University of Tartu
Faculty of Mathematics and Computer Science
Institute of Computer Science

Martin Vels

Concurrent programming languages: Scala

Research paper for Distributed Systems Seminar

Advisor: Oleg Batrašev

Tartu 2011

Table of Contents

Introduction

The name Scala stands for "scalable language". This name comes from the idea that the language grows together with the demands of its user.
Scala runs on standard Java platform and interoperates with all Java libraries.
In Scala it is possible to use object-oriented as well as functional programming concepts.
In Scala every value is an object and every operation is a method call. For example, when you say 1 + 2 in Scala, you are invoking a method named "+" defined in class Int.
With functional programming in Scala you can benefit from the two main ideas of the functional programming. First of them is that functions as first-class values. This means that functions are values just like integer or string. It is possible to pass functions as arguments to other functions, also it is possible to return functions from other functions and define new functions inside functions. The second idea is that functions are not changing the data, but instead maps input values to output values. This means that immutable datatypes are used and it is not possible to change the input values, which leads to programs that have no side effects.
Scala compiler produces JVM byte code. Also, Scala programs can call Java class methods, access Java fields, inherit from Java classes and implement Java interfaces. Almost all Scala code uses Java libraries in a way that programmers are not even aware of it happening. Run-time performance of Scala byte code is comparable with native Java code. Scala also uses Java types, but makes them more convenient for a programmer, e.g. using implicit conversions, thus making it possible to convert from one type to another without programmer explicitly needing to do it himself.
It is also possible to invoke Scala code from Java code, but it can be a bit complex as Scala has some advanced features that Java lacks.
One interesting feature of Scala is that programs written in it are short. There are reports that program written in Scala can be ten times shorter that the same program written in Java. More realistic estimation would be that usually Scala programs are around half the size of Java program. Shorter programs mean that it takes less time to produce results as well as less bugs in code. Where all this comes from? Here is a short example how a simple class with a constructor looks both in Java and Scala:

```
// this is Java
class MyClass {
     private int index;
     private String name;

     public MyClass(int index, String name) {
          this.index = index;
```

```
            this.name = name;
      }
}

// this is Scala
class MyClass(index: Int, name: String)
```

As can be seen the code written in Scala is only one line and thus can be written more quickly, is less error prone and is more easily understood than Java code that achieves the same functionality.
Scala is statically typed, which means that it has following benefits like verifiable properties, safe refactorings and better documentation. At the same time it avoids verbosity that usually comes with safe typing using type inference and is flexible because of pattern matching.
There are a lot of different languages that Scala has borrowed its properties. Syntax of classes, packages and imports, as well as expressions, statements and blocks are from Java. Also basic types and class libraries come from Java. Smalltalk is where the uniform object model was borrowed. Universal nesting comes from Algol, Simula and Beta. Implicit parameters come from Haskell. Actors come from Erlang.
There are many companies who are using Scala: Linkedin, Twitter, Novell, Xerox, FourSquare, Sony, Siemens and many others. They have chosen Scala mainly because they have found that it helps them solve scalability issues that were not possible to solve with other languages. For example Twitter was using Ruby for their main message queue, but as the service usage grew, it was obvious that the Ruby-based solution was not able to handle the load. After rewriting the message queue in Scala they were able to handle the load and scale their back end.

Scala runs on most of modern operating systems and is available freely from the Scala website: http://www.scala-lang.org/
There are basically two ways to use Scala: first is to use Scala interpreter via interactive shell, second is to write Scala programs and compile them into Java byte code and run that code in Java Virtual Machine.

Variables

In Scala there are two different kind of variables: vals and vars. Val is similar to Java final variable, which means that once val has been initialized it can never be reassigned. A var on the other hand is similar to non-final variable in Java. Var can be reassigned at any time.

This is how you define val:

```
val msg = "Hello, world!"
```

and this is how you define var:

```
var msg = "Hello, world!"
```

As can be seen, the syntax is exactly the same, the difference comes after the

variable definition. In case of val it is not possible to assign new value to the variable and compiler will give you an error: reassignment to val message.

Functions

Here is the basic syntax on how to define a function in Scala:

```
def max(x: Int, y: Int): Int = {
     if (x > y) x
     else y
}
```

As can be seen the function definition starts with the keyword "def". Function name follows after the "def". Function parameters are inside the parentheses. Each parameter name is followed by the colon and the type. After the parentheses there is another colon followed by the type. This type is the type of the result value of the function. After the result type equals sign and curly braces start. The body of the function is in the curly braces. If the function only has one line in its body, it is possible to leave the curly braces out of the function definition. As can be seen, there is no explicit return statement in the function body. This means that Scala uses one of the if-statement results as the function result. However, it is possible to use explicit "return" keyword in functions to make your code more understandable.
There are couple of special cases in function definition, that are important to know: first if there are no parameters for the function then empty parentheses should be present after function name. Second, if the function does not return anything, then you do not have to specify the return type. In that case Scala will use on special type, called Unit as return type. Unit-type is similar to Java's void type.

if ... else

If-statement in Scala is very similar to if-statements in many other languages. All it does is testing a condition and executes a code that is in one of the two branches depending on the result of the expression. So a simple if-condition would look like this:

```
val filename =
     if (!args.isEmpty)
          args(0)
     else
          "default.txt"
```

Loops

There are several loops available in Scala: while, foreach and for. They behave very similarly to their Java counterparts.

while

While loops looks like this:

```
var i = 0
while (i < args.length) {
     println(args(i))
     i += 1
}
```

This short code snippet is iterating over the command line arguments and printing every one of them on a new line.
As can be seen, the code does not use `++i` nor `i++` as it is possible in several other languages. Instead you should use `i = i + 1` or `i += 1` instead.

There is also do … while loop available:

```
var line = ""
do {
     line = readLine()
     println("Read: " + line)
} while (line != "")
```

foreach

Instead of imperative style while-loops as seen above, it is possible to use more functional style in Scala. So the command line printing functionality could be written in functional style like this:

```
args.foreach(arg => println(arg))
```

Here the foreach method of args is called and another function println is passed in. To be more precise the function literal that takes one parameter named arg is passed into method.

for

There is also a possibility to use for-loops, so our example of looping through command line arguments could look like this:

```
for (arg <- args)
     println(arg)
```

Here args is an array and arg is a val. Now the for iterates over all the args array and creates a new val in every pass, which is used by println function.

You could also use for to iterate over some range of numbers:

```
for (i <- 1 to 4)
     println("Iteration: " + i)
```

Classes and Objects

It is very easy to define classes:

```
class ChecksumAccumulator {
     private var sum = 0

     def add(b: Byte): Unit = {
          sum += b
     }

     def checksum(): Int = {
          return ~(sum & 0xFF) + 1
     }
}
```

First there is a keyword "class" that is followed by the class name. Class body is inside curly braces. Class variables are following as well as method definitions. As can be seen keyword "private" was used to define var sum to be private. If the keyword is omitted, the variable will be public. It is possible to define methods to not have curly braces just like was mentioned above when we talked about functions.

To init a class instance just use new operator as this:

```
val acc = new ChecksumAccumulator
```

As Scala cannot have static members like Java, there are singleton objects in Scala that are defined like this:

```
import scala.collection.mutable.Map

object ChecksumAccumulator {
     private val cache = Map[String, Int]()

     def calculate(s: String): Int =
          if (cache.contains(s))
               cache(s)
          else {
               val acc = new ChecksumAccumulator
               for (c <- s)
                    acc.add(c.toByte)
               val cs = acc.checksum()
               cache += (s -> cs)
               cs
          }
}
```

As can be seen this example singleton object has the same name as the class that was defined above. This object is called class's companion object. And class is called companion class of the singleton object. Both singleton object and the class can access each other's private members.

Now it is possible to invoke the calculate method in ChecksumAccumulator singleton object like this: `ChecksumAccumulator.calculate("Every value is an object.")`

One difference between singleton objects and classes is that singleton object can not take parameters. As it is not possible to instantiate a singleton object with new keyword there is no way to pass parameters as well. There a many ways to use singleton objects, one of them is defining an entry point to a Scala application.

This leads us to an example of how a simple Scala application would look like:

```
import ChecksumAccumulator.calculate

object Summer {
     def main(args: Array[String]) {
          for (arg <- args)
               println(arg + ": " + calculate(arg))
     }
}
```

There are abstract classes available in Scala, these are classes that declare methods with no implementations and that can not be instantiated:

```
abstract class Element {
     def contents: Array[String]
}
```

To be able to instantiate new objects, we need to have concrete classes. To create a concrete class from abstract class we use extends keyword:

```
class ArrayElement(conts: Array[String]) extends Element {
     def contents: Array[String] = conts
}
```

Using extends keyword we are creating inheritance bind between abstract class and its concrete child class.

Now if we would like to extend our concrete class further we can again use extend keyword to define a new class. But to be able to override existing methods in our concrete parent class we need to use "override" modifier in front of the method definition:

```
class Cat {
     val dangerous = false
}

class Tiger(param1: Boolean, param2: Int) extend Cat {
     override val dangerous = param1
     private var age = param2
}
```

If parent class wants to make sure that a member is not overridden by subclass, it is possible to define member as final:

```
class ArrayElement extends Element {
     final override def demo() {
          println("ArrayElement's implementation invoked")
     }
}
```

Now if we would like to override this method in subclass of ArrayElement we would get a compiler error which says that method demo cannot override final member.

Data types and literals

The basic data types in Scala are very similar to types in Java and these types are actually based on the actual Java types. Here are some of the types: Byte, Short, Int, Long, Char, String, Float, Double, Boolean.
Literals in Scala are also very similar to Java, e.g. integer literals are written like this: "val decimal = 10", floating point literals: "val big = 1.234", character literals: "val a = 'A'", string literals: "val hello = "hello", boolean literals: "val bool = true".

Operators

As was mentioned before, every value in Scala is an object and all the operations that can be performed on these values/objects are actually methods defined in according classes. So for example if you are doing addition like this: "val sum = 1 + 2", Scala actually invokes "(1).+(2)", which means that "+" is a method in Integer class. It is also interesting that in Scala every method can be an operator, which means that it is possible to call methods like operators:

```
val s = "Hello, world!"
s indexOf 'o'
```

Here you can see that method "indexOf" is called on object s (we can write the same statement like this: "s.indexOf('o')").
Same applies to all the arithmetic operators, comparison operators, bitwise operators, equality operators etc. All operators are methods.

To define an operator in class all you need to do is define a method in class. The name of the method can be used as an operator. So for example we could define an addition operator like this:

```
def + (that: Rational): Rational =
     new Rational(
          numer * that.denom + that.numer * denom,
          denom * that.denom
     )
```

Constructors

In Scala there are no constructors existing in the same way as they are in Java for example. Instead when class is defined, everything that is written in the class body that is not in separate methods is effectively interpreted as the constructor body. There is also possibility to create auxiliary constructors. These are needed if you want to create a class instance with different parameters. Auxiliary constructors would look like this:

```
def this(n: Int) = this(n, 1)
```

here the method "this" is an auxiliary constructor which calls primary constructor of the class. This means that there should be only one entering point to any of the classes and auxiliary constructors are only shortcuts to primary constructor with different parameters.

Exception handling

Exception handling in Scala works very similarly as in many other languages. To throw an exception you need to create an exception object and then use keyword "throw" to throw it: `throw new IllegalArgumantException`
To catch an exception, try … catch is used:

```
try {
     val f = new FileReader("input.txt")
} catch {
     case ex: FileNotFoundException => // Handle missing file
     case ex: IOException => // Handle other I/O error
}
```

There is also finally clause available in Scala, which can be used if there is a need to execute some code no matter how the expression terminates. For example, if you need to close all the open files:

```
val file = new FileReader("input.txt")
try {
     // Use the file
} finally {
```

```
    file.close() // Be sure to close the file
}
```

Match expressions

In Scala there is a match expression that acts like switch-statement in Java and in other languages. Here is an example how to create match expression in Scala:

```
val firstArg = if (args.length > 0) args(0) else ""

firstArg match {
    case "salt" => println("pepper")
    case "chips" => println("salsa")
    case "eggs" => println("bacon")
    case _ => println("huh?")
}
```

Here different cases are checked and some text is output, based on the conditions. Underscore "_" is used as default-case. Also it can be seen that no "break" keyword is used as in Scala the break is implicit and there is no fall through available.

Closures

Closure is the function value that is created at runtime from this function literal. For example in following example:

```
var more = 1
val addMore = (x: Int) => x + more
addMore(10) // results is 11
```

the function value contains variable more, that is closed into the function. In Scala the captured variable in closure can see the changes of the variable. So if we extend our previous example like this:

```
more = 9999
addMore(10) // results in 10009
```

we can see that the change of more variable will also be seen in closure.

Named parameters and default parameter values

In Scala it is possible to specify the function parameters by name, thus making it possible to not follow the exact order of the parameters. So, it is possible to define a function:

```
def speed(distance: Float, time: Float): Float =
     distance / time
```

and call this function like this: `speed(time = 10, distance = 100)`
As can be seen, the order of the parameters is reversed, but using the parameter names, it is still possible to invoke the function correctly.

It is also possible to specify default values for parameters, so if the parameter has no value set during the function call, there will be default value set instead:

```
def printTime(out: java.io.PrintStream = Console.out) =
     out.println("time = " + System.currentTimeMillis())
```

Now it is possible to call this function like this: `printTime()`, which results in out being set to `Console.out`, or like this: `printTime(Consoler.err)`, which sets the out parameter to `Console.err` instead.

Traits

For programmers familiar with Java, traits can be thought of as interfaces which can have concrete members. Traits can also declare fields and maintain state. In fact, it is possible to do anything in a trait definition that can be done in class definition. There are however two important exceptions: first, a trait can not have class parameters, second, super calls are dynamically bound in traits.

Trait definition is very similar to class definition:

```
trait Philosophical {
     def philosophize() {
          println("I consume memory, therefore I am!")
     }
}
```

Using traits in classes can be done in two different ways, using `extend`:

```
class Frog extends Philosophical {
     override def toString = "green"
}
```

or `with` keywords:

```
class Animal

class Frog extends Animal with Philosophical {
     override def toString = "green"
}
```

It is possible to mix in as many traits as you like, just adding new `with TraitName` one after another in class declaration.

There are some rules that can help Scala-programmer to decide if traits should be used:

If the behavior will not be reused, then make it a concrete class.
If it might be reused in multiple, unrelated classes, make it a trait.
If you want to inherit from it in Java code, use an abstract class.
If you plan to distribute it in compiled form, use abstract class.
If efficiency is very important, use a class.
If you are still not sure, after considering all of above, start by making it a trait as it can always be modified later. Using traits keep options open.

Packages and imports

Scala supports two different ways on how to divide you classes into packages. First one is similar to Java packages, where you just add `package this.is.packagename` in the beginning of the source file and second way is more similar to C# namespaces where you nest packages inside each other like this:

```
package launch {
     class Booster3
}

package bobsrockets {
     package navigation {
          package launch {
               class Booster1
          }
          class MissionControl {
               val booster1 = new launch.Booster1
               val booster2 = new bobsrockets.launch.Booster2
               val booster3 = new _root_.launch.Booster3
          }
     }
     package launch {
          class Booster2
     }
}
```

As you can see, there is one special package name _root_ used to access the class Booster3 from the launch package. This special package is used to address the top level package.

To import packages, keyword `import` is used. And it works almost like in Java

imports with the difference that instead of asterisk (*) used in Java, Scala uses underscore (_) instead:

```
import bobsdelights._
```

There are three main differences in Scala imports however compared to Java:

- imports may appear anywhere
- imports may refer to objects in addition to packages
- imports let you rename and hide some of the imported members

Case classes and pattern matching

Case class

When adding a "case" keyword in front of a class definition, this class is called case class. For every case class, Scala compiler automatically adds following:

- factory method with the name of the class, which allows you to write val v = Var("x") instead of val v = new Var("x"), which in turn allows you to nest classes and not have confusing "new" keyword in the code.
- "val"-field for every parameter of a case class
- additional methods "toString", "hashCode" and "equals"
- "copy"-method, so it is possible to create modified copies of a class

The biggest advantage of case classes is that they support pattern matching.

Pattern matching

Pattern matching is Scala is achieved with following construct:

```
selector match { alternatives },
```

where selector is is the expression that we are trying to match and alternatives are a sequence that start with case-keyword. Each alternative includes a pattern and one or more expressions, which will be evaluated if the pattern matches. An arrow symbol => separates the pattern from the expressions. `match` is similar to `switch` in other languages, but there are three differences:

- match always results in a value
- match never "falls through" like in switch without break
- if none of the patterns match, an exception MatchError is throw, which means that all the cases have to be covered.

There are several kinds of patterns:

- wildcard patterns: this is expressed with underscore (_) and it matches any object whatsoever
- constant patterns: any literal may be used as constant, e.g. 5, true, "hello" etc. and it only matches itself

- variable pattern: just like wildcard pattern, it can match any object, Scala also binds the variable to whatever the object is, you can use that variable afterwards in your expression or not.
- constructor patterns: the most powerful feature of pattern matching, where case classes are used, which helps to check if the given object is a member of a class and if constructor parameters match certain pattern. This is called deep matching, when not only top-level object is checked but also the contents of the object is checked.
- sequence patterns: it is also possible to match against sequence types like List or Array
- tuple patterns: also tuples can be used in pattern matching
- typed patterns: convenient replacement for type tests and type casts

As was mentioned above, every possible case has to be matched in pattern matching. This means that if case classes are used for example there is a possibility that someone defines a new case class that inherits from an abstract parent class. This could break the match-statement. To avoid such a problem it is possible to make superclass sealed:

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case UnOp(operator: String, arg: Expr) extends Expr


def describe(e: Expr): String = e match {
     case Number(_) => "a number"
     case Var(_) => "a variable"
}
```

Now as can be seen, UnOp is not covered in match, which results in compiler warning about missing combination. This helps to reduce the bugs in the code.

Actors

If there is a need to create a program where things have to happen independently, the usual way to achieve this is by using threads. However, in Java and several other languages, it is quite hard to get threads working as expected. There are many issues that need to be addressed, like synchronization of the threads, possible dead lock situations and race conditions. It is not that simple for a programmer to get all this right and it is very difficult to find bugs in code that is using threads.

Scala introduces an alternative to threads, called actors. Actors are addressing the fundamental problem of threads by providing share-nothing, message-passing model that is more easy to reason about. An actor is a thread-like entity that has a mailbox for receiving messages.

To implement an actor, scala.actors.Actor subclass needs to be subclassed and act-method has to be implemented:

```
import scala.actors._
object MessageActor extends Actor {
     def act() {
          loop {
               receive {
                    case msg => println("received message: " + msg)
               }
          }
     }
}
```

To start an actor, you need to call start method of the actor: `MessageActor.start()`

To send a message to an actor, "!" method is used:

```
MessageActor ! "hi, there"
```

As can be seen, the MessageActor has a loop which receives messages from the actor's message box by calling a receive method, passing in a partial function and handles it by simply printing it to screen. Actor will only process messages that are matching one of the cases in the partial function passed to receive.

Actors are implemented on top of normal Java threads. Every actor has its own thread so all the act methods get their turn. However, in reality Java threads are quite expensive in the sense of memory consumption as well as CPU cycles. To make programs more efficient, there is a possibility to optimize how actors are working with the messages. Instead of receive it is possible to use react method. As react will not return like receive, it is possible for the library to reuse the current thread for the next actor that wakes up. This way for every actor using react, only single thread is needed to host all these actors.

Remote Actors

Scala actors can also communicate over the network, not only inside the same JVM. Remote actors differ from the regular Actors by only two things: remote actor itself has to specify what is the port it will be listening and what is the name of the actor listening that port, as there could be many actors listening the same port. So the act()-function would look like this:

```
def act() {
     alive(9000)
```

```
    register('servicename, self)
    loop {
        .... // receive or react just like regulara actors
    }
}
```

The client who will be using remote actor has to use select-method to subscribe to remote actor like this:

```
val client = select(Node("123.123.123.123", 9000), 'servicename)
```

Here the Node is a case class that takes two parameters, first the ip-address of the server and second the port number. Also the 'servicename symbol is used in select-method to identify the exact service.

That is all there is to use remote actors. Now all the message passing works just like with regular actors:

```
client ! "this is a message"
```

Experiment with remore actors

As an experiment I created a simple application that consisted of two remote actors, both running on separate machines. Virtual machines with Ubuntu Linux 11.10 were used with Scala 2.9.1 installed. First machine acted as a Computational server that was receiving integers from the client and checked if these numbers were prime numbers. The result was sent from the computational serve to result server which had another remote actor listening.

Here is the source code for the Prime case class (Prime.scala):

```
case class Prime(number: Int, isPrime: Boolean)
```

Here is the source code of the ComputeServer (ComputeServer.scala):

```
import scala.actors.Actor
import Actor._
import scala.actors.remote.RemoteActor.{alive, register, select}
import scala.actors.remote.Node

class ComputeServer extends Actor {
        val resultClient = select(Node("172.16.53.149", 9001), 
'resultserver)

    def isPrime(n: Int): Boolean = {
```

```
        (2 until n) forall (n % _ != 0)
    }

    def computeAndSendAnswer(x: Int) {
        if (isPrime(x)) {
            val answer = new Prime(x, isPrime(x))
            resultClient ! answer
        }
    }

    def act() {
        alive(9000)
        register('computeserver, self)
        loop {
            react {
                case x: Int  => {
                     val reponse = "Got integer: " + x + " starting
calculations"
                    println(reponse)
                    reply(reponse)
                    computeAndSendAnswer(x)
                }
                case msg => println("unknown message: " + msg)
            }
        }
    }
}

object ComputeServer extends App {
    print("Starting ComputeServer ...")
    val server = new ComputeServer
    server.start()
    println("done")
}
```

Here is the source code of the result server (ResultServer.scala):

```
import scala.actors.Actor
import Actor._
import scala.actors.remote.RemoteActor.{alive, register, select}
import scala.actors.remote.Node

class ResultServer extends Actor {
    scala.actors.remote.RemoteActor.classLoader =
getClass().getClassLoader()
    def act() {
        alive(9001)
        register('resultserver, self)
        loop {
            react {
                case Prime(number, isPrime)  => {
                    val answer = "Got new prime number: " + number
                    println(answer)
                }
                case msg => println("unknown message: " + msg)
            }
        }
    }
}

object ResultServer extends App {
    print("Starting ResultServer ...")
    val server = new ResultServer
    server.start()
    println("done")
}
```

Here is the client that sends prime numbers to ComputeServer (PrimeStarter.scala):

```
import scala.actors.remote.RemoteActor.{select}
import scala.actors.remote.Node

object PrimeStarter extends App {
    //scala.actors.Debug.level = 3
```

```
    val computeServer = select(Node("172.16.53.148", 9000),
'computeserver)
    for (i <- 90000000 to 90000099) {
        val ret = computeServer ! i
        println("Sent to ComputeServer: " + i)
    }
    println("done")
}
```

Bibliography

- Martin Odersky, Lex Spoon, Bill Venners. Programming in Scala. Second Edition. Artima Press, 2010.
- Philipp Haller and Frank Sommers. Actors in Scala. Artima Press, 2011.