UNIVERSITY OF TWENTE

Alia Programming Language

Fedor Beets
s1227874
Campuslaan 27

Joost van Doorn
s1095005
Dalsteindreef 2404, Diemen

July 5, 2014

Introduction

The Alia programming language was built for the subject compiler engineering at the University of Twente for the final project. 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 be vary from language to language. Over the first half of compiler engineering 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 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.

Chapter 1

Alia Programming Language

Chapter 2

Problems and solutions

Chapter 3

Syntax, context-constraints and semantics

Chapter 4

Translation rules

Chapter 5

Java-code

t 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.

Chapter 6

Tests

Chapter 7

Conclusion

Chapter 8

Appendices