Monads and scalaz – What, When, Why and How? (Part 1 – The State Monad)

FEBRUARY 28, 2016FEBRUARY 28, 2016

CAMERONJOANNIDIS

1 COMMENT

This is part 1 of a multipart blog series on a bunch of useful monads that scalaz offers that the scala standard library doesn't have. I found some of these concepts quite hard to learn initially because there isn't much clear content explaining the what, when, why and how. In this series I will attempt to remedy this and help to demystify these monads and help you understand where they can be applied, rather than just what they are.

I will finish the series with an example that pieces all these parts together so you can see how using these monads in reality actually looks.

This series assumes that you are familiar with the basics of monads and the scala language.

Now without further ado…

The State Monad

What

The state monad is a simple wrapper around the following function:

```
1 | S => (S, A)
```

In short, given some state `S1`, it produces a new state `S2` along with a value `A` associated with the state change.

In scalaz you use it as `State[S, A]` which is simply a monad that wraps the above behaviour.

Why

In order to see why we would bother using this, lets consider how we might typically deal with mutable state. As a simple example to illustrate the concepts, say we have a `Generator` that simply generates a new `Int` every time a function is called.

```
1  case class Generator(private var seed: Int) {
2    def generateRandomInt(): Int = {
3      val rand = seed + 5
4      seed = seed + 1
5      rand
6    }
7  }
8
9  val gen = Generator(30)
10 gen.generateRandomInt()
11 // res0: Int = 36
12 gen.generateRandomInt()
13 // res1: Int = 37
14 gen.generateRandomInt()
15 // res2: Int = 38
```

This all looks quite simple, however every time you call the function `generateRandomInt()`, you get a different value. Calling the same function with the same value and getting a different result every time not only breaks referential transparency of that function, but it also means our code is harder to reason about since we need to know the state of an object to know what it is going to do.

So how can we improve this situation?

If we follow common functional programming wisdom, we would know that we should create a new object every time something changes to maintain immutability. In the above example, we might consider doing this by returning a new `Generator` that has a different internal state, along with the value it produced.

```
1  case class Generator(private val seed: Int) {
2    def generateRandomInt(): (Generator, Int) = {
3      (Generator(seed + 1), seed + 5)
4    }
5  }
6
7  val g = Generator(30)
8  val (g2, res0) = g.generateRandomInt()
9  // g2: Generator = Generator(31)
10 // res0: Int = 35
11 val (g3, res1) = g2.generateRandomInt()
12 // g3: Generator = Generator(32)
13 // res1: Int = 36
14 val (g4, res2) = g3.generateRandomInt()
15 // g4: Generator = Generator(33)
16 // res2: Int = 37
```

What have we done here?

As you can see, we have removed the internal state of the `Generator`. It now always returns the same value no matter how many times we call it. We only have two issues left to deal with.

1. We currently have an object that taken an `Int` with a method that returns a `(Generator, Int)` pair. What we really want is a method that takes a `Generator` and returns a `(Generator, Int)` (You'll see why this is important shortly).
2. We now need to explicitly capture the new `Generator` that is output and use it instead of the old one to get a new random value.

Lets rearrange our code to deal with the first point.

```
1  case class Generator(seed: Int)
2
3  object Generator {
4    def generateRandomInt(g: Generator): (Generator, Int) = {
5      (Generator(g.seed + 1), g.seed + 5)
6    }
7  }
8
9  import Generator._
10
11 val g = Generator(30)
12 val (g2, res0) = generateRandomInt(g)
13 // g2: Generator = Generator(31)
14 // res0: Int = 35
15 val (g3, res1) = generateRandomInt(g2)
16 // g3: Generator = Generator(32)
17 // res1: Int = 36
18 val (g4, res2) = generateRandomInt(g3)
19 // g4: Generator = Generator(33)
20 // res2: Int = 37
```

Now we have a `Generator` which is effectively our input data, and a function with the signature `Generator =>(Generator, Int)`. This looks exactly like the definition of the state monad from earlier: `S =>(S, A)`. It is still really tedious to use though because we have to capture the new generator and feed it through the computation for every step.

This is where the state monad comes in.

Since we have a function `generateRandomInt(g: Generator): (Generator, Int)` that effectively models a state change of the `Generator` object passed in without any side effects, it is a perfect candidate for the State monad.

We can rewrite our function signature now as `generateRandomInt(): State[Generator, Int]`.

The meaning of our function remains the same, but since the computation is now a monad, it makes working with the changing state really simple.

We can now write code like this:

```
1  import scalaz._
2  import Scalaz._
3
4  case class Generator(seed: Int)
5
6  object Generator {
7    def generateRandomInt(): State[Generator, Int] = ???  // We will c
8  }
9
10 import Generator._
11
12 val computation = for {
13   x <- generateRandomInt()
14   y <- generateRandomInt()
15 } yield (x, y)
16
17 val (newGenerator, (xOutput, yOutput)) = computation.run(Generator(5
18 // newGenerator: Generator = Generator(59)
19 // xOutput: Int = 62
20 // yOutput: Int = 63
```

As you can see we now have a side effect free, stateful computation that has very little change when compared to the initial mutable version.

How

The last part of this is to understand how we implement the function: `def generateRandomInt(): State[Generator, Int]`.

Scalaz defines a bunch of helper functions to work with State monads in `StateOps`. The two we are going to use here are `get` and `put`.

```
1  def get[S]: State[S, S] = State(s => (s, s))
2
3  def put[S](s: S): State[S, Unit] = State(_ => (s, ()))
```

Before we use them, lets quickly have a look at what they do.

`get[S]` returns a state monad of type `State[S, S]`. In doing this, it passes the current state into both the new state AND the output value. This means that in our generator case, we would have `State[Generator, Generator]` which would allow us to actually extract the generator in our for comprehension.

(A simple way to think about it is that the for comprehension plucks out the value `A` from a state monad, and updates the state `S` without you having to explicitly deal with it like we were before).

So if we had:

```
1  for {
2    x <- ourState: State[Generator, A]
3  } yield ()
```

The value of x would be of type `A`. Since `get` creates `State[S, S]`, that means the value of x above would be of the same type as the state i.e. `Generator`.

So in the following snippit, x would be of type `Generator`.

```
for {
  x <- get[Generator]
} yield ()
```

One important bit of intuition that I initially struggled with here is that `get[Generator]` appears to pluck some `Generator` state out of thin air. The reason this works is that we aren't actually doing any computation yet here. Recall that State wraps `S =>(S, A)`. We are actually saying that "when we have an S, get it". It is not until later when you actually invoke this method that you will provide that state `S` and thus have it available to you. This is more clear when you realise that the type of this for comprehension is actually `State[Generator, Unit]`. We are simply describing a stateful computation. To run the actual computation we have to call the `run` method on a state monad. We will see an example of this shortly.

Now the other function we mentioned we were going to use was `put`. We can see that `put(s: S)` creates a new State monad of type `State[S, Unit]`. To do this, it actually ignores the current state, and simply creates a new state using the value `s` we pass in.

So for example, if we wanted to update our State to hold a new `Generator` we might be able to leverage this.

Putting this all together, I encourage the reader to try and implement `def getRandomInt(): State[Generator, Int]` before looking at the implementation below.

```
object Generator {
  def generateRandomInt(): State[Generator, Int] = {
    for {
      g <- get[Generator]             // Get the current state
      _ <- put(Generator(g.seed + 1)) // Update the state with the new
    } yield g.seed + 5                // return the "randomly" generat
  }
}
```

As you can see, we simply `get` the state of the current `Generator`, create a new one with a new seed and return our "random" number.

This is a simplified example of how you can use the state monad to produce side effect free state changes. In order to get there, aside from the State monad usage, it should be noted that we restructured our code to follow more of a functional paradigm which consists of Data and functions. The generator was converted from a stateful object, into a simple data type and our `generateRandomInt()` became a stateless function. This is quite a powerful paradigm but takes a while to become comfortable with when starting with functional programming.

When

The final question is when do I use this?

In general, the state monad can be used anywhere you have stateful or side effecting computations. We saw a simple example above with our Generator, but we can extend this for anything. For example, we often use side effecting log statements to log things in production. Instead of modelling this as simple side effecting statements in our functions, we would treat it as a stateful computation where the state is the statements to log and the value returned is the value normally returned by the function.

e.g.

```
1  def add(a: Int, b: Int): Int = {
2    log.info(s"Adding $a and $b")
3    a + b
4  }
5
6  add(5, 3)  // Produces a side effect
```

can be rewritten as:

```
1  def add(a: Int, b: Int): State[List[String], Int] = {
2    for {
3      curr <- get[List[Sting]]
4      _ <- put(curr +: s"Adding $a and $b")
5    } yield a + b
6  }
7
8  add(5, 3) // explicitly captures the side effect
```

We will actually see in a later blog that this particular use case of the state monad is called the `Writer` monad.

As another example (since I love concrete examples), consider a Robot that has two buttons, a light and an (x,y) position. Instead of having the robot mutate its own internal state whilst moving around and responding to button presses, we can do a similar thing we did with the generator example above and define functions that transform the robot “data”. (Note that I use the word data since the robot has a fixed internal state and is thus effectively just data).

```scala
import scalaz._
import Scalaz._

// Lets create a few types of buttons to push
sealed trait Button
case object Button1 extends Button
case object Button2 extends Button

// And represent our different light states
sealed trait LightStatus
case object On extends LightStatus
case object Off extends LightStatus

// Our simple robot has a position and a light that is in a particula
case class Robot(x: Int, y: Int, lightStatus: LightStatus)

// When we press a button, we want to turn on the light and move the
// Since the state of the robot will change, we can model this as a
// State monad.
// You can read the type signature State[Robot, Unit] as Robot => (Ro
// This essentially mean "Given some robot state R1, this operation w
// a new robot state R2 and a value associated with that state change
// In this case, the value returned is a Unit since moving a robot do
// create an actual value (simply a side effect in the real world).
// (There are ways to deal with this side effect more explicitly usin
// IO monad, but for now lets not worry about that.
def pressButton(b: Button): State[Robot, Unit] = {
  for {
    currS <- get[Robot]
    _ <- turnOnLight()
    moveSideEffect <- move(b)
  } yield moveSideEffect
}

def turnOnLight(): State[Robot, Unit] = {
  for {
    currS <- get[Robot]
    producedValue <- put(currS.copy(lightStatus = On))  // Create a n
  } yield producedValue                                 // This could
                                                        // for clarit
}

def move(button: Button): State[Robot, Unit] = {
  for {
    currS <- get[Robot]
    producedValue <- put {
      button match {
        case Button1 =>
          // pressing button 1 moves the robot forward
          currS.copy(y = currS.y + 1)
        case Button2 =>
          // pressing button 2 moves the robot backward
          currS.copy(y = currS.y - 1)
      }
    }
  } yield producedValue
}

pressButton(Button1).run(Robot(3, 4, Off))
// res0: (Robot(3,5,On),())
```

As can be seen, the state monad can lead to really nice composable programs that model mutable problems without actually mutating the objects we're working with.

Stay tuned for the next installment where we will be investigating the `Writer` monad.