Programming in Scala

Lecture Three – Part B

Angelo Corsaro

20 October 2021

Chief Technology Officer, ADLINK Technology Inc.

Table of contents i

For Expression Revisited

For Expression

The general form of a **for** expression is:

for (*seq*) **yield** *expr*

Here, *seq* is a sequence of *generators*, *definitions*, and *filters*, with semicolons between successive elements.

Generator, Definition and Filter

A **generator** is of the form:

$$pat \leftarrow expr$$

A **definition** is of the form:

$$pat = expr$$

A **filter** is of the form:

$$if \quad expr$$

for expression: Examples

Example

```
1  for (x <- List(1, 2); y <- List("one", "two")) yield (x, y)
2
3  for ( i <- 1 to 10; j <- i to 10) yield (i,j)
4
5  val names = List("Morpheus", "Neo", "Trinity", "Tank", "Dozer")
6
7  for (name <- names; if name.length == 3) yield name
8
9  val xxs = List(List(1,2,3,4), List(6, 8, 10, 12), List(14, 16, 18))
10
11 for (xs <- xxs; e <- xs; if (e % 2 == 0)) yield e
12
13 for (xs <- xxs; if (xs.isEmpty == false); h = xs.head) yield h
```

Type Parameterization

Type Constructors in Scala

Beside first order types, Scala supports Type Constructors, – a kind of higher order types.

In the case of Scala, a parametric class is an n-ary type operator taking as argument one or more types, and returning another type.

Example
The list time we have seen this far, is a type constructor declared as:

$$class \quad List[+T]$$

Thus *List[Int]* and *List[String]* are two instances of the List[+T] type constructor.

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
(>>) 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 >>= k
Just a >>= k    ;; by definition of return
k a             ;; by definition of >>=
                ;; C.V.D.
```

Verifying the Second Monad Laws for Maybe

The second law states that:

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

Let's play the substitution game:

```
Just x >>= return       ;; by definition of >>=
return x = Just x       ;; by definition of return
                        ;; C.V.D.

Nothing >>= return = Nothing  ;; by definition of >>=
                              ;; C.V.D.
```

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)  ;; by definition of >>=
k x >>= h                      ;; adding parenthesis
(k x) >>= h                    ;; by definition of >>=
(Just x >>= k) >>= h           ;; C.V.D.
```

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:

```
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] {
 // ...
}
```

How we could do it in Scala 3

In Scala 3 we could have implemented an `Option`, which we'll call `Maybe` type using enums as follows:

```
enum Maybe[+T]:
  case Just(v: T)
  case Nothing extends Maybe[Nothing]

def flatMap[T, Q](f: T => Maybe[Q])(m: Maybe[T]): Maybe[Q] = m match
   case Maybe.Just(v) => f(v)
   case Maybe.Nothing => Maybe.Nothing
```

Notice how the `flatMap` has been written in curryed form.

Option Type in Scala

The monad operations provided are the following:

```
// 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.

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

Monadic Computation with the Option Type

```scala
def parseInt(s: String): Option[Int] = {
  try {
    Some(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

```
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.

```
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

Programming Exercies

- Make the factorial function a full function in `Int` by using the `Option` monad.
- Write a functional version of Insertion Sort for a `List[Int]`.
- Write a functional version of Quick Sort for a `List[Int]`.
- Write a functional program that given a `List[Int]` computes the the factorial of each member and then sums the results. **Hint:** the list may contain negative numbers, thus you better use the monadic factorial.
- Make the computation of each of the factorial happen in parallel using futures.

Reading

From the **Programming in Scala** book you should read:

- Chapter 10
- Chapter 11
- Chapter 15
- Chapter 19 (Variance concetps already covered in lecture 1 and 2)
- Chapter 23
- Chapter 32