Scala−−: an LLVM-targeted Scala compiler

Da Liu, UNI: dl2997

Contents

1 Background

Scala is heavily used in daily production among various industries. Being as a general purpose programming language, it is influenced by many ancestors including, Erlang, Haskell, Java, Lisp, OCaml, Scheme, and Smalltalk. Scala has many attractive features, such as cross-platform taking advantage of JVM; as well as with higher level abstraction agnostic to the developer providing immutability with persistent data structures, pattern matching, type inference, higher order functions, lazy evaluation and many other functional programming features. . Scala is truly good at breaking down complex large scale projects into manageable small solutions that work together in a functional fashion.

LLVM as a very powerful compiler infrastructure providing many advantages in compiler optimization and interfacing with many languages to ease the compiler backend writing magnificently.

2 Introduction

Scala−− is a prototype towards to be a full-fledged production-ready functional programming language with full support of current version of Scala. It will have fast startup time and potentially be able to leverage LLVM optimization/analyse. The prototype compiler will translate source code with a subset of Scala syntax to LLVM IR, The intermediate representation from frontend of the compiler will be implemented in OCaml, machine code from LLVM eventually running on all LLVM-supported architectures.

The "frontend" means the beginning part of the compiler, which consists of lexical analyzer, tokenizer, abstract syntax tree generator, semantic checker, the end product (intermediate representation) of this compiler frontend will be the input of the subsequent "backend" compiler, LLVM language binding will be used to create this part. So taking advantage of LLVM, the final output from this compiler will be low level machine code-generated executables; i.e., writing clean and concise, easy to maintain Scala code with functional programming features, while achieving performance approaching to assembly languages.

3 Project Design

- Compiler
  - LLVM-IR
  - Support x86 and ARM
  - Support a subset of Scala features[1]
- Benchmarking
  - Testing code
    - Basic algorithms written in Scala, or the targeting Scala-like language
    - Algorithm testing code counterpart in OCaml and/or other programming languages
  - Testing compiler For comparison purposes, to see the differences between the project and existing compilers and interpreters
    - target compiler (LLVM)
    - gcc/g++
    - Sun Java Development Kit (JDK)
    - Perl
    - Python
  - Benchmark utilities

4 Language Prototype Features

The LLVM-Scala−− *Prototype* will be a succinct, statically typed, functional programming, JVM-free language with relatively sophisticated type system.

The Prototype will share the same file suffix .scala as the Scala programming language. Only a small subset of Scala features will be supported in this Prototype, which were documented in this *Proposal*; all others that were not mentioned in this Proposal will not be supported or be supported in a very limited way. From the language processing mode point of view, only compilation will be supported in the Prototype; there will be no support for REPL or scripting. Object-oriented feature will be omitted. For instance, there will be no `trait`, or `class`. Another example was supporting `import` in a limited way, in that only *supported* library or Prototype-compatible code would be allowed to imported into a Prototype code and there is no definition of `package`. Further more, there will be no support for named arguments, variable number of arguments, companion objects, object comparison, nested functions, nested classes, Actors, file I/O, or exception handling, The undecidable supporting features will be documented in future report when necessary, for instance, the range difference between the Prototype and Scala for Int values.

4.1 Language Features

Typically, the supported features include the following:

- Comment: //, and /* */
- Semicolon
- `import`
- Literals
  - Boolean: `true`, and `false`
  - Int
  - Character
  - String
- List

[1] Details mentioned in the section of *Language Prototype Features*

- Tuple
- Mathematical operation
  - $+, -, *, /, \%$
  - Shorthand notations: $+=, -+, *=, /=, \%=$
  - Basic functions: `min, max, abs, ceil, floor, round, pow, log, log10, sqrt`
- Conditionals:
  - Conditional operators: $==, !=, <, <=, >, >=$
  - Logical operators: `&&`, $\|$, `!`
  - `if, else`
  - Loop: `for`-loop, `to`, `until`, `while`-loop, `do-while`-loop
- Pattern matching
- Input/output: string interpolation, `readLine`, `print`, `println`, `printf`
- Variable definition and type inference
- Method declaration and definition
- Higher order functions
- `class`, `extends`, `override`

4.2 Code Example

4.2.1 General Demonstration with Pattern Matching

The Prototype language will be a subset of the native Scala, hence the compatible code is essentially Scala with confined features.

Listing 1: Hello World!

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!");
  }
}
HelloWorld.main(args)
```

Listing 2: Functional programming paradigm

```
// Pattern matching, anonymous functions, partition, tuple decomposition
def quickSort(a: List[Double]): List[Double] = a match {
        case Nil        => Nil
        case x :: xs    =>
                val (lt, gt) = xs.partition(_ < x)
                quickSort(lt) ++ List(x) ++ quickSort(gt)
}
println(quickSort(List(2,3,1,3.1,3,3.1415926,14,3.1415)))
```

4.2.2 Conditionals

Listing 3: If-Else statement

```
if ((age >= 5) && (age <= 6)) {
        println("Go to Kindergarten")
} else if ((age > 6) && (age <= 7)) {
        println("Go to Grade 1")
} else {
        println("Go to Grade " + (age - 5))
}
```

4.2.3 Looping

The following looping example shows the basic supported feature in the Prototype language.

Listing 4: Looping

```
var i = 0;
while (i <= 5) {
        println(i)
        i += 1
}

do {
        println(i)
        i += 1
} while (i <= 9)

for (i <- 1 to 10) {
        println(i)
}
```

However, the `for`-looping will not support the following semicolon-separated conditional:

Listing 5: NO support

```
for (i <- 1 to 5; j <- 6 to 10) {
        println("i: " + i)
        println("j: " + j)
}
```

Listing 6: `until` keywords in looping

```
val randLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
for (i <- 0 until randLetters.length) {
        println(randLetters(i))
}
```

Listing 7: Iterate through a List

```
val aList = List(1,2,3,4,5)
for (i <- aList) {
        println("List iitem " + i)
}
```

4.2.4 Input and Output

Listing 8: STDOUT with string interpolation

```
val name = "Derek"
val age = 39
val weight = 175.5
println(s"Hello $name")
println(f"I am ${age + 1} and weight $weight%.2f")
```

The following `printf` example showed limited support of native Scala:

Listing 9: STDOUT with string interpolation

```
printf("%d %s %f %c", 1, "string", 3.1, 'c') // Supported
printf("'%5d'\n",5) // Right justify, NOT supported
printf("'%-5dHi'\n",5) // Left justify, NOT supported
printf("'%05d\n", 5) // Zero fill, NOT supported
printf("'%.5f'\n", 3.14) // Five decimal minimum & maximum, NOT supported
printf("'%-5s'\n", "Hi") // Left justify String, NOT supported
```

4.2.5 Strings

Listing 10: Available methods in String object

```
var randString = "I saw a dragon fly by"
println("String: " + randString)
println("3rd index value: " + randString(3))
println("String length: " + randString.length())
println("Concatenate: " + randString.concat(" and explode"))
println("Compare strings for equality; are strings equal " + "I saw a dragon".equals(ran
println("Get index of a match; dragon starts at index: ", randString.indexOf("dragon"))
println("Dragon at: " + twoDragonString.indexOf("dragon"))
// Convert a string into an array
val randArray = randString.toArray
for ( v <- randArray)
        println(v)
```

4.2.6 Functions

The function definition takes the following format:

Listing 11: Function definition

```
def funcName (param1: dataType, param2: dataType) : returnType = {
        function body
        return valueToReturn
}
// Give parameters default values
def getSum(num1: Int = 1, num2: Int = 2) : Int = {
        return num1 + num2
}
println("Default sum: " + getSum())
println("5 + 4 = " + getSum(5, 4))

// Recursion function
def factorial(num : BigInt) : BigInt = {
        if (num == 1)
        1
        else
        num * factorial(num - 1)
}
println("Factorial 20: " + factorial(20))
```

Named arguments and variable number of arguments are not supported.

Listing 12: NO support

```
println("5 + 4 = " + getSum(num2 = 5, num1 = 3))

def getSum2(args: Int*) : Int = {
        var sum : Int = 0
        for (num <- args) {
                sum += num
        }
        sum
}
println("getSum2: " + getSum2(1,2,3,4))
```

4.2.7 Tuples

Listing 13: Supported Tuple operations

```
// Tuples can hold values of many types, but they are immutable
var tupleMarge = (103, "Marge Simpson", 10.25)
printf("%s owes us $%.2f\n", tupleMarge._2, tupleMarge._3)
// Iterate through a tuple
tupleMarge.productIterator.foreach{i => println(i)}
// Convert Tuple to String
println(tupleMarge.toString())
```

4.2.8 Higher Order Functions

Listing 14: Example of supported higher order functions

```
// Functions can be passed like any other variable
// '_' is required after the function to state the meant function
val log10Func = log10 _
println("Log10 is: " + log10Func(1000))
// Apply a function to all items of a list with map
List(1000.0, 10000.0).map(log10Func).foreach(println)
// Use an anonymous function with map; receives an Int x and multiplies everyone by 50
List(1,2,3).map((x : Int) => x * 50).foreach(println) // i.e.,
List(1,2,3).map(_ * 50).foreach(println)
// Filter passes only those values that meet a condition
List(1,2,3,4).filter(_ % 2 == 0).foreach(println)
// Pass different functions to a function
def times3(num : Int) = num * 3
def times4(num : Int) = num * 4
// Define the function parameter type and return type
def multIt(func : (Int) => Double, num : Int) = {
        func(num)
}
printf("3 * 100 = %.1f\n", multIt(times3, 100))
printf("4 * 100 = %.1f\n", multIt(times4, 100))
```

Closure will not be supported. For instance, the following code will not work in the Prototype:

Listing 15: NO support for closure

```
val divisorVal = 5
val divisor5 = (num : Double) => num / divisorVal
println("5 / 5 = " + divisor5(5.0))
```

4.3 Language Reserved Words

`case` starts for matched expression

`class` starts a class declaration

`def` starts a method declaration

`do` starts a do-while loop

`else` starts else clause for an if clause in case the evaluation of if expression is false

`extends` indicates inheritance via parent class extension to derive child class(es)

`false` boolean value

`for` starts for loop

`if` starts if expression

`import` import members into current scope

`match` starts pattern matching expression

`new` creates a new instance of a class

`null` represents a null value

`object` instantiated class object

`override` indicates an update for a definition of original member of class

`return` indicates termination from a function call and pass the value on to call stack, if there is any

`super` refers an object's parent

`this` represents object itself

`to` used in loop comprehensions

`true` boolean value

`type` starts a type declaration

`val` represents read-only value variable

`var` represents modifiable variable

`until` used in loop comprehensions

`while` starts a while loop or while block of a do-while loop

`yield` generate results from a loop

_ represents function literal

`:` separator between identifier and type annotation

`=` assignment

`<-` generator expression in comprehensions

`=>` separator between argument list and function body in function literals

`;` optional separator between expression and statement

`,` separator between expression or literals

`!` logical negation

`&` logical AND

| logical OR

`&&` conditional AND

|| conditional OR

`==, <, <=, >, >=, !=` comparison

`(,),{,}` scope confinement symbols for expressions, statements, function blocks; or position access for List, Array etc

`//, /*, */` comments

5 Reference

5.1 Scala programming language

1. Martin Odersky, The Scala Language Specification, *Programming Methods Laboratory*, 2014
2. http://www.scala-lang.org/files/archive/spec/2.11/
3. http://docs.scala-lang.org/cheatsheets/

5.2 Scala programming language development

1. https://wiki.scala-lang.org/display/SIW/Compiler+Walk-Through
2. http://www.scala-lang.org/old/node/215.html
3. http://www.scala-lang.org/contribute/hacker-guide.html
4. https://github.com/lampepfl/dotty.git

5.3 Compile Scala to LLVM

1. http://vmkit.llvm.org/
2. https://github.com/scala-native/scala-native
3. https://github.com/greedy/scala
4. https://code.google.com/archive/p/slem/

5.4 Benchmarking

1. http://benchmarksgame.alioth.debian.org/