and there were no issues with implementing a minimal RISC.

Rewriting in Kiama works somewhat well, it is based on a good language and library, but is unable to implement this properly. The need to explicitly copy nodes is an usability problem and might be a deal-breaker for projects that require a large amount of rewriting.

The performance of Kiama is worse than JastAdd. Considering that Kiama is a library and does not support any kind of preprocessing, the performance is expected to be worse than those tools. Projects using a large amount of attributes might encounter performance problems in Kiama.

Kiama together with Scala offer a simpler way of getting started with compiler construction, albeit the user has to be familiar with Scala. The need for only one library, which can be loaded via Scala Build Tool, is great. Especially when compared with the plethora of tools needed for JastAdd to perform the same tasks. With just one library (with internal dependencies) and syntax, Kiama manages to do most of the things available from JastAdd, Beaver and JFlex.

Acknowledgments

In closing the authors would like to thank Christoff Bürger and “The Dog Catcher.”

References

[1] Kiama Wiki (2014-11-10) - http://code.google.com/p/kiama/

[2] JastAdd home page(2014-11-10) - http://jastadd.org/web/

[Parsing Expression Grammars: A Recognition-Based Syntactic Foundation] Bryan Ford's original paper on PEGs page (2015-01-20) - http://bford.info/pub/lang/peg.pdf

[4] The xtc (eXTensible Compiler) project hom page (2015-01-20) - http://cs.nyu.edu/rgrimm/xtc/

[5] Beaver home page(2014-11-10) - http://beaver.sourceforge.net/

[6] JFlex home page(2014-11-10) - http://jflex.de/

[7] Scala Wiki (2014-12-04) - http://www.scala-lang.org/files/archive/api/2.11.2/scala-parser-combinators/#scala.util.parsing.combinator.PackratParsers

[8] Kiama Wiki - Attribution. (2014-12-11). - https://code.google.com/p/kiama/wiki/Attribution

[9] AM. Sloane, LCL. Kats, E. Visser: A pure embedding of attribute grammars. Science of Computer Programming 78 (10), 1752-1789. - http://wiki.kiama.googlecode.com/hg-history/v1.2.0/papers/SCP11.pdf

[10] G. Hedin: An Introductory Tutorial on JastAdd Attribute Grammars. Generative and Transformational Techniques in Software Engineering III. pp 166-200. Springer Berlin Heidelberg. 2011.

[11] JastAdd Reference Manual (2014-12-17) - http://jastadd.org/web/documentation/reference-manual.php#Collection

[12] J. Hughes: The design of a pretty-printing library. Advanced Functional Programming. pp 53-96. Springer Berlin Heidelberg. 1995.

[13] B. Venners: Scala's Stackable Trait Pattern. 2015-01-06 - http://www.artima.com/scalazine/articles/stackable_trait_pattern.html

[14] S. Doaitse Swierstra, O Chitil: Journal of Functional Programming / Volume 19 / Issue 01 / January 2009, pp 1-16. Cambridge University Press 2008. - http://dx.doi.org/10.1017/S0956796808006990

[15] AM. Sloane, LCL. Kats, E. Visser: Lightweight Language Processing in Kiama, 1752-1789. Generative and Transformational Techniques in Software Engineering III. pp 408-425. Springer Berlin Heidelberg. 2011.

[16] Kiama Wiki - Machines. (2014-12-11) - https://code.google.com/p/kiama/wiki/Machines

[17] Kiama Wiki - Rewriting. (2014-12-11) - https://code.google.com/p/kiama/wiki/Rewriting

[18] Stratego/XT (2014-12-08) - http://strategoxt.org/

[19] E. Visser: Program Transformation with Stratego/XT. Domain-Specific Program Generation. pp 216-238. Springer Berlin Heidelberg. 2004.

[20] The Point of Pattern Matching in Scala. (2014-12-11) - http://www.artima.com/scalazine/articles/pattern_matching.html

A. Project Code Listings

Listing 25. Tokens.scala

```
package se.lth.cs.edan70.simplic

object Tokens {
  val LBracket = "{"
  val RBracket = "}"
  val LPar = "("
  val RPar = ")"
  val Semicolon = ";"
  val Comma = ","
  val Assign = "="

  val Equals = "=="
  val NotEquals = "!="
  val GreaterThan = ">"
  val GreaterEquals =  ">="
  val LessThan = "<"
  val LessEquals = "<="
  val Add = "+"
  val Sub = "-"
  val Mul = "*"
  val Div = "/"
  val Mod = "%"

  val Return = "return"
  val While = "while"
  val If = "if"
  val Else = "else"
  val Int = "int"
  val Bool = "bool"

  val Num = "[0-9]+".r
  val Var = "[a-zA-Z][a-zA-Z0-9]*".r
  val Special = "[^a-zA-Z0-9]".r
}
```

Listing 26. AST.scala

```
package se.lth.cs.edan70.simplic

object AST {
  import org.kiama.util.TreeNode
  import org.kiama.attribution.Attributable
  import scala.collection.immutable.Seq

  sealed abstract class ASTNode extends TreeNode
  type Id = String

  abstract class Type(val typeName: String) extends ASTNode
  case object IntT extends Type("int")
  case object BoolT extends Type("bool")
  case object UnknownT extends Type("UnknownType")

  case class Program(decls: Seq[FuncDecl]) extends ASTNode
  abstract class IdDecl extends ASTNode {
    def name: Id
  }
  case class VarDecl(name: Id) extends IdDecl
  case class FuncDecl(name: Id,
                      params: Seq[Param],
                      block: Block) extends IdDecl
  case class UnknownDecl() extends IdDecl {
    val name = "unknown"
  }
  case class Param(name: Id) extends IdDecl

  abstract class Stmt extends ASTNode
  case class Label(name: String) extends Stmt
  case class Goto(label: Label) extends Stmt
  case class Block(decls: Seq[VarDecl], stmts: Seq[Stmt]) extends Stmt
  case class Assign(lhs: IdUse, rhs: Exp) extends Stmt
  case class Return(exp: Exp) extends Stmt
  case class If(cmp: CompExp, thn: Block, el: Option[Block]) extends Stmt
  case class While(cmp: CompExp, block: Block) extends Stmt
  case class ExpStmt(exp: Exp) extends Stmt

  abstract class Exp extends ASTNode
  case class Num(i: Int) extends Exp
```

```
41   case class IdUse(name: Id) extends Exp
42   case class FuncCall(name: Id, param: Seq[Exp]) extends Exp
43
44   abstract class BinExp() extends Exp {
45     def l: Exp
46     def r: Exp
47   }
48
49   case class Add(l: Exp, r: Exp) extends BinExp
50   case class Sub(l: Exp, r: Exp) extends BinExp
51   case class Mul(l: Exp, r: Exp) extends BinExp
52   case class Div(l: Exp, r: Exp) extends BinExp
53   case class Mod(l: Exp, r: Exp) extends BinExp
54
55   abstract class CompExp extends BinExp
56   case class EqExp(l: Exp, r: Exp) extends CompExp
57   case class NeExp(l: Exp, r: Exp) extends CompExp
58   case class GeExp(l: Exp, r: Exp) extends CompExp
59   case class GtExp(l: Exp, r: Exp) extends CompExp
60   case class LeExp(l: Exp, r: Exp) extends CompExp
61   case class LtExp(l: Exp, r: Exp) extends CompExp
62 }
```

Listing 27. SyntaxAnalyser.scala

```
1 package se.lth.cs.edan70.simplic
2
3 import org.kiama.util.PositionedParserUtilities
4 import scala.language.postfixOps
5 import scala.util.parsing.combinator.PackratParsers
6 import scala.collection.immutable.Seq
7 import AST._
8
9 trait SyntaxAnalyser extends PositionedParserUtilities {
10   lazy val parser: Parser[ASTNode] = phrase(program)
11
12   lazy val program: PackratParser[Program] = (funcDecl*) ^^ Program
13
14   lazy val stmt: PackratParser[Stmt] =
15     ifStmt | whileStmt | block | assign | retStmt | exprStmt
16
17   lazy val block: PackratParser[Block] =
18     Tokens.LBracket ~> ((intDecl*) ~ (stmt*)) <~ Tokens.RBracket ^^ Block
19
20   lazy val assign: PackratParser[Stmt] = idUse ~ ((Tokens.Assign ~> exp) <~ Tokens.Semicolon) ^^ Assign
21   lazy val retStmt: PackratParser[Stmt] = Tokens.Return ~> exp <~ Tokens.Semicolon ^^ Return
22   lazy val exprStmt: PackratParser[Stmt] = exp <~ Tokens.Semicolon ^^ ExpStmt
23   lazy val whileStmt: PackratParser[Stmt] =
24     Tokens.While ~> (Tokens.LPar ~> cmp <~ Tokens.RPar) ~ block ^^ While
25   lazy val ifStmt: PackratParser[Stmt] =
26     Tokens.If ~> (Tokens.LPar ~> cmp <~ Tokens.RPar) ~ block ~ (Tokens.Else ~> block?) ^^ If
27
28   lazy val funcDecl: PackratParser[FuncDecl] =
29     Tokens.Int ~> id ~ (Tokens.LPar ~> repsep(param, Tokens.Comma) <~ Tokens.RPar) ~ block ^^ FuncDecl
30
31   lazy val param =
32     Tokens.Int  ~> id ^^ Param |
33     Tokens.Bool ~> id ^^ Param
34
35   lazy val intDecl: PackratParser[VarDecl] = Tokens.Int ~> id <~ Tokens.Semicolon ^^ VarDecl
36
37   lazy val cmp: PackratParser[CompExp] =
38     exp ~ (Tokens.Equals ~> exp) ^^ EqExp |
39     exp ~ (Tokens.NotEquals ~> exp) ^^ NeExp |
40     exp ~ (Tokens.GreaterEquals ~> exp) ^^ GeExp |
41     exp ~ (Tokens.GreaterThan  ~> exp) ^^ GtExp |
42     exp ~ (Tokens.LessEquals ~> exp) ^^ LeExp |
43     exp ~ (Tokens.LessThan  ~> exp) ^^ LtExp
44
45   lazy val exp: PackratParser[Exp] =
46     cmp |
47     exp ~ (Tokens.Add ~> term) ^^ Add |
48     exp ~ (Tokens.Sub ~> term) ^^ Sub |
49     term
50
51   lazy val term: PackratParser[Exp] =
52     term ~ (Tokens.Mul ~> factor) ^^ Mul |
53     term ~ (Tokens.Div ~> factor) ^^ Div |
54     term ~ (Tokens.Mod ~> factor) ^^ Mod |
55     factor
56
```

```
57   lazy val factor: PackratParser[Exp] = funcCall | idUse | num | Tokens.LPar ~> exp <~ Tokens.RPar
58
59   lazy val idUse = id ^^ IdUse
60   lazy val num = Tokens.Num ^^ (s => Num(s.toInt))
61   lazy val funcCall: PackratParser[Exp] = id ~ (Tokens.LPar ~> repsep(exp, Tokens.Comma) <~ Tokens.RPar) ^^ FuncCall
62
63   lazy val id: PackratParser[String] = not(keyword) ~> Tokens.Var
64
65   lazy val keyword = keywords(Tokens.Special,
66     Seq(Tokens.While, Tokens.If, Tokens.Int, Tokens.Bool))
67
68   def ast(s: String): Program = parse(parser, s).get.asInstanceOf[Program]
69 }
```

Listing 28. ASTPrinter.scala

```
1 package se.lth.cs.edan70.simplic
2
3 import org.kiama.output.PrettyPrinter
4
5 trait ASTPrinter extends PrettyPrinter {
6   import AST._
7
8   def prettyAST(t: ASTNode): Layout = pretty(show(t))
9
10   def show(t: ASTNode): Doc = t match {
11     case Program(decls) => decls.foldLeft(empty) {
12       (doc, decl) => doc <@> line <> show(decl)
13     }
14
15     case VarDecl(name) => Tokens.Int <+> name <> Tokens.Semicolon
16
17     case FuncDecl(name, params, block) => {
18       val parameters = list(params, "", show)
19       Tokens.Int <+> name <> parameters <+> show(block)
20     }
21
22     case Param(n) => Tokens.Int <+> n
23
24     /* Statements */
25     case Block(decls, stmts) => {
26       val docdecls = decls.foldLeft(empty) {
27         (doc, decl) => doc <@> show(decl)
28       }
29
30       val docstmts = stmts.foldLeft(empty) {
31         (doc, stmt) => doc <@> nest(show(stmt))
32       }
33
34       Tokens.LBracket <> nest(docdecls <> docstmts) <@> Tokens.RBracket
35     }
36
37     case Assign(id, e) => show(id) <+> Tokens.Assign <+> show(e) <> Tokens.Semicolon
38     case Return(e) => Tokens.Return <+> show(e) <> Tokens.Semicolon
39
40     case If(cmp, t, e) => {
41       val el = e map { empty <+> Tokens.Else <+> show(_) } getOrElse { empty }
42       Tokens.If <+> Tokens.LPar <> show(cmp) <> Tokens.RPar <+> show(t) <> el
43     }
44
45     case While(cmp, b) => Tokens.While <+> Tokens.LPar <> show(cmp) <>
46       Tokens.RPar <+> show(b)
47
48     case Goto(l) => "goto" <+> show(l)
49
50     case Label(n) => s"label '$n'"
51
52     case ExpStmt(e) => show(e) <> Tokens.Semicolon
53
54     /* Exp */
55     case Num(i) => value(i)
56     case IdUse(n) => value(n)
57     case FuncCall(n, params) => list(params, n, show)
58
59     case e: Add => showBinExp(e, Tokens.Add)
60     case e: Sub => showBinExp(e, Tokens.Sub)
61     case e: Mul => showBinExp(e, Tokens.Mul)
62     case e: Div => showBinExp(e, Tokens.Div)
63     case e: Mod => showBinExp(e, Tokens.Mod)
64     case e: EqExp => showBinExp(e, Tokens.Equals)
65     case e: NeExp => showBinExp(e, Tokens.NotEquals)
```

```
66      case e: GeExp => showBinExp(e, Tokens.GreaterEquals)
67      case e: GtExp => showBinExp(e, Tokens.GreaterThan)
68      case e: LeExp => showBinExp(e, Tokens.LessEquals)
69      case e: LtExp => showBinExp(e, Tokens.LessThan)
70    }
71
72    def showBinExp(e: BinExp, op: String) = show(e.l) <+> op <+> show(e.r)
73  }
```

Listing 29. SemanticAnalysis.scala

```
1  package se.lth.cs.edan70.simplic
2
3  import AST._
4  import scala.collection.immutable.Seq
5  import org.kiama.==>
6  import org.kiama.attribution.Attribution._
7
8  trait SemanticAnalysis {
9    type CA[T, U] = CachedAttribute[T, U]
10   type CPA[T, U, V] = CachedParamAttribute[T, U, V]
11 }
12
13 trait NameAnalysis extends SemanticAnalysis {
14   val lookup: CPA[Id, ASTNode, IdDecl] = paramAttr {
15     case id => {
16       case n @ Program(decls) =>
17         (n->predef ++ decls) find { _.name == id } getOrElse { UnknownDecl() }
18
19       case n @ FuncDecl(_, params, _) =>
20         params find { _.name == id } getOrElse { (n.parent[ASTNode])->lookup(id) }
21
22       case n @ Block(decls, _) =>
23         decls find { _.name == id } getOrElse { (n.parent[ASTNode])->lookup(id) }
24
25       case n => (n.parent[ASTNode])->lookup(id)
26     }
27   }
28
29
30   val predef: CA[Program, Seq[FuncDecl]] = attr {
31     case n => Seq(
32       FuncDecl("print", Seq(Param("")), Block(Nil, Nil)),
33       FuncDecl("read", Nil, Block(Nil, Nil))
34     )
35   }
36
37   val decl: CA[ASTNode, IdDecl] = attr {
38     case n @ IdUse(name) => n->lookup(name)
39     case n @ FuncCall(name, _) => n->lookup(name)
40   }
41
42   /* isMultiplyDecl: by ref. */
43   val isMultiplyDecl: CA[IdDecl, Boolean] = attr {
44     id => id->lookup(id.name) ne id
45   }
46 }
47
48 trait TypeAnalysis extends SemanticAnalysis {
49   self: NameAnalysis =>
50   val tipe: CA[ASTNode, Type] = attr {
51     case _: CompExp => BoolT
52     case _: IdDecl | _: Num | _: BinExp => IntT
53     case n @ (IdUse(_) | FuncCall(_, _)) => n->decl->tipe
54   }
55
56   /* parent is not Exp, therefor ASTNode */
57   val expTipe: CA[ASTNode, Type] = childAttr {
58     case e: Exp => {
59       case _: BinExp => IntT
60       case _: Assign => IntT
61       case _: FuncCall => IntT
62       case _: Return => IntT
63       case _: ExpStmt => IntT
64       case If(`e`, _, _) => BoolT
65       case While(`e`, _) => BoolT
66       case _ => UnknownT
67     }
68
69     case _ => {
70       case _ => UnknownT
```

```
71     }
72   }
73 
74   val compTipe: CPA[Type, Type, Boolean] = paramAttr {
75     n1 => {
76       n2 => (n1, n2) match {
77         case (UnknownT, _) | (_, UnknownT) => true
78         case (n1, n2) => n1 == n2
79       }
80     }
81   }
82 
83   val correctParams: CA[ASTNode, Boolean] = attr {
84     case c @ FuncCall(_, params) => c->decl match {
85       case decl: FuncDecl => decl.params.length ==  params.length
86       case _ => false
87     }
88     case _ => false
89   }
90 
91   val isVariable: CA[ASTNode, Boolean] = attr {
92     case (_: VarDecl | _: Param) => true
93     case _ => false
94   }
95 
96   val isParam: CA[ASTNode, Boolean] = attr {
97     case (_: Param) => true
98     case _ => false
99   }
100 
101   val isFunction: CA[ASTNode, Boolean] = attr {
102     case _: FuncDecl => true
103     case _ => false
104 
105   }
106 }
```

Listing 30. Errors.scala

```
1 package se.lth.cs.edan70.simplic
2 
3 import AST._
4 import scala.collection.immutable.Seq
5 import org.kiama.==>
6 import org.kiama.attribution.Attribution._
7 import org.kiama.rewriting.Rewriter.{collect, collectall}
8 import org.kiama.util.Messaging.{check, checkuse, message, Messages, noMessages}
9 
10 /* Multiple Error checks are created using the
11    stackable trait pattern. The base trait
12    does not contain any checks. */
13 trait Errors {
14   def err[T]: T ==> Messages = Map.empty
15   final val errors = attr(collectall(err))
16 }
17 
18 trait NameAnalysisErrors extends Errors {
19   self: NameAnalysis =>
20 
21   abstract override def err[T]: T ==> Messages = {
22     def errors: T ==> Messages = {
23       case n @ IdUse(name) if (n->decl == UnknownDecl()) =>
24         message(n, s"$name␣is␣undeclared")
25 
26       case n @ FuncCall(name, _) if (n->decl == UnknownDecl()) =>
27         message(n, s"$name␣is␣undeclared")
28 
29       case n: IdDecl if (n->isMultiplyDecl) =>
30         message(n, s"${n.name}␣is␣multiply␣declared")
31     }
32 
33     errors orElse super.err
34   }
35 }
36 
37 trait TypeAnalysisErrors extends Errors {
38   self: TypeAnalysis with NameAnalysis =>
39 
40   abstract override def err[T]: T ==> Messages = {
41     def errors: T ==> Messages = {
42       case n: Exp if (!(n->tipe->compTipe(n->expTipe))) =>
```

```scala
        message(n, s"${n->tipe} not compatible with ${n->expTipe}")

      case n @ FuncCall(name, _) if (!(n->decl->isFunction)) =>
        message(n, s"${name} is not a function")

      case n @ IdUse(name) if (!(n->decl->isVariable)) =>
        message(n, s"${name} is not a variable")

      case f: FuncCall if (!(f->correctParams)) =>
        message(f, s"${f.name} call has incorrect number of params")
    }

    errors orElse super.err
  }
}
```

Listing 31. InstructionSet.scala

```scala
package se.lth.cs.edan70.simplic

object InstructionSet {
  import org.kiama.util.Emitter
  import scala.collection.immutable.Seq

  type Code = Seq[Instr]
  type RegNbr = Int
  type Word = Int

  abstract class Instr

  abstract class LABEL extends Instr
  case class LABELOFFSET(disp: Int) extends LABEL
  case class LABELNAME(str: String) extends LABEL {
    override def toString = s"""LABELName("$str")"""
  }
  case class GOTO(label: LABELNAME) extends Instr

  def sayReg(reg: RegNbr): String = s"$$$reg"

  /* Arithmetic */
  case object NOP extends Instr
  case class PSH(src: RegNbr) extends Instr
  case class POP(dest: RegNbr) extends Instr
  case class MOV(dest: RegNbr, im: Int) extends Instr
  case class ADD(dest: RegNbr, src1: RegNbr, src2: RegNbr) extends Instr
  case class ADDI(dest: RegNbr, src1: RegNbr, src2: Int) extends Instr
  case class SUB(dest: RegNbr, src1: RegNbr, src2: RegNbr) extends Instr
  case class SUBI(dest: RegNbr, src1: RegNbr, src2: Int) extends Instr
  case class MUL(dest: RegNbr, src1: RegNbr, src2: RegNbr) extends Instr
  case class MULI(dest: RegNbr, src1: RegNbr, src2: Int) extends Instr
  case class MOD(dest: RegNbr, src1: RegNbr, src2: RegNbr) extends Instr
  case class MODI(dest: RegNbr, src1: RegNbr, src2: Int) extends Instr
  case class DIV(dest: RegNbr, src1: RegNbr, src2: RegNbr) extends Instr
  case class DIVI(dest: RegNbr, src1: RegNbr, src2: Int) extends Instr
  case class CMPI(src: RegNbr, offset: Int, dest: RegNbr) extends Instr

  /* Memory */
  case class LDW(dst: RegNbr, origin: RegNbr, im: Int) extends Instr
  case class STW(src: RegNbr, origin: RegNbr, im: Int) extends Instr

  /* Control */
  sealed abstract class Branch extends Instr {
    def label: LABEL
  }
  case class BAL(label: LABEL) extends Branch
  case class BEQ(op1: RegNbr, op2: RegNbr, label: LABEL) extends Branch
  case class BNE(op1: RegNbr, op2: RegNbr, label: LABEL) extends Branch
  case class BGT(op1: RegNbr, op2: RegNbr, label: LABEL) extends Branch
  case class BGE(op1: RegNbr, op2: RegNbr, label: LABEL) extends Branch
  case class BLT(op1: RegNbr, op2: RegNbr, label: LABEL) extends Branch
  case class BLE(op1: RegNbr, op2: RegNbr, label: LABEL) extends Branch
  case object RET extends Instr
  case class CALL(label: LABEL) extends Instr
  case class EXIT(ret: RegNbr) extends Instr

  /* IO */
  abstract class IO(name: String) extends Instr {
    def reg: RegNbr
    //override def toString: String = s"$name ${sayReg(reg)}"
  }
  case class READ(reg: RegNbr) extends IO("rd")
```

```
64   case class WRITE(reg: RegNbr) extends IO("wrd")
65
66 }
```

Listing 32. Translation.scala

```
1 package se.lth.cs.edan70.simplic
2
3 import AST._
4 import InstructionSet._
5 import scala.collection.immutable.Seq
6 import org.kiama.attribution.Attributable
7 import org.kiama.attribution.Attribution._
8 import org.kiama.util.TreeNode
9
10 trait Translation {
11   self: TypeAnalysis with NameAnalysis =>
12
13   val OP1 = 1
14   val OP2 = 2
15   val ANS = 3
16   val RETR = 4
17   val PC = 28
18   val FP = 29
19   val SP = 30
20
21   val paramIndex: CA[IdDecl, Int] = childAttr {
22     case p : Param => {
23       case FuncDecl(_, ps, _) => (ps indexOf p) - ps.length
24     }
25   }
26
27   val topFunc: CA[ASTNode, FuncDecl] = attr {
28     case f: FuncDecl => f
29     case n => (n.parent[ASTNode])->topFunc
30   }
31
32   val varIndex: CA[IdDecl, Int] = childAttr {
33     case d : VarDecl => {
34       case b: Block => {
35   val FuncDecl(_, _, block) = b->topFunc
36   ((block->allDecls) indexOf d)
37       }
38     }
39   }
40
41   val allDecls: CA[Attributable, Seq[VarDecl]] = attr {
42     case n: VarDecl  => Seq(n)
43     case o =>
44       (o.children.toList) flatMap { _->allDecls }
45   }
46
47   val translate: CA[ASTNode, Code] = childAttr {
48
49     case p: Program => parent => {
50       val code = Seq(CALL(LABELNAME("main")), EXIT(RETR)) ++
51       (p.decls flatMap { _->translate })
52       code
53     }
54
55     case f: FuncDecl => parent => {
56       val nbr = (f.block->allDecls).length
57       val push = Stream.continually(PSH(0)) take nbr
58       (LABELNAME(f.name) +: push) ++
59       (f.block.stmts flatMap { _->translate })
60     }
61
62     /* Expressions. */
63     case c: CompExp => parent => {
64       def transCmp(label: LABELNAME) = {
65         val ops = c.l->translate ++ (c.r->translate)
66         val cmp = c match {
67           case EqExp(_, _) => BNE(OP1, OP2, label)
68           case NeExp(_, _) => BEQ(OP1, OP2, label)
69           case GtExp(_, _) => BLE(OP1, OP2, label)
70           case GeExp(_, _) => BLT(OP1, OP2, label)
71           case LtExp(_, _) => BGE(OP1, OP2, label)
72           case LeExp(_, _) => BGT(OP1, OP2, label)
73         }
74         ops ++ Seq(POP(OP2), POP(OP1), cmp)
75       }
```

```
76
77      parent match {
78        case n: While => transCmp(LABELNAME(s"end:␣${n.hashCode}"))
79        case n: If => transCmp(LABELNAME(s"else:␣${n.hashCode}"))
80      }
81    }
82
83    case exp: Exp => parent => exp match {
84      case n @ IdUse(name) => {
85        val offset =
86          if (n->decl->isParam)
87            n->decl->paramIndex - 2
88          else
89            n->decl->varIndex
90
91        Seq(
92          LDW(ANS, FP, offset),
93          PSH(ANS)
94        )
95      }
96
97      case b: BinExp =>  {
98        val ops = b.l->translate ++ (b.r->translate)
99        val op = b match {
100         case Add(_, _) => ADD(ANS, OP1, OP2)
101         case Sub(_, _) => SUB(ANS, OP1, OP2)
102         case Mul(_, _) => MUL(ANS, OP1, OP2)
103         case Div(_, _) => DIV(ANS, OP1, OP2)
104         case Mod(_, _) => MOD(ANS, OP1, OP2)
105       }
106
107       ops ++ Seq(POP(OP2), POP(OP1), op, PSH(ANS))
108     }
109
110     case Num(i) => Seq(MOV(OP1, i), PSH(OP1))
111
112     case FuncCall("print", Seq(e)) => /* inline optimization */
113       e->translate ++ Seq(POP(OP1), WRITE(OP1))
114
115     case FuncCall(name, params) => {
116       val ps = params flatMap { _->translate }
117       val pops = (1 to params.length) map { _ => POP(OP1) }
118       (ps :+ CALL(LABELNAME(name))) ++ pops :+ PSH(RETR)
119     }
120   }
121
122   /* Statements */
123   case n @ Block(_, stmts) => parent => stmts flatMap { _->translate }
124   case n @ While(c, b) => parent => {
125     val startLabel = LABELNAME(s"start:␣${n.hashCode}")
126     val endLabel = LABELNAME(s"end:␣${n.hashCode}")
127     val comp = c->translate
128     val body = b.stmts flatMap { _->translate }
129     val ops = c.l->translate ++ (c.r->translate)
130     val ins = (startLabel +: comp) ++ body
131     ins ++ Seq(BAL(startLabel), endLabel)
132   }
133
134   case n @ If(c, t, e) => parent => {
135     val comp = c->translate
136     val elseLabel = LABELNAME(s"else:␣${n.hashCode}")
137     val endLabel = LABELNAME(s"end:␣${n.hashCode}")
138     val thn = t.stmts flatMap { _->translate }
139     val el = e.map { x =>
140       x.stmts flatMap { _->translate }
141     } getOrElse Nil
142
143     (comp ++ thn ++ Seq(GOTO(endLabel), elseLabel) ++ el) :+ endLabel
144   }
145
146   case Assign(id, exp: Exp) => parent => {
147     val offset =
148       if (id->decl->isParam)
149         id->decl->paramIndex - 2
150       else
151         id->decl->varIndex
152
153     exp->translate ++
154     Seq(
155       POP(ANS),
156       STW(ANS, FP, offset)
```

```
157       )
158     }
159
160     case ExpStmt(exp: Exp) => parent => exp->translate
161     case Return(exp: Exp) => parent => exp->translate ++ Seq(POP(RETR), RET)
162     case Label(str) => parent => Seq(LABELNAME(str))
163     case Goto(Label(str)) => parent => Seq(GOTO(LABELNAME(str)))
164   }
165
166   def labelize(code: Code): Code = {
167     def genLabel(src: Instr, dst: LABEL) = dst match {
168       case dst: LABELOFFSET => dst
169       case dst: LABELNAME => {
170         val index = code indexWhere { _ eq src }
171         val iLabel = code indexOf dst
172         LABELOFFSET(iLabel - index)
173       }
174     }
175
176     code map {
177       case n @ CALL(l)  => CALL(genLabel(n, l))
178       case n @ GOTO(l)       => BAL(genLabel(n, l))
179       case n @ BEQ(a, b, l) => BEQ(a, b, genLabel(n, l))
180       case n @ BNE(a, b, l) => BNE(a, b, genLabel(n, l))
181       case n @ BGT(a, b, l) => BGT(a, b, genLabel(n, l))
182       case n @ BGE(a, b, l) => BGE(a, b, genLabel(n, l))
183       case n @ BLT(a, b, l) => BLT(a, b, genLabel(n, l))
184       case n @ BLE(a, b, l) => BLE(a, b, genLabel(n, l))
185       case n @ BAL(l)        => BAL(genLabel(n, l))
186       case x                 => x
187     }
188   }
189
190   def translateProgram(p: Program): Code = {
191     import org.kiama.attribution.Attribution.initTree
192     initTree(p)
193     labelize(p->translate)
194   }
195 }
```

Listing 33. StateMachine.scala

```
1 package se.lth.cs.edan70.simplic
2
3 import org.kiama.machine.Machine
4 import org.kiama.util.{ Console, Emitter, StringEmitter }
5 import scala.collection.immutable.HashMap
6
7 import InstructionSet._
8
9 class RISC(code: Code, console: Console, emitter: Emitter = new StringEmitter)
10   extends Machine("RISC", emitter) {
11
12   override def debug: Boolean = false
13
14   val R    = new ParamState[RegNbr, Int]("R")
15   val PC   = R(28)
16   val FP   = R(29)
17   val SP   = R(30)
18
19   /* STATES */
20   val Mem   = new ParamState[Int, Int]("Mem") // Byte addressed store of words
21   val halt  = new State[Boolean]("halt") // Halt flag
22
23   /* EXECUTION */
24   override def init {
25     R(0) := 0
26     PC := 0
27     FP := 0
28     SP := 0
29     halt := false
30   }
31
32   override def main { if(!halt) execute(code(PC)) }
33
34   def execute(instr: Instr) {
35     if (debug) {
36       emitter emitln s"$name exec: $instr"
37       emitter emitln  s"info: sp $SP pc $PC fp $FP"
38     }
39
```

```
40    try {
41      arithmetic(instr)
42      memory(instr)
43      // Control function increases PC as necessary
44      control(instr)
45      inputOutput(instr)
46    } catch {
47      case e: Exception => {
48        emitter emitln s"Exception␣$e␣at␣$instr"
49
50        emitter emitln s"RISC.R␣=␣"
51        emitter emit "␣␣␣␣Map("
52        for (r <- R.keys.toList.sorted)
53          emitter emit s"$r␣->␣${R(r)},␣"
54        emitter emitln ")"
55
56        emitter emitln "RISC.Mem␣="
57        emitter emit "␣␣␣␣Map("
58        for (m <- Mem.keys.toList.sorted)
59          emitter emit s"$m␣->␣${Mem(m)},␣"
60        emitter emitln ")"
61
62        halt := true
63      }
64    }
65  }
66
67  def arithmetic(instr: Instr) = instr match {
68    case MOV(a, im)      =>  R(a) := im
69    case ADD(a, b, c)    =>  R(a) := R(b) + R(c)
70    case ADDI(a, b, im) =>  R(a) := R(b) + im
71    case SUB(a, b, c)    =>  R(a) := R(b) - R(c)
72    case SUBI(a, b, im) =>  R(a) := R(b) - im
73    case MUL(a, b, c)    =>  R(a) := R(b) * R(c)
74    case MULI(a, b, im) =>  R(a) := R(b) * im
75    case MOD(a, b, c)    =>  R(a) := R(b) % R(c)
76    case MODI(a, b, im) =>  R(a) := R(b) % im
77    case DIV(a, b, c)    =>  R(a) := R(b) / R(c)
78    case DIVI(a, b, im) =>  R(a) := R(b) / im
79
80    case _ => // Not an arithmetic instruction
81  }
82
83  def control(instr: Instr) = instr match {
84    case BEQ(a, b, LABELOFFSET(i)) if R(a) =:= R(b)     => PC := PC + i
85    case BNE(a, b, LABELOFFSET(i)) if !(R(a) =:= R(b)) => PC := PC + i
86    case BGT(a, b, LABELOFFSET(i)) if R(a) > R(b)       => PC := PC + i
87    case BGE(a, b, LABELOFFSET(i)) if R(a) >= R(b)      => PC := PC + i
88    case BLT(a, b, LABELOFFSET(i)) if R(a) < R(b)       => PC := PC + i
89    case BLE(a, b, LABELOFFSET(i)) if R(a) <= R(b)      => PC := PC + i
90    case BAL(LABELOFFSET(i)) => PC := PC + i
91
92    case CALL(LABELOFFSET(i)) => {
93      /* save PC and FP */
94      Mem(SP) := PC
95      Mem(SP + 1) := FP
96
97      SP := SP + 2
98      FP := SP + 2
99      PC := PC + i
100   }
101
102   case RET => {
103     SP := FP - 2
104     PC := Mem(FP - 2) + 1
105     FP := Mem(FP - 1)
106   }
107
108   case EXIT(reg)      => {
109     halt := true
110     emitter emit "Return␣code:␣"
111     emitter emit R(reg)
112     emitter.emitln()
113   }
114   case _         => PC := PC + 1
115 }
116
117 def memory(instr: Instr) = instr match {
118   case PSH(src: RegNbr) => {
119     Mem(SP) := R(src)
120     SP := SP + 1
```

```
121     }
122
123     case POP(dst: RegNbr) => {
124       if (SP =:= 0)
125         throw new RuntimeException("stack␣underflow")
126
127       R(dst) := Mem(SP - 1)
128       SP := SP - 1
129     }
130
131     case LDW(dst, origin, im) => R(dst) := Mem(R(origin) + im)
132     case STW(src, origin, im) => Mem(R(origin) + im) := R(src)
133     case _ => // Not a memory instruction
134   }
135
136   def inputOutput(instr: Instr) = instr match {
137     case READ(a)  => R(a) := console.readInt("Enter␣int␣value:␣")
138     case WRITE(a) => emitter emitln R(a)
139     case _ => // Not an IO instruction
140   }
141 }
```

Listing 34. Optimization.scala

```
1 package se.lth.cs.edan70.simplic
2
3 import scala.util.Random
4 import AST._
5 import scala.collection.mutable
6 import scala.collection.immutable.Seq
7 import org.kiama.attribution.Attributable
8 import org.kiama.attribution.Attribution._
9 import org.kiama.util.TreeNode
10
11 trait Optimization {
12   self: Translation with NameAnalysis with TypeAnalysis =>
13
14   import org.kiama.rewriting.Rewriter._
15   import org.kiama.rewriting._
16
17   val funcCalls: CA[Attributable, Seq[FuncCall]] = attr {
18     case node => {
19       val es = node.children.filter { _.isInstanceOf[Exp] }
20       node match {
21         case f: FuncCall => Seq(f)
22         case _ => es.toList flatMap { _->funcCalls }
23       }
24     }
25   }
26
27   val parentDecl: CA[Attributable, IdDecl] = attr {
28     case decl: FuncDecl => decl
29     case _: Program => UnknownDecl()
30     case n => n.parent[Attributable]->parentDecl
31   }
32
33   val lonelyStmt = strategy[Stmt] {
34     case stmt => {
35       if (stmt.children forall { _.isInstanceOf[Exp] })
36         Some(stmt)
37       else None
38     }
39   }
40
41   val hasFuncCall = strategy[Stmt] {
42     case stmt => {
43       val q = count {
44         case f: FuncCall => 1
45       }
46
47       if (q(stmt) > 0) Some(stmt) else None
48     }
49   }
50
51   val isMain = strategy[ASTNode] {
52     case node => {
53       val decl = node->parentDecl
54       if (decl.name == "main") Some(node) else None
55     }
56   }
57
```

```
58   val inlined: CA[FuncCall, (Seq[VarDecl], Block)] = attr {
59     case f => {
60       val FuncCall(name, params) = f
61       val FuncDecl(fname, fparams, fblock) = f->decl
62
63       val exitLabel = Label(s"inl_exit:␣${f.index}_${f.hashCode}")
64       val retDecl = VarDecl(s"inl_ret_${fname}_${f.index}_${f.hashCode}")
65
66       val paramDecls = fparams map {
67         case Param(name) => VarDecl(s"inl_par_${fname}_${f.index}_${name}_${f.hashCode}")
68       }
69
70       val assigns = params.zip(paramDecls) map {
71         case (e, VarDecl(name)) => Assign(IdUse(name), e)
72       }
73
74       val fixReturn = rule[Stmt] {
75         case Return(e) => {
76           Block(Nil, Seq(
77             Assign(IdUse(retDecl.name), e),
78             Goto(exitLabel)
79           ))
80         }
81       }
82
83       val fixIduse = rule[Exp] {
84         case n @ IdUse(name) if n->decl->isParam => IdUse(s"inl_par_${fname}_${f.index}_${name}_${f.hashCode}")
85       }
86
87       val block = rewrite[Block](everywhere(fixIduse) <* everywhere(fixReturn))(fblock)
88       val allDecls = paramDecls :+ retDecl
89       val allStmts = Block(Nil, (assigns :+ block) :+ exitLabel)
90       (allDecls, allStmts)
91     }
92   }
93
94   val inline = strategy[Stmt] {
95     case stmt: Stmt => {
96       val calls = stmt->funcCalls
97
98       val (decls, blocks) = calls.foldLeft((Seq.empty[VarDecl], Seq.empty[Block])) {
99         case ((decls, blocks), f) => {
100          val (fdecls, block) = f->inlined
101
102          (fdecls ++ decls, blocks :+ block)
103        }
104      }
105
106      val fixCalls = rule[Exp] {
107        case f @ FuncCall(name, _) => IdUse(s"inl_ret_${name}_${f.index}_${f.hashCode}")
108      }
109
110      val rwStmt = rewrite(repeat(oncetd(fixCalls)))(stmt)
111
112      Some(Block(decls.toList, blocks :+ rwStmt))
113    }
114  }
115
116  val optStrategy = leaves(inline, lonelyStmt <* hasFuncCall)
117  def opt[T <: ASTNode] = rewrite[T](optStrategy)(_)
118
119  def optimizeAST(p: Program): Program = {
120    import org.kiama.attribution.Attribution.initTree
121    initTree(p)
122    opt(p)
123  }
124 }
```