UNIVERSITY OF TWENTE

Alia Programming Language

Fedor Beets
s1227874
Campuslaan 27

Joost van Doorn
s1095005
Dalsteindreef 2404, Diemen

July 7, 2014

Introduction

The Alia programming language was built for the final project of the compiler construction course at the University of Twente. In this final project a programming language is specified and implemented in antlr (ANother Tool for Language Recognition). A compiler is built to translate code into a type of machine instructions. The type of machine instructions can vary from language to language. Over the first half of compiler construction several small pieces of the process of building a compiler are individually learned and tested, brought to culmination in the final project where the entire track is gone through and a usable language is produced.

This document serves as a specification of the Alia Programming Language (Alia for short) as well as an explanation of the inner workings of the Alia compiler. In it we will give a short description of the Alia Programming Language in the context of programming languages, explain some of the problems faced and solutions concoted during the construction of the Alia compiler, give a specification of the Alia language with the help of the syntax, context-constraints and semantics. Also in this document are the transformations that show how the symbols of the language are turned into JVM instructions, a description of all the auxiliary java code made for the compiler, a set of tests is described to give confidence in the correctness of the compiler and lastly conclusions are drawn from the project.

1 Alia Programming Language

The Alia programming language is an expression language with type inference. Alia code is compiled to Java bytecode and can be run using the Java Virtual Machine (JVM). As an expression language every statement has a return type. Functional statements such as print, read and conditional statements all return values. For example the print function may return a value which can be used in an assignment statement.

```
x = print(34) + 1 // x is assigned 35
```

Alia contains compound statement which are a series of statements with the last statement as return value, compound statements are used in conditional statements and can be explicitly used for scoping.

```
x = begin
     y = 3 // Only declared in this scope
     y + 1 // Return value for the compound statement
end
if y = true; y && x < 10 do // y here has a different
    scope
     print(y) // Print the value of y
     print('t')
end
```

Types in Alia are inferred and do not need explicit declarations. Types can be declared explicitly within an assignment statement, but are not required. In the current form of the programming language, type inference can always deduce the type of a declaration. However in an extended version of Alia with functions and procedures, explicit type declarations will be required in cases where type inference will not be able to deduce the type of the variable or function return type. Type inference makes programming easier, it reduces the work of the programmer by making explicit type declarations optional, and reduces the amount of code required for variable declarations. Type checking is maintained and the programmer still has the option to add type declarations if it helps clarify the code.

```
x = 54 : int // x is assigned 54, and is explicitly
    declared as an int
```

2 Problems and solutions

During the construction of the Alia compiler we ran into some problems, as is to be expected during any first time construction of a compiler. In this section we will explain some of the bigger problems that we faced as well as the solutions that we applied.

Scope definition: Because we do not have explicit declarations we cannot redefine a variable inside a new scope. There would be no way to distinguish between redefining a variable in a new scope and reassigning the value that was given to a variable to be used later. With a new variable that overwrites an old one for a temporary scope you would need to assign a new space in memory. We have decided that we want this in our programming language, this is because if you can redeclare a variable inside a new scope then you are only making it more confusing for yourself as a coder. On the other hand not being able to assign a new value to a variable in a new scope destroys a large part of the functionality of language. For these reasons we have chosen to leave it as is.

3 Syntax, context-constraints and semantics

The syntax of Alia is defined as follows:

```
program = (func_def | (statement end_statement) | \n)*;

statements = (statement (end_statement statements)? | \n statements)?;
statements_cond = (statement (end_statement statements)? | \n statements_cond )?;

statement = (expr_assignment | const_assignment) (; type)?
| while_stmnt
;
```

```
end_statement = \n | ";" | EOF;

expr_assignment = (exprPrint "=") expr_assignment
| expr
;

const_assignment = CONST exprPrint "=" primitive;

expr = expr1 ((or | "||") expr1)*;
expr1 = expr2 ((and | "&&") expr2)*;
expr2 = expr3 ((">" | ">=" | "<" | "<=" | "==" | "!=" )^ expr3)*;
expr3 = expr4 (("+" | "-")^ expr4)*;
expr4 = expr5 (("*" | "/" | "%")^ expr5)*;
expr5 = "!" operand | operand | expr_minus | expr_plus;
expr_minus = "-" operand;
expr_plus = "+" operand;
operand = read |
      print |
      if_stmnt |
      "(" expr ")" |
      compound_stmnt |
      primitive |
      func_exprPrint
;

compound_stmnt = begin statements end;

primitive = number | character | boolean;

func_exprPrint = exprPrint ( "(" exprlist? ")" )?;

while_stmnt = WHILE statements_cond DO statements END;

if_stmnt = IF statements_cond DO statements else_stmnt? END;

else_stmnt = ELSEIF statements_cond DO statements else_stmnt?
| (ELSE statements)
;

print = PRINT "(" exprlist ") ;
read = READ "(" varlist ") ;

varlist = exprPrint ("," exprPrint)*;
exprlist = expr ("," expr)*;

func_def = DEF exprPrint "(" varlist ")";
```

4 Translation rules

The translation rules for Alia to Java bytecode are shown here. Some details have been abstracted away in favor of readability, these details include specific label names, translation rules which are dependent on the type of the expression (such as print and read), and some specific rules where pop statements are included.

```
execute [I = E]
    expr [E]
    istore a // address of variable I
    exprPrint [I]

expr [while C do S end] =
    goto COND
    WHILE:
    execute [S]
    COND:
    execute [C]
    ifne WHILE

expr [if C do S E end] =
    execute [C]
    ifeq ELSE
    execute [S]
    goto NEXT
    ELSE:
    exprElse [E]
    NEXT:

exprElse [elseif C do S E]
    execute [C]
    ifeq ELSE
    execute [S]
    goto NEXT
    ELSE:
    exprElse [E]
    NEXT:


exprElse [else S] =
    execute [S]

expr [E1 O E2] =
    expr [E1]
    expr [E2]
    instruction [O] // The specific instruction, e.g. iadd etc.
```

```
expr [E1 OC E2] =
    expr [E1]
    expr [E2]
    if_icmp $+7 // Go to iconst_1 if it is true, this line contains the specific instructio
    iconst_0
    goto $+4 // Go to the line after iconst_1
    iconst_1

expr [-E]
    expr [E]
    ineg

expr [+E]
    expr [E]

expr [not E]
    expr[E]
    ifeq $+7
    iconst_0
    goto $+4
    iconst_1

expr [begin S end]
    execute [S]

print [S] =
    getstatic java/lang/System/out Ljava/io/PrintStream;
    execute [S]
    invokevirtual java/io/PrintStream/println(T)V

expr [print(S)] =
    getstatic java/lang/System/out Ljava/io/PrintStream;
    execute [S]
    istore_1
    iload_1
    invokevirtual java/io/PrintStream/println(T)V
    iload_1

expr [print(S, L)] =
    print [S]
    executePrint [L]

executePrint [S, L]
    print [S]
    executePrint [L]
executePrint [S]
    print [S]

read [] =
    invokestatic java/lang/System/console()Ljava/io/Console;
    invokevirtual java/io/Console/readLine()Ljava/lang/String;
    invokestatic java/lang/Type/parseType(Ljava/lang/String;)T

execute [read(I)] =
    read []
    istore_1
    iload_1
    istore a ; address of variable I
    execute [S]
    iload_1
execute [read(I, L)] =
    read []
    istore a ; address of variable I
    exprRead [L]

exprRead [I, L]
    read []
    istore a ; address of variable I
    exprRead [L]

exprRead [I]
    read []
    istore a ; address of variable I

execute [S \n S] =
    execute [S]
    execute [S]

execute [S ; S] =
    execute [S]
    execute [S]


execute [S] =
    expr [S]

exprPrint [I] =
    iload a ; address of variable I

operand [N] =
    number [N] // iconst n

program [S] =
    .class public filename.j ; target file
    .super java/lang/Object

    .method public \<init\>()V
       aload_0
       invokenonvirtual java/lang/Object/\<init\>()V
       return
    .end method

    .method public static main([Ljava/lang/String;)V
       .limit stack stackMax ; stackMax = maximum size of the stack
       .limit locals localSize ; localSize = amount of local variables required

       execute [S]

       return
    .end method
```

5 Java-code

The checker uses an auxiliary class CheckerAux that handles a large portion of the logic of the checking, such as if two types are the same. This class also declares variables and constants into the symbol table. CheckerAux also has methods to access the symbolTable so that it throws AliaExceptions instead of more general exceptions. The symbol table has a HashMap of Names, IdEntries and a scopestack that has all exprPrints declared on a scope. Like every symbolTable it keeps track of what exprPrints have been declared on what levels. The IdEntries also store information about whether the exprPrint is a constant and what type it is.

Most of the logic for type checking is implemented in CheckerAux, to do the type checking a set of type classes are used, such as _Int and _Bool. All of these classes inherit from _Type and have a string with their typename. We chose to make all types into distinct classes instead of an enum because this will allow for extension of say the _Int class with a _Float class or of the _Char class with a _String class. In this way we can more easily add addtional types to Alia and a future _Long and _Float could be compared using inheritence.

The code generation makes use of CodeGeneratorAux. This seperates some of the logic from the antlr files. In particular CodeGeneratorAux calculates what kind of java type can be used for any given number, this choice is explained in the problems section. To do this it uses the NumberType class, which acts as a container for a number of booleans so that they can be passed more elegantly. The other part that CodeGeneratorAux takes care of is the logic for the stack management, incrementing and decrementing the amount that is still to be pushed off the stack in the code generation.

For error handling AliaException and AliaTypeException are used. These exceptions are thrown in the checker when ever a type is violated. If there is a syntactical mistake then the classes generated by antlr will throw exceptions. For run time errors standard java exceptions are also used.

After the checking fase has been completed a decorated AST is returned. The decorated AST stores the type information that was found in the corresponding nodes, such as for all binary expressions. We also store the identifying numbers for all applied usages of exprPrints (except for constants which are replaced), these ascending numbers are gotten from the IdEntries using CheckerAux and are stored with the nodes, for later use in the code generation.

6 Tests

7 Conclusion

8 Appendices