UNIVERSITY OF TWENTE

Alia Programming Language

Fedor Beets
s1227874
Campuslaan 27

Joost van Doorn
s1095005
Dalsteindreef 2404, Diemen

July 8, 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.

String template expressions: StringTemplate does not allow you to evaluate an expression inside of a string template. This was purposefully implemented to stop you from putting a large part of the logic in the string template itself. The problem that we have with this is that there are issues that are specific to the creation of java bytecode that must now be evaluated in the antlr part of the program, and must be passed in the creation of any possible target platform. The first time this became a problem was what instructions to use for outputting any given number. A naive solution to this would be to always use the instruction for loading a large number like an integer. But java has special instructions for loading the numbers -1 through 5 and loading smaller numbers that fit on a byte or a short, so we wanted to use these. To solve this issue we created a function that calculated what type a number can fit into in the CodeGeneratorAux class which passes a number of booleans wrapped in a NumberType to the string template. We then use conditional templates to emit different instructions depending on the number to be put on the stack.

We chose to put the optimalisation of replacing all constants by their actual reference in the checker stage. The choice was made because in the checker there is already a list of declared variables, and java functions to do this. This made it very easy to implement this feature in the checker, and much more ugly to implement in the code generation. In an ideal world there would be a seperate stage inbetween the checker and the code generation that optimises the abstract syntax tree, but this one feature was easily done in the checker. We have also chosen to not support constants that can vary, being defined by another identifier. In the Alia programming language a constant can only be defined by a single primitive type.

3 Syntax, context-constraints and semantics

The concrete syntax of Alia is defined as follows, the abstract syntax follows afterwards:

```
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 = (identifier "=") expr_assignment
| expr
;

const_assignment = CONST identifier "=" 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_identifier
;

compound_stmnt = begin statements end;

primitive = number | character | boolean;

func_identifier = identifier ( "(" 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 = identifier ("," identifier)*;
exprlist = expr ("," expr)*;

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

3.1 Semantics and context constraints:

The semantics and context constraints are defined using the abstract syntax of the Alia language.

```
program = statement+;
```

The program 'C' is run by executing the command C.

```
statements = statement*;

statement =  WHILE statements DO statements END
| expr
;
```

The while statement 'while S1 do S2 end' is executed as follows. The statement S1 is evaluated, if its value is true then S2 is evaluated and the while statement is run again. If the value of S1 is false then the execution is completed. S1 must be of type boolean. This statement is of type void. Declarations made in S1 are valid in S1 and S2. The scope of declarations made in S2 is only S2. Any expression 'E' is executed.

```
expr = operand
|   expr OR expr
|   expr OR_ALT expr
|   expr AND expr
|   expr AND_ALT expr
|   expr EQ expr
|   expr NQ expr
|   expr LE expr
|   expr GE expr
|   expr GT expr
|   expr LT expr
|   expr PLUS expr
|   expr MINUS expr
|   expr TIMES expr
|   expr DIV expr
|   expr MOD expr
|  NOT operand)
| (PLUS_OP | MINUS_OP) operand
|  PRINT exprlist
|  READ varlist
|   IF statements
    DO statements
    (ELSEIF statements
DO statements)*
ELSE statements
|   COMPOUND statements
|   IDENTIFIER BECOMES expr (COLON type)?
|   CONST IDENTIFIER BECOMES primitive (COLON type)?
```

The expressions 'or', '||', 'and', '&&' preceded by E1 and followed by E2 are evaluated by performing a logical or (True iff E1 or E2) in case of 'or' and '||'. In the case of 'and' and '&&' it is evaluated by performing a logical and on the two expressions (True iff E1 and E2). E1 and E2 must be of type boolean. The type of the expression is Boolean. These are the logical operators.

The expression 'E1 == E2' is true iff E1 equals E2. 'E1 != E2' is true iff E1 is not equal to E2. 'E1 <= E2' true iff E1 is smaller than or equal to E2. 'E1 >= E2' is true iff E1 greater than or equal to E2. 'E1 > E2' is true iff E1 is greater than E2. 'E1 < E2' is true iff E1 is smaller than E2. Of all these comparative operators, E1 and E2 must be of the same type. The type of the expressions is Boolean. These are the comparitive operators.

The expression 'E1 + E2' is executed as E1 plus E2. 'E1 - E2' is E1 minus E2. 'E1 * E2' is E1 times E2. 'E1 / E2' is E1 divided by E2, E2 is not allowed to be zero. '

The operator O in '!O' is inverted. O must be of type boolean, the expression is of type boolean. The '+O' and '-O' are executed as follows. For '+O' nothing is done. For

'-O' the operand is negated. O must be of type Int, the expression is of type Int. These are the unary operators.

The previous expressions have the following priority, from highest to lowest. Unary operators (-, +, !), then *,/,

The expression 'print EL' is executed as follows. The expression list EL is evaluated. All evaluated expressions are then written to the output. The type of the expression is the type of EL. The expression 'read VL' is executed as follows. The variable list evaluated. For every variable a line is read from the input, the first character of this line is assigned as value to the variable. The type of the expression is the type of VL.

If statements of the form 'if S1 do S2 (elseif S3 do S4)* (else S5)?' are executed as follows. S1 is executed. If S1 is true, then S2 is evaluated. If S1 is false and there is an S3, then S3 is evaluted and if true S4 is executed. If the evaluated S3 is false there is another elseif statement then it is evaluated, same as an if statement is. If S1, S3 and all other elseifs have evaluated to false, then S5 is executed. If there is no S5, execution has completed. The type os S1 and S3 must be boolean. If there is no else part, the type of the statement is void. If there is an else part, then if all S2, S4, S5 are the same type, that is the type of the conditional statement. If S2, S4, S5 are not of the same type then the type of the statement is void. Special scope rules apply, a declaration in S1 or any S3 is valid in S2, S4, S5 as long as the declaration precedes the use. The expression 'COMPOUND S1' is executed as follows. S1 is executed. This expression is of type S1.

The assignment 'I = E (:T)?' is executed as follows. I is bound to the value yielded by the expression E. If T was included, T and E must be of the same type. The expression is of type E. I can thereafter be used in applied occurrences, as an assignment generates a result it can be used as a 'sub-expression' in another statement or expression. The expression 'const I = P (:T)?' is executed as follows. I is bound to the value P. If T was included, T and E must be of the same type. The expression is of type P. I can be used in applied occurrences. I can not be assigned a different value at a later time.

```
operand =  IDENTIFIER
| primitive
;
```

The operand 'I' identifies a the value or variable bound to I. The operand I must have been previously declared. The type of the operand is the type of that value or variable. A primitive is one of the three primitive types NUMBER, CHAR_EXPR and BOOLEAN.

```
varlist = identifier (identifier)*;
```

The list 'I I*' evaluates to a list of identifiers. If there is one identifier the type of the list is the type of that identifier, and the result is its value. If there are 2 or more, the type is void and there is no result.

```
exprlist = expr (expr)*;
```

The list 'E E* evaluates to a list of expressions. None of the expressions may be void. If there is one expression, the list of the type of E, and the value is E. If there are 2 or more, the type is void, and thus there is no result.

```
primitive = NUMBER
| CHAR\_EXPR
| BOOLEAN
;
```

The operand 'N' evaluates to a number. N can be no larger than 2147483647 and no smaller than -2147483648. N is of type Int. The operand 'C' evaluates to a character. C is of type Char. The operand 'B' evaluates to a boolean, either true or false. C is of type Bool.

In Alia there are 4 types. 'int', 'char', 'boolean' and 'void'. If something is of type void it is an empty value that cannot be used.

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.

Pop lines A pop line is included after every statement that returns a value but has no higher expression using it. The amount of variables generated on the stack are counted by the compiler and after each complete statement the leftover expressions are popped.

Translation rules

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

identifier [I] =
    iload a // address of variable I

operand [I] =
    identifier [I]

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

operand [true] =
    iconst_1

operand [false] =
    iconst_0


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 identifiers declared on a scope. Like every symbolTable it keeps track of what identifiers have been declared on what levels. The IdEntries also store information about whether the identifier 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 identifiers (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

The Alia programming language has been thoroughly tested using a collection of test programs. These test programs have been designed to check the correct workings of parser, checker and compiler of the Alia programming language. Tests are located in the tests folder of the Alia project. Tests are divided in three categories.

1. Syntax: Test correct syntax of the
2. Context
3. Semantics: Tests which should trigger runtime errors.

6.1 Basic functionality test

The following test is designed to test the basic functionalities of the programming language. The functionalities tested in this test set are the compound statements, the read and print statements and the assignment statement.

```
ivar = begin
    ivar1 = ivar2 = 0
    read(ivar1, ivar2);
    print(ivar1, ivar2);
    const iconst1 = 1;
    const iconst2 = 2;
    ivar2 = ivar1 = +16 + 2 * -8;
    print(ivar1 < ivar2 && iconst1 <= iconst2,iconst1 * iconst2 >
     ivar2 - ivar1);
    ivar1 < read(ivar2) && iconst1 <= iconst2;
    ivar2 = print(ivar2) + 1;
  end + 1
bvar = begin
    bvar = false
    read(bvar);
    print(bvar);
    bvar = 12 / 5 * 5 + 12 % 5 == 12 && 6 >= 6;
    const bconst = true;
    print(!false && bvar == bconst || true != false);
  end && true;
cvar = begin
    cvar1 = 'c'
    read(cvar1);
    const cconst = 'c';
    cvar2 = 'z';
    print('a', cvar1 == cconst && cvar2 != 'b' || !true);
    'b';
  end;
```

```
print(ivar, bvar, cvar);
```

7 Conclusion

8 Appendices