25 November 2013

Understanding the State Monad

After recently tweeting about trampolines in Scala, I was subsequently asked about the state monad, as the paper from the tweet talks about implementing `State` in terms of `Free`. So then, it seemed only reasonable to write up a quick post about how to use the "normal" `State[S,A]` monad in Scalaz.

For the sake of this post, we will be implementing a process that is no doubt familiar to anyone who has ever driven, ran, cycled or otherwise on a highway: the ubiquitous traffic light - using the UK signalling semantics, obviously. These traffic signals are just rotating through a set of finite states: red to amber, amber to green and so forth. In this way, the light has a given state, and it can move between certain states but not others, and if the light malfunctions it needs to revert to flashing red lights to act as a stop sign. This could of course be implemented using mutable state to keep track of the currently enabled lights etc, but that would suck. The `scalaz.State` monad allows us to model these state transitions in an immutable fashion, and transparently - but intuitively - thread that state through the computation. Let's get cracking!

Algebraic data types: we gots' em'

So, first up, let's construct a super simple ADT set that models the various parts of a traffic light: colour aspects, modes of operations, and the states of a given signal display set:

```
sealed trait Aspect
case object Green extends Aspect
case object Amber extends Aspect
case object Red   extends Aspect

sealed trait Mode
case object Off      extends Mode
case object Flashing extends Mode
case object Solid    extends Mode

// represents the actual display set: must be enabled before
// it can be used.
case class Signal(
    isOperational: Boolean,
    display: Map[Aspect, Mode])

object Signal {
  import scalaz.syntax.state._
  import scalaz.State, State._

  // just a lil' bit of sugar to use later on.
  type ->[A,B] = (A,B)

  // convenience alias as all state ops here will deal
  // with signal state.
  type SignalState[A] = State[Signal,A]

  // dysfunctional lights revert to their flashing
  // red lights to act as a stop sign to keep folks safe.
  val default = Signal(
    isOperational = false,
    display = Map(Red -> Flashing, Amber -> Off, Green -> Off))
}
```

This is the shell of the work right here, so now we just need to get to the meat of the matter and implement the state transitioning functions.

Enabling Signals

Let's start by assuming that the signal is entirely disabled, so we need to make it operational:

```
def enable: State[Signal, Boolean] =
  for {
    a <- init
    _ <- modify((s: Signal) => s.copy(isOperational = true))
    r <- get
  } yield r.isOperational
```

This block of code firstly (i.e. the first generator of the comprehension) initialises the `State`. It's purpose in life is simply that: no more and no less - it gives you the "context" for the stateful computation, so to speak. This can easily be comprehended by looking at the definition in the Scalaz source:

```
// State.scala in scalaz
def init[S]: State[S, S] = State(s => (s, s))
```

It just lifts the `S` into the `State` constructor. Simple. The next line in the comprehension is the one doing the actual work of modifying the state, which in this case is a `Signal` instance. As you can see, its just accepting a function that takes a given `Signal` and sets the `isOperational` value. Internally, the `State` monad machinery applies this function to the given `Signal` instance that is being operated on. This example is of course trivial, but i will show later why this is even useful. Finally, the `get` method is also from `State`, and is defined to read the current state, whatever that might be:

```
// State.scala in scalaz
def get[S]: State[S, S] = init
```

Hopefully that was straight forward - essentially the `State` monad just provides us a way to read / set / replace some value of some application state, that we don't currently have, and then later read said state without ever directly "having" an instance of that data type. Nifty, eh?

The astute reader will be thinking about higher-order functions right about now.

Flashing lights all around

The guts of this particular example: changing the light configuration. In order to not over complicate (I assume real-life is not this trivial), the `change` method just takes arguments of tuples, which are then applied to to a given `Signal` instance using the `display` method, which in turn just stuffs that into the `signal.display` value (I rely on it being a map, and that each aspect in a light is unique - again, this will not every conceivable case, but it serves the example)

```
def change(seq: Aspect -> Mode*): State[Signal, Map[Aspect, Mode]] =
  for {
    m <- init
    _ <- modify(display(seq))
    s <- get
  } yield s.display

// FIXME: requires domain logic to prevent invalid state changes
// or apply any other domain rules that might be needed.
// I leave that as an exercise for the reader.
def display(seq: Seq[Aspect -> Mode]): Signal => Signal = signal =>
  if(signal.isOperational)
```

```
    signal.copy(display = signal.display ++ seq.toMap)
  else default
```

So this is our "primitive" for changing lights, but it will be an awkward user API if they have to constantly pass in the exact combination of the entire world of lights, so let's make their lives easy by making some combinators of the possible operations.

Combinators FTW

There are four common states that a UK traffic light might be in:

- All stop: solid red
- Get Ready: solid green + solid amber
- Go: solid green
- Prepare to stop: solid amber

Let's make combinators that use the `change` method under the hood:

```
// common states the signal can be in.
def halt  = change(Red -> Solid, Amber -> Off,   Green -> Off)
def ready = change(Red -> Solid, Amber -> Solid, Green -> Off)
def go    = change(Red -> Off,   Amber -> Off,   Green -> Solid)
def slow  = change(Red -> Off,   Amber -> Solid, Green -> Off)
```

Awesome. And putting that all together in a small application that we can run:

```
def main(args: Array[String]): Unit = {
  import Signal._
  import scalaz.State.{get => current}

  val program = for {
    _  <- enable
    r0 <- current // debuggin
    _  <- halt
    r1 <- current // debuggin
    _  <- ready
    r2 <- current // debuggin
    _  <- go
    r3 <- current // debuggin
    _  <- slow
    r4 <- current
  } yield r0 :: r1 :: r2 :: r3 :: r4 :: Nil

  program.eval(default).zipWithIndex.foreach { case (v,i) =>
    println(s"r$i - $v")
  }
}
```

Which when you run it, will print to the console each intermediate step (which of course you wouldn't do in practice, but i do here simply to illustrate the resulting alteration occurring at each step of the contextual state provided by the `State` monad):

```
r0 - Signal(true,Map(Red -> Flashing, Amber -> Off, Green -> Off))
r1 - Signal(true,Map(Red -> Solid, Amber -> Off, Green -> Off))
r2 - Signal(true,Map(Red -> Solid, Amber -> Solid, Green -> Off))
r3 - Signal(true,Map(Red -> Off, Amber -> Off, Green -> Solid))
r4 - Signal(true,Map(Red -> Off, Amber -> Solid, Green -> Off))
```

And that's all there is too it - state monad is really powerful, and is so general it can be applied to many cases where you might otherwise use mutation.