Modern Functional Programming and Actors With Scala and Akka

Aaron Kosmatin

Computer Science Department
San Jose State University
San Jose, CA 95192
707-508-9143
akosmatin@gmail.com

Abstract

For many years, functional programming has had an academic presence in computer science, but has been poorly represented in popular languages. Functional programming has been making a resurgence lately. With Java 8, Oracle has introduced lambda functions, an essential part of functional programming, to Java. Much of the resurgent interest in functional programming originates with Martin Odersky, designer of Scala and Co-founder of Typesafe.

The Actor Model of concurrent programming pre-dates Scala, with implementations in languages such as Erlang, but has been an underutilized model. The Actor Model is well adapted to multi-core and multi-machine programming. The Actor Model has regained interest recently due to the work of Jonas Bonér, developer of the Akka Actor library for Java and Scala and Co-founder of Typesafe.

Scala and Akka actors provide developers with well developed tools for parallel and distributed computing. Due to the tools available with the Typesafe platform, Scala and Actors have become a core part of the technology stack at companies like Twitter, LinkedIn, and Netflix.

1 Functional Programming

Functional programming has not seen wide use in the programming industry. There are several reasons for this. Functional programming languages place a strong emphasis on minimization of side effects, and immutable states. Functional programming is an expansion of lambda calculus. The idea of functional programming is that the program executes as a series of mathematical function evaluations[2]. From its mathematical nature, every function call that has the same arguments should produce the same return statement.

The emphasis on reduction of side effects has hurt the adoption of functional languages in the industry, many programs require some form of side effect. Side effects include any modification of a state, such as modifying a variable, printing to the console and writing to files. As well, for many programmers, it is easier to conceptualize problems by modifying states.

While there are drawbacks to functional programming, there are two important advantages. First, functional code tends to be easier to read and maintain between programmers. By its nature, functional programming has few mutable variables. It is not required for a programmer to track how a variable is changing through the execution of the code. Second, with its emphasis on immutability, functional programming tends to be much more parallelizable.

```scala
1  object VarValDeclarations {
2    var a = 1
3    val b = 2
4    val c:Int=3
5    def aFunction(value:Int):Int = {
6      value
7    }
8    def bFunction(value:Any) = {
9      value
10   }
11 }
```

Figure 1: Variable and Value declarations

```scala
object firstClassFunctions {
  def f(x: Int) = x + 2;
  println(Array(1,2,3).map(f))
}
```

Output:

```
Array(3, 4, 5)
```

Figure 2: First class functions in Scala

2 Scala

Traditionally, Object oriented programming has been a subset of procedural languages, with languages like C++ and Java. In 2003, Martin Odersky developed Scala for his programming language class at École Polytechnique Fédérale de Lausanne[1]. He wanted to show that object oriented programming was compatible with functional programming.

Strictly speaking, Scala is not a functional programming language, nor is it a procedural language. Scala fully supports both forms of programming. Scala is built on top of the JVM, and compiles to Java bytecode. As such, Scala is compatible with Java libraries, and Scala libraries are compatible with Java. Scala addresses one of the limiting factors of previous functional languages, in that it allows the programmer to use procedural programming when required.

2.1 Scala Language Basics

Scala has a philosophy of offloading much of the typing to the compiler. Like Java, Scala is a strongly typed language, but the type is usually inferred by the compiler at compile time. Lines 3 and 4 of figure 1 show the declaration of two integers. On line 3, the type is inferred by the compiler, and on line 4, the type is explicitly set by the programmer.

Scala also supports variables and values. Variables work as expected from other languages. Values are the immutable variant of variables. Scala, in general, prefers immutable objects. An example of Scala's preference for immutability is in the Scala collections. By default the members of a collection are immutable, unless a mutable collection is imported by the programmer.

A last thing to notice in figure 1 is that the functions do not have an explicit return statement. Scala does support the return keyword, but will default to returning the last line of a function block. This makes sense given Scala's functional influence. Function's with no return, or void functions, are used to create side effects, but since functional languages try to eliminate side effects, every function should have a return value.

Figure 2 shows how functions can be used as first class objects in Scala. This is a standard feature of functional programming, where a functions can be used in the same manner as other types. In this example, the function f is used as an argument to the map function[3].

As can be seen from figure 3, Scala and Java are similar. One important difference is that unlike Java, Scala supports the Singleton Design Pattern at the language level. In Scala, an object is a singleton. Since Scala is object oriented, it also supports classes and an object hierarchy like Java's.

2.2 For Yield Condition/Map Filter

Scala supports functional constructs, such as maps. In figures 4, 5 and 6 show examples of Scala's for-yield syntax. In figure 4, maps are used to generate a cartesian product. In figure 5, a for-yield statement is used. At compilation, the code in figure 5 is converted into code similar to figure 4, however, depending on the situation, one may be more readable than the other[4]. In figure 6, a condition is added to the for statement, removing symmetric results. A similar condition can be applied to the map syntax.

Listing 1: Java Hello World

```java
public class HelloJava {
  public static void main(String[]
    args){
    System.out.println("Hello Java");
  }
}
```

Listing 2: Scala Hello World

```scala
object HelloScala {
  def main(args: Array[String]){
    println("Hello Scala")
  }
}
```

Figure 3: Hello Worlds in Scala and Java

Listing 3: Sequential Loop Execution

```scala
object ParallelExample extends App{
  val integers=1 until 5
  val timesTable=for(integer1 <-
    integers) yield {
    for(integer2 <- integers) yield {
      integer1*integer2
    }
  }
  timesTable.foreach(println(_))
}
```

Output:

```
Vector(1, 2, 3, 4)
Vector(2, 4, 6, 8)
Vector(3, 6, 9, 12)
Vector(4, 8, 12, 16)
```

Listing 4: Parallel Loop Execution

```scala
object ParallelExample extends App{
  val integers=1 until 5
  val timesTable=for(integer1 <-
    integers.par) yield {
    for(integer2 <- integers) yield {
      integer1*integer2
    }
  }
  timesTable.foreach(println(_))
}
```

Output:

```
Vector(1, 2, 3, 4)
Vector(2, 4, 6, 8)
Vector(4, 8, 12, 16)
Vector(3, 6, 9, 12)
```

Figure 7: Sequential and parallel execution of loop statements.

```scala
object CartesianProduct extends App {
  val points = 0 until 5
  val cartesianProduct = points.flatMap(x
    => points.map(y => (x, y)))
  println(cartesianProduct)
}
```

Output:

```
Vector((0,0), (0,1)...(6,6))
```

Figure 4: Map Syntax

```scala
object CartesianProduct extends App {
  val points = 0 until 5
  val cartesianProduct=for{x<-points
            y<-points} yield {
      (x,y)
  }
  println(cartesianProduct)
}
```

Output:

```
Vector((0,0), (0,1)...(6,6))
```

Figure 5: For Yield Syntax

```scala
object ParallelExample extends App {
  val integers = 1 until 5
  val timesTable = for (integer1 <-
    integers) yield {
    for {integer2 <- integers
         if (integer2 <= integer1)
    } yield {
      integer1 * integer2
    }
  }
  timesTable.foreach(println(_))
}
```

Output:

```
Vector(1)
Vector(2, 4)
Vector(3, 6, 9)
Vector(4, 8, 12, 16)
```

Figure 6: Conditional For Yield

Listing 5: Parallel Loop Execution

```scala
import scala.collection.parallel.ParSeq

object ParallelExample extends App{
  val integers=ParSeq(1,2,3,4)
  val timesTable=for(integer1 <- integers)
    yield {
    for(integer2 <- integers) yield {
      integer1*integer2
    }
  }
  timesTable.foreach(println(_))
}
```

Output:

```
ParArray(1, 2, 3, 4)
ParArray(4, 8, 12, 16)
ParArray(3, 6, 9, 12)
ParArray(2, 4, 6, 8)
```

Figure 8: Parallel Collections

2.3 Parallel Execution of Collections

The first example of parallel computation methods that exist in Scala, but not Java, can be seen in figure 7. In this case, the .par operator is added on line 3. The code on the left executes sequentially, the code on the right is executed in parallel. By adding the .par operator, Scala knows to treat that collection as parallel. While executing, it will determine how many threads are available, create the threads, parse parts of the collection to each thread and then join the results. It should be noted that the resulting collection timesTable is no longer deterministic and may change between different executions of the code. This is why the third and fourth lines of the output have reversed order from the original output. This parallelization of mappings can be done in other languages, but its execution is much simpler in Scala.

In fact, Scala supports a large range of collection implementations. Some common ones are Seq, which is similar to an Array in Java, List, which is similar to an ArrayList in Java, and Stream, a collection type that was added to Java 8 in which the individual members have lazy evaluation. For most collections, Scala also supports a parallel version. In figure 8, the code has been changed to use a parallel collection instead of a sequential collection. Similar to the previous parallel example, the order of rows is not deterministic. In this example, the order of the columns has been preserved, but this can not be relied on. Parallel collections provide a simple, powerful data type for processing large collections where the order of the collection is not important.

2.4 Pattern Matching

Another important feature of Scala is pattern matching. Pattern matching is used in several ways in Scala. In figure 9, several examples of pattern matching can be seen. The first example is pattern matching on value, the second is pattern matching on type, and the third is using the Option data type. Option is a wrapper for optional values. It is used for passing optional data. Each example of pattern matching can be further developed to provide more complex behaviours.

2.5 Concurrent Execution of Futures

Futures are another method of parallel execution in Scala. Futures generate a new thread, where the future is evaluated. While the future is evaluating, the main thread of execution is still processing. When the future completes, it return to the

```
object PatternMatching extends App {
  def matchValue(x: Int): String = x match
    {
    case 1 => "one"
    case 2 => "two"
    case _ => "many"
  }
  println(matchValue(3))

  def matchType(x: Any): String = x match {
    case _:Int => "Int"
    case _:String => "String"
    case _ => "Something else"
  }
  println(matchType(3.5))

  def matchOption(x: Option[String]):
    String = x match {
    case Some(str) => str
    case None => "No String"
  }
  println(matchOption(Some("A string")))
}
```

Output:

```
many
Something else
A String
```

Figure 9: Pattern Matching

```
1  import scala.concurrent._
2  import scala.concurrent.duration._
3  import scala.concurrent.ExecutionContext
4          .Implicits.global
5  import scala.util.{Failure, Success}
6
7  object FuturesExample extends App{
8    val a:Future[String] = future {
9      Thread sleep 1000
10     "A String assignment after sleep"
11   }
12
13   a onComplete {
14     case Success(str) => println(str)
15     case Failure(t) =>
       println(t.getMessage)
16   }
17
18   println("After declaration")
19   Await.result(a, 30 second)
20 }
```

Output:

```
After declaration
A String assignment after sleep
```

Figure 10: Futures in Scala

main thread. In figure 10, a future is declared on line 8. This future will evaluate in its own thread. Starting on line 13, pattern matching is used to let the future know what to do when it completes. In the output, it can be seen that the println on line 18 is executed before the println on line 9. This is because both threads are executing at the same time, but the future has a sleep statement which delays its println.

Futures are a method of concurrent execution that will be familiar to AJAX programmers. Futures allow a second thread to be generated so execution of the first thread is not blocked during the evaluation.

3 Akka Actors

The Akka library was originally developed by Jonas Bonér. The core Scala library also includes an actor implementation, but the Akka library is considered to be the more robust of the two implementations. The actor model has been used in other functional languages. Most notably for Scala is the Erlang implementation of actors. Jonas Bonér credits this implementation as being his guide for the Akka library.

Actors are a method of message passing between threads. A main thread of execution can define the actors, and pass messages to the actors, as well as let actors pass messages directly between each other. The big advantage of actors is that the actors are agnostic to which machine they are on. The implementation of actors is similar if they are running as separate threads on the same machine, or threads on different machines. Actors allow a method of developing large, scalable applications across multiple machines.

Figure 11 shows a basic hello world with an actor[5]. The actor is defined on lines 2-7, and the main thread starts on line 8. In this example, a single actor is created. Pattern matching is used to match the messages sent to the actor and different println's are executed depending on the message.

```
1  import akka.actor.{Actor, Props,
       ActorSystem}
2  class HelloActor extends Actor {
3    def receive = {
4      case "hello" => println("hello back at
       you")
5      case _ => println("huh?")
6    }
7  }
8  object Main extends App {
9    val system = ActorSystem("HelloSystem")
10   val helloActor =
       system.actorOf(Props[HelloActor], name
       = "helloactor")
11   helloActor ! "hello"
12   helloActor ! "buenos dias"
13 }
```

```
hello back at you
huh?
```

Figure 11: An actor hello world

On lines 11 and 12, two messages are sent to the actor, and in the output, the response of the actor to those messages can be seen.

The actor in figure 11 can be redefined to the actor in figure 12. During execution, the same two messages are sent to this actor, but now the actor will still be processing the first message when it receives the second. As can be seen from the output, the order of the println's executed by the new actor is the same as figure 11. The actor receives the second message while it is processing the first message. The message will wait on the actors message queue while the actor is completing execution of the first message. Only once the first message has been completed does the actor start processing the second message.

In figure 13, the main method is changed. Two copies of the actor are instantiated and a router is used to send messages. Akka provides a several different pre-built routers, as well as allowing the programmer to define their own routers[6]. In this example, a pre-built round robin router is being used. The router will direct messages between the two actors, and each actor will execute in its own thread. As can be seen from the output, the second message completes execution before the first message.

```
class HelloActor extends Actor {
  def receive = {
    case "hello" => {
      Thread sleep 2000
      println("hello back at you")
    }
    case _ => {
      Thread sleep 1000
      println("huh?")
    }
  }
}
```

```
hello back at you
huh?
```

Figure 12: An actor whose responses take different execution times.

```
object Main extends App {
  val system = ActorSystem("ha")
  val ha1 =
    system.actorOf(Props[HelloActor], name
    = "ha1")
  val ha2 =
    system.actorOf(Props[HelloActor], name
    = "ha2")
  val routerProps =
    Props.empty.withRouter(RoundRobinRouter(
      routees = Vector(ha1,ha2)))
  val router = system.actorOf(routerProps)
  router ! "hello"
  router ! "buenos dias"
}
```

```
huh?
hello back at you
```

Figure 13: Creating a pool of actors

Actors provide a flexible way of creating scalable concurrent applications. Both Scala and Akka provide implementations of the actor model. Starting with Scala 2.10, the Scala actor model was deprecated[7]. Typesafe is continuing to make Akka the default actor model.

4 Conclusion

Scala provides a number of additional constructs for parallel programming. The basic constructs of Semaphores and Mutex's exist, Scala inherits these from Java. However, we program in high level languages to speed up development time and increase code readability. The additional parallel programming methods, parallel collections, futures, and actors, help programmers quickly write parallel code and increase the readability of that code.

References

[1] *"Scala Programming Language."* Wikipedia. Wikimedia Foundation, 16 Nov. 2014. Web. 17 Nov. 2014. <http://en.wikipedia.org/wiki/Scala_(programming_language) >.

[2] *"Functional Programming."* Wikipedia. Wikimedia Foundation, 16 Nov. 2014. Web. 17 Nov. 2014. <http://en.wikipedia.org/wiki/Functional_programming>.

[3] *""functions Are First Class Values" What Does This Exactly Mean?"* Scala. Web. 17 Nov. 2014. <http://stackoverflow.com/questions/10777333/functions-are-first-class-values-what- does-this-exactly-mean>.

[4] *"How Does Yield Work?"* Faq. Web. 17 Nov. 2014. <http://docs.scala-lang.org/tutorials/FAQ/yield.html>.

[5] *"Simple Scala Akka Actor Examples (Hello, World Examples)."* Scala, Java, Linux, Mac Os X, Iphone, Perl, Drupal, Tutorials. Web. 17 Nov. 2014. <http://alvinalexander.com/scala/simple-scala-akka-actor-examples-hello-world-actors>.

[6] *"Routing."* Akka Documentation. Web. 17 Nov. 2014. <http://doc.akka.io/docs/akka/snapshot/scala/routing.html>.

[7] *"The Scala Actors Migration Guide."* - Scala Documentation. Web. 17 Nov. 2014. <http://docs.scala-lang.org/overviews/core/actors-migration-guide.html>.