Programming in Scala

Lecture Four

Angelo Corsaro

15 December 2017

Chief Technology Officer, ADLINK Technology Inc.

Table of contents

Implicits

Scala Implcits

Scala implicit definitions provide a general mechanism to control how the compiler resolves types or argument mismatches.

In other terms, it control the set of transformation available to the compiler to *fix* a compile time error.

Example

Suppose we have a function that sums the elements of a lists and we want to also use it to sum arrays.

If we just pass an array the compiler will raise a type error as he does not know how to convert an Array[Int] into a List[Int].

```scala
def sum(xs: List[Int]): Int = xs match {
  case List() => 0
  case y::ys => y + sum(ys)
}

sum(Array[Int](1,3,4,5,6)) // Compiler error!
```

Implicit Functions

To fix this error and allow this function operate on Arrays as well as on List we can define an implicit conversion as shown below:

```scala
import scala.language.implicitConversions

object Converter {

  implicit def array2List(as: Array[Int])= as.toList

  implicit def strintToInt(s: String) = s.toInt

}

import Converter._

sum(Array(1,3,4,5,6)) // Compiler inserts the conversion array2List
sum(Array[Int](1,"3",4,"5",6)) // Compiler inserts the conversions array2List and stringToInt
```

Implicit Classes

Implicit classes are commonly used to emulate *open classes* in Scala.

In other terms, one can define implicit transformation to extend or enrich the protocol supported bu given type.

Example

```
object Implicits {

  implicit class RichRunnable[T](runner:  => T) extends Runnable {
    def run() { runner() }
  }

}

val thread = new Thread {
        print("Hello!")
}
```

Implicit Parameters

Implicit parameters can be used to let the compiler fill in missing parameters in curried functions.

Example

```
object Logger {
  def log(msg: String)(implicit prompt: String) {
    println(prompt + msg)
  }
}

implicit val defaultPrompt = ">>> "

Logger.log("Testing logger")
```

Implicit Resolution Rules

The compiler looks for potential conversions available in the current scope.

Thus, conversions needs to be imported in order for the compiler to apply them.

Monads: A Gentle Introduction

Monad: Basic Idea

Monads can be thought of as composable computation descriptions.

The essence of monad is the separation of composition timeline from the composed computation's execution timeline, as well as the ability of computation to implicitly carry extra data, as pertaining to the computation itself, in addition to its one (hence the name) output, that it will produce when run (or queried, or called upon).

This lends monads to supplementing pure calculations with features like I/O, common environment, updatable state, etc.

Monad: Basic Idea / Cont.

Each monad, or computation type, provides means, subject to Monad Laws, to:

- create a description of a computation that will produce (a.k.a. ”return”) a value, and
- combine (a.k.a. ”bind”) a computation description with a *reaction* to it, a pure function that is set to receive a computation-produced value (when and if that happens) and return another computation description.

Reactions are thus computation description constructors. A monad might also define additional primitives to provide access to and/or enable manipulation of data it implicitly carries, specific to its nature; cause some specific side-effects; etc..

Monad Class and Monad Laws

Monads can be viewed as a standard programming interface to various data or control structures, which is captured by the Monad class. All common monads are members of it:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a ->  m b        -> m b
  return ::   a                -> m a
  fail   :: String             -> m a
```

In addition to implementing the class functions, all instances of Monad should obey the following equations, or Monad Laws:

```
return a >>= k                  =  k a
m        >>= return             =  m
m        >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Maybe Monad

The Haskell Maybe type is defined as:

```haskell
data Maybe a = Just a | Nothing
     deriving (Eq, Ord)
```

It is an instance of the Monad and Functor classes.

Maybe Monad Implementation

Let's see how the Monad operations are defined for the Maybe type:

```haskell
return :: a -> Maybe a
return x = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) Nothing _ = Nothing
(>>=) (Just x) f = f x

(>>) :: Maybe a ->  Maybe b -> Maybe b
(>>) Nothing _ = Nothing
(>>) (Just x) Nothing = Nothing
(>>) x y =  x >>= (\_ -> y)
```

Verifying the First Monad Laws for Maybe

The first law states that:

```
return a >>= k =  k a
```

Let's play the substitution game:

```
return a >>= Just a >>= k  = k a
```

Verifying the Second Monad Laws for Maybe

The second law states that:

```
m >>= return =  m
```

Let's play the substitution game:

```
Just x >>= return = return x = Just x
Nothing >>= return = Nothing
```

Verifying the Third Monad Laws for Maybe

The third law states that:

```
m  >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Let's play the substitution game:

```
Just x  >>= (\y -> k y >>= h)  =  Just k x >>= h = (Just k x) >>= h = (Just x >>= k) >>= h
```

Thus we have proved that the Maybe type satisfies the three key monads rule. There are types you are already familiar with that satisfy the monadic laws, such as lists.

Monads in Scala

Scala does not define a Monad higher-order type, but monadic type (as well as some time that are not strictly monads), provide the key monadic operations.

These operations, however are named differently.

Option Type in Scala

The **Option** type is Scala's equivalent to Haskell's *Maybe* Monad. Looking at it is a good way of learning about monadic computations in Scala.

The *Option* type is defined as follows:

```scala
sealed abstract class Option[+A] extends Product with Serializable

case object None extends Option[Nothing] {
  // ...
}
final case class Some[+A](value: A) extends Option[A] {
 // ...
}
```

Option Type in Scala

The monad operations provided are the following:

```scala
// this is  the equivalent of Haskell's bind (>>=)
def flatMap[B](f: A => Option[B]): Option[B]
```

Notice that the equivalent of *return* is the constructor. That said, Scala does not provide the sequencing operator ($>>$). But as we've seen that can be easily defined from bind.

The Option type is technically also a functor (as lists also are), thus it also provides the *map* operation.

```scala
// this is  the equivalent of Haskell's bind (>>=)
def map[B](f: A => B): B
```

Monadic Computation with the Option Type

```
def parseInt(s: String): Option[Int] = {
  try {
    s.toInt
  } catch {
    case e: java.lang.NumberFormatException => None
  }
}

val x = parseInt("10")
val y = parseInt("10")

val z = x.flatMap(a => y.flatMap(b => Some(a+b)))

// Or using for

for (a <- x; b <- y) yield (a + b)
```

Parallel and Concurrent Computations

Motivation

As the number of cores on processors are constantly increasing while the clock frequencies are essentially stagnating, exploiting the additional computational power requires to leverage parallel and concurrent computations.

Scala provides some elegant mechanism for supporting both parallel as well as concurrent computations.

Parallel Collections

Scala supports the following parallel collections:

- ParArray
- ParVector
- mutable.ParHashMap
- mutable.ParHashSet
- immutable.ParHashMap
- immutable.ParHashSet
- ParRange
- ParTrieMap

Using Parallel Collections

Using parallel collections is rather trivial. A parallel collection can be created from a regular one by using the *.par* property.

Example

```scala
val parArray = (1 to 10000).toArray.par
parArray.fold(0)(_ + _)

val lastNames = List("Smith","Jones","Frankenstein","Bach","Jackson","Rodin").par
lastNames.map(_.toUpperCase)
```

Given this *out-of-order* semantics, also must be careful to perform only associative operations in order to avoid non-determinism.

Concurrent Computation on the JVM

The traditional approach to concurrent computations on the JVM is to leverage threads and the built-in support for monitors, through the *synchronized* keyword, to avoid race conditions.

This concurrency model is based on the assumption that different threads of execution collaborate by mutating some shared state. This shared state has to be protected to avoid race conditions.

This programming model is quite error-prone and is often source of bugs that are very hard to debug due to the inherent non-determinism of concurrent computations.

Furthermore, lock-based synchronization can lead to deadlocks.

Scala futures

Scala provide a mechanism for concurrent execution that makes it easier to work under the share nothing model and that support more importantly supports composability.

This mechanism is provided by Scala's futures.

Futures are monads, thus, you can operate and compose them as any other monad.

Futures Example

Below is an example of computing and composing futures.

```scala
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future

val f1 = Future {
  Thread.sleep(2)
  10
}

val f2 = Future {
  Thread.sleep(1)
  20
}

val c = for (a <- f1; b <- f2) yield a + b
c.onComplete(r => r.foreach(a => println(a)))
```

Promises

The most general way of creating a future and is through *promises*.

The completion of *future* create from a promise is controlled by completing the *promise*.

Example

```
def asynchComputation(): Future[Int]
        import Converters._
        val promise = Promise[Int]
        new Thread {
                val r = someComputeIntenseFun()
                promise.success(r)
        }.start()
        return promise.future
}
```

Homeworks

Reading Assignment

Read the following chapters:

- Chapter 21
- Chapter 32

Project Exposition

What

You will have 15 minutes to present your project. This is what I expect:

1. A few slides summarizing the project and showing some code snippets for the code you are most proud of
2. A walk-through your code – using the IDE to navigate
3. A demo demonstrating that your code actually works.
4. Ideally, I'd like to see the code on github and would like to receive the URL to your project at latest the day before the presentation.

Presentation Order

1. Gilles Ponnouradjane
2. Sanjiev Sellathurai
3. Wuhui Wu
4. Ozan Yildrim
5. Pierre Adam
6. Arnaud Cavrois
7. Hicham Derkaoui
8. Valentin Fabianski
9. Bohemond Flamand and Aris Lester
10. Sarah Khadir
11. Valentin Korenblit
12. Senda Li
13. Martin Mas

Where

The presentation will be done over the web, using Zoom. The web-conferencing coordinates are included below:

Topic: Scala Project Expo

Time: Jan 3, 2018 3:00 PM Paris

Join from PC, Mac, Linux, iOS or Android: `https://zoom.us/j/347843568`

Or Telephone:
France: +33 (0) 1 8288 0188
Meeting ID: 347 843 568
International numbers available:
`https://zoom.us/zoomconference?m=xt59vB8KxG26De9lFmh5zqVrJVQ2ybg9`