One Div Zero

An exploration of software development.

Tuesday, November 6, 2007

Monads are Elephants Part 4

Until you experience an adult elephant first hand you won't really understand just how big they can be. If monads are elephants then so far in this series of articles I've only presented baby elephants like List and Option. But now it's time to see a full grown adult pachyderm. As a bonus, this one will even do a bit of circus magic.

Functional Programming and IO

In functional programming there's a concept called referential transparency. Referential transparency means you can call a particular function anywhere and any time and the same arguments will always give the same results. As you might imagine, a referentially transparent function is easier to use and debug than one that isn't.

There's one area where referential transparency would seem impossible to achieve: IO. Several calls to the same readLine console function may result in any number of different strings depending on things like what the user ate for breakfast. Sending a network packet may end in successful delivery or it might not.

But we can't get rid of IO just to accomplish referential transparency. A program without IO is just a complicated way to make your CPU hot.

You might guess that monads provide a solution for referentially transparent IO given the topic of this series but I'm going to work my way up from some simple principles. I'll solve the problem for reading and writing strings on the console but the same solution can be extended to arbitrary kinds of IO like file and network.

Of course, you may not think that referentially transparent IO is terribly important in Scala. I'm not here to preach the one true way of purely functional referential transparency. I'm here to talk about monads and it just so happens that the IO monad is very illustrative of how several monads work.

The World In a Cup

Reading a string from the console wasn't referentially transparent because readLine depends on the state of the user and "user" isn't one of its parameters. A file reading function would depend on the state of the file system. A function that reads a web page would depend on the state of the target web server, the Internet, and the local network. Equivalent output functions have similar dependencies.

All this could be summed up by creating a class called WorldState and making it both a parameter and a result for all IO functions. Unfortunately, the world is a big place. My first attempt to write a

WorldState resulted in a compiler crash as it ran out of memory. So instead I'll try for something a bit smaller than modeling the whole universe. That's where a bit of circus magic comes in.

The slight-of-hand I'll use is to model only a few aspects of the world and just pretend WorldState knows about the rest of the world. Here are some aspects that would be useful

1. The state of the world changes between IO functions.
2. The world's state is what it is. You can't just create new ones whenever you want (val coolWorldState = new WorldState(){def jamesIsBillionaire = true}).
3. The world is in exactly one state at any moment in time.

Property 3 is a bit tricky so let's deal with properties 1 and 2 first.

Here's a rough sketch for property 1

```
//file RTConsole.scala
object RTConsole_v1 {
  def getString(state: WorldState) =
    (state.nextState, Console.readLine)
  def putString(state: WorldState, s: String) =
    (state.nextState, Console.print(s) )
}
```

getString and putString use functions defined in scala.Console as raw primitive functions. They take a world state and return a tuple consisting of a new world state and the result of the primitive IO.

Here's how I'll implement property 2

```
//file RTIO.scala
sealed trait WorldState{def nextState:WorldState}

abstract class IOApplication_v1 {
  private class WorldStateImpl(id:BigInt)
      extends WorldState {
    def nextState = new WorldStateImpl(id + 1)
  }
  final def main(args:Array[String]):Unit = {
    iomain(args, new WorldStateImpl(0))
  }
  def iomain(
      args:Array[String],
      startState:WorldState):(WorldState, _)
}
```

WorldState is a sealed trait; it can only be extended within the same file. IOApplication defines the only implementation privately so nobody else can instantiate it. IOApplication also defines a main function that can't be overridden and calls a function named iomain that must be implemented in a subclass. All of this is plumbing that is meant to be hidden from programmers that use the IO library.

Here's what hello world looks like given all this

```
// file HelloWorld.scala
class HelloWorld_v1 extends IOApplication_v1 {
  import RTConsole_v1._
```

```
  def iomain(
        args:Array[String],
        startState:WorldState) =
    putString(startState, "Hello world")
}
```

That Darn Property 3

The 3rd property said that the world can only be in one state at any given moment in time. I haven't solved that one yet and here's why it's a problem

```
class Evil_v1 extends IOApplication_v1 {
  import RTConsole_v1._
  def iomain(
      args:Array[String],
      startState:WorldState) = {
    val (stateA, a) = getString(startState)
    val (stateB, b) = getString(startState)
    assert(a == b)
    (startState, b)
  }
}
```

Here I've called getString twice with the same inputs. If the code was referentially transparent then the result, a and b, should be the same but of course they won't be unless the user types the same thing twice. The problem is that "startState" is visible at the same time as the other world states stateA and stateB.

Inside Out

As a first step towards a solution, I'm going to turn everything inside out. Instead of iomain being a function from WorldState to WorldState, iomain will return such a function and the main driver will execute it. Here's the code

```
//file RTConsole.scala
object RTConsole_v2 {
  def getString = {state:WorldState =>
    (state.nextState, Console.readLine)}
  def putString(s: String) = {state: WorldState =>
    (state.nextState, Console.print(s))}
}
```

getString and putString no longer get or put a string - instead they each return a new function that's "waiting" to be executed once a WorldState is provided.

```
//file RTIO.scala
sealed trait WorldState{def nextState:WorldState}

abstract class IOApplication_v2 {
  private class WorldStateImpl(id:BigInt)
      extends WorldState {
    def nextState = new WorldStateImpl(id + 1)
  }
  final def main(args:Array[String]):Unit = {
    val ioAction = iomain(args)
```

```
    ioAction(new WorldStateImpl(0));
  }
  def iomain(args:Array[String]):
    WorldState => (WorldState, _)
}
```

IOApplication's main driver calls iomain to get the function it will execute, then executes that function with an initial WorldState. HelloWorld doesn't change too much except it no longer takes a WorldState.

```
//file HelloWorld.scala
class HelloWorld_v2 extends IOApplication_v2 {
  import RTConsole_v2._
  def iomain(args:Array[String]) =
    putString("Hello world")
}
```

At first glance we seem to have solved our problem because WorldState is nowhere to be found in HelloWorld. But it turns out it's just been buried a bit.

Oh That Darn Property 3

```
class Evil_v2 extends IOApplication_v2 {
  import RTConsole_v2._
  def iomain(args:Array[String]) = {
    {startState:WorldState =>
      val (statea, a) = getString(startState)
      val (stateb, b) = getString(startState)
      assert(a == b)
      (startState, b)
    }
  }
}
```

Evil creates exactly the kind of function that iomain is supposed to return but once again things are broken. As long as the programmer can create arbitrary IO functions he or she can see through the WorldState trick.

Property 3 Squashed For Good

All we need to do is prevent the programmer from creating arbitrary functions with the right signature. Um...we need to do what now?

Okay, as we saw with WorldState it's easy to prevent programmers from creating subclasses. So let's turn our function signature into a trait.

```
sealed trait IOAction[+A] extends
  Function1[WorldState, (WorldState, A)]

private class SimpleAction[+A](
   expression: => A) extends IOAction[A]...
```

Unlike WorldState we do need to create IOAction instances. For example, getString and putString are in a separate file but they would need to create new IOActions. We just need them to do so safely. It's a bit of a dilemma until we realize that getString and putString have two separate pieces: the piece that

does the primitive IO and the piece that turns the input world state into the next world state. A bit of a factory method might help keep things clean, too.

```
//file RTIO.scala
sealed trait IOAction_v3[+A] extends
  Function1[WorldState, (WorldState, A)]

object IOAction_v3 {
  def apply[A](expression: => A):IOAction_v3[A] =
    new SimpleAction(expression)

  private class SimpleAction [+A](
      expression: => A) extends IOAction_v3[A] {
    def apply(state:WorldState) =
      (state.nextState, expression)
  }
}

sealed trait WorldState{def nextState:WorldState}

abstract class IOApplication_v3 {
  private class WorldStateImpl(id:BigInt)
      extends WorldState {
    def nextState = new WorldStateImpl(id + 1)
  }
  final def main(args:Array[String]):Unit = {
    val ioAction = iomain(args)
    ioAction(new WorldStateImpl(0));
  }
  def iomain(args:Array[String]):IOAction_v3[_]
}
```

The IOAction object is just a nice factory to create SimpleActions. SimpleAction's constructor takes a lazy expression as an argument, hence the "=> A" annotation. That expression won't be evaluated until SimpleAction's apply method is called. To call SimpleAction's apply method, a WorldState must be passed in. What comes out is a tuple with the new WorldState and the result of the expression.

Here's what our IO methods look like now

```
//file RTConsole.scala
object RTConsole_v3 {
  def getString = IOAction_v3(Console.readLine)
  def putString(s: String) =
    IOAction_v3(Console.print(s))
}
```

And finally our HelloWorld class doesn't change a bit

```
class HelloWorld_v3 extends IOApplication_v3 {
  import RTConsole_v3._
  def iomain(args:Array[String]) =
    putString("Hello world")
}
```

A little thought shows that there's no way to create an Evil IOApplication now. A programmer simply has no access to a WorldState. It has become totally sealed away. The main driver will only pass a WorldState to an IOAction's apply method, and we can't create arbitrary IOAction subclasses with custom definitions of apply.

Unfortunately, we've got a combining problem. We can't combine multiple IOActions so we can't do something as simple as "What's your name", Bob, "Hello Bob."

Hmmmm, IOAction is a container for an expression and monads are containers. IOAction needs to be combined and monads are combinable. Maybe, just maybe...

Ladies and Gentleman I Present the Mighty IO Monad

The IOAction.apply factory method takes an expression of type A and returns an IOAction[A]. It sure looks like "unit." It's not, but it's close enough for now. And if we knew what flatMap was for this monad then the monad laws would tell us how to create map using it and unit. But what's flatMap going to be? The signature needs to look like def flatMap[B](f: A=>IOAction[B]):IOAction[B]. But what does it do?

What we want it to do is chain an action to a function that returns an action and when activated causes the two actions to occur in order. In other words, getString.flatMap{y => putString(y)} should result in a new IOAction monad that, when activated, first activates the getString action then does the action that putString returns. Let's give it a whirl.

```
//file RTIO.scala
sealed abstract class IOAction_v4[+A] extends
    Function1[WorldState, (WorldState, A)] {
  def map[B](f:A => B):IOAction_v4[B] =
    flatMap {x => IOAction_v4(f(x))}
  def flatMap[B](f:A => IOAction_v4[B]):IOAction_v4[B]=
    new ChainedAction(this, f)

  private class ChainedAction[+A, B](
      action1: IOAction_v4[B],
      f: B => IOAction_v4[A]) extends IOAction_v4[A] {
    def apply(state1:WorldState) = {
      val (state2, intermediateResult) =
        action1(state1);
      val action2 = f(intermediateResult)
      action2(state2)
    }
  }
}

object IOAction_v4 {
  def apply[A](expression: => A):IOAction_v4[A] =
    new SimpleAction(expression)

  private class SimpleAction[+A](expression: => A)
      extends IOAction_v4[A] {
    def apply(state:WorldState) =
      (state.nextState, expression)
  }
}
```

```
// the rest remains the same
sealed trait WorldState{def nextState:WorldState}

abstract class IOApplication_v4 {
  private class WorldStateImpl(id:BigInt) ...
```

The IOAction factory and SimpleAction remain the same. The IOAction class gets the monad methods. Per the monad laws, map is just defined in terms of flatMap and what we're using as unit for now. flatMap defers all the hard work to a new IOAction implementation called ChainedAction.

The trick in ChainedAction is its apply method. First it calls action1 with the first world state. This results in a second world state and an intermediate result. The function it was chained to needs that result and in return the function generates another action: action2. action2 is called with the second world state and the tuple that come out is the end result. Remember that none of this will happen until the main driver passes in an initial WorldState object.

A Test Drive

At some point you may have wondered why getString and putString weren't renamed to something like createGetStringAction/createPutStringAction since that's in fact what they do. For an answer, look at what happens when we stick 'em in our old friend "for".

```
object HelloWorld_v4 extends IOApplication_v4 {
  import RTConsole_v4._
  def iomain(args:Array[String]) = {
    for{
        _ <- putString(
            "This is an example of the IO monad.");
        _ <- putString("What's your name?");
        name <- getString;
        _ <- putString("Hello " + name)
    } yield ()
  }
}
```

It's as if "for" and getString/putString work together to create a mini language just for creating a complex IOActions.

Take a Deep Breath

Now's a good moment to sum up what we've got. IOApplication is pure plumbing. Users subclass it and create a method called iomain which is called by main. What comes back is an IOAction - which could in fact be a single action or several actions chained together. This IOAction is just "waiting" for a WorldState object before it can do its work. The ChainedAction class is responsible for ensuring that the WorldState is changed and threaded through each chained action in turn.

getString and putString don't actually get or put Strings as their names might indicate. Instead, they create IOActions. But, since IOAction is a monad we can stick it into a "for" statement and the result looks as if getString/putString really do what they say the do.

It's a good start; we've almost got a perfectly good monad in IOAction. We've got two problems. The first is that, because unit changes the world state we're breaking the monad laws slightly (e.g. m

flatMap unit === m). That's kinda trivial in this case because it's invisible. But we might as well fix it.

The second problem is that, in general, IO can fail and we haven't captured that just yet.

IO Errors

In monadic terms, failure is represented by a zero. So all we need to do is map the native concept of failure (exceptions) to our monad. At this point I'm going to take a different tack from what I've been doing so far: I'll write one final version of the library with comments inline as I go.

The IOAction object remains a convenient module to hold several factories and private implementations (which could be anonymous classes, but it's easier to explain with names). SimpleAction remains the same and IOAction's apply method is a factory for them.

```
//file RTIO.scala
object IOAction {
  private class SimpleAction[+A](expression: => A)
      extends IOAction[A] {
    def apply(state:WorldState) =
      (state.nextState, expression)
  }

  def apply[A](expression: => A):IOAction[A] =
    new SimpleAction(expression)
```

UnitAction is a class for unit actions - actions that return the specified value but don't change the world state. unit is a factory method for it. It's kind of odd to make a distinction from SimpleAction, but we might as well get in good monad habits now for monads where it does matter.

```
  private class UnitAction[+A](value: A)
      extends IOAction[A] {
    def apply(state:WorldState) =
      (state, value)
  }

  def unit[A](value:A):IOAction[A] =
    new UnitAction(value)
```

FailureAction is a class for our zeros. It's an IOAction that always throws an exception. UserException is one such possible exception. The fail and ioError methods are factory methods for creating zeroes. Fail takes a string and results in an action that will raise a UserException whereas ioError takes an arbitrary exception and results in an action that will throw that exception.

```
  private class FailureAction(e:Exception)
      extends IOAction[Nothing] {
    def apply(state:WorldState) = throw e
  }

  private class UserException(msg:String)
    extends Exception(msg)

  def fail(msg:String) =
    ioError(new UserException(msg))
  def ioError[A](e:Exception):IOAction[A] =
```

```
    new FailureAction(e)
}
```

IOAction's flatMap, and ChainedAction remain the same. Map changes to actually call the unit method so that it complies with the monad laws. I've also added two bits of convenience: >> and <<. Where flatMap sequences this action with a function that returns an action, >> and << sequence this action with another action. It's just a question of which result you get back. >>, which can be pronounced "then", creates an action that returns the second result, so 'putString "What's your name" >> getString' creates an action that will display a prompt then return the user's response. Conversely, <<, which can be called "before" creates an action that will return the result from the first action.

```
sealed abstract class IOAction[+A]
    extends Function1[WorldState, (WorldState, A)] {
  def map[B](f:A => B):IOAction[B] =
    flatMap {x => IOAction.unit(f(x))}
  def flatMap[B](f:A => IOAction[B]):IOAction[B]=
    new ChainedAction(this, f)

  private class ChainedAction[+A, B](
      action1: IOAction[B],
      f: B => IOAction[A]) extends IOAction[A] {
    def apply(state1:WorldState) = {
      val (state2, intermediateResult) =
        action1(state1);
      val action2 = f(intermediateResult)
      action2(state2)
    }
  }

  def >>[B](next: => IOAction[B]):IOAction[B] =
    for {
      _ <- this;
      second <- next
    } yield second

  def <<[B](next: => IOAction[B]):IOAction[A] =
    for {
      first <- this;
      _ <- next
    } yield first
```

Because we've got a zero now, it's possible to add a filter method by just following the monad laws. But here I've created two forms of filter method. One takes a user specified message to indicate why the filter didn't match whereas the other complies with Scala's required interface and uses a generic error message.

```
  def filter(
      p: A => Boolean,
      msg:String):IOAction[A] =
    flatMap{x =>
      if (p(x)) IOAction.unit(x)
      else IOAction.fail(msg)}
  def filter(p: A => Boolean):IOAction[A] =
    filter(p, "Filter mismatch")
```

A zero also means we can create a monadic plus. As some infrastructure for creating it, HandlingAction is an action that wraps another action and if that action throws an exception then it sends that exception to a handler function. onError is a factory method for creating HandlingActions. Finally, "or" is the monadic plus. It basically says that if this action fails with an exception then try the alternative action.

```
  private class HandlingAction[+A](
      action:IOAction[A],
      handler: Exception => IOAction[A])
      extends IOAction[A] {
    def apply(state:WorldState) = {
      try {
        action(state)
      } catch {
        case e:Exception => handler(e)(state)
      }
    }
  }

  def onError[B >: A](
      handler: Exception => IOAction[B]):
      IOAction[B] =
    new HandlingAction(this, handler)

  def or[B >: A](
      alternative:IOAction[B]):IOAction[B] =
    this onError {ex => alternative}
}
```

The final version of IOApplication stays the same

```
sealed trait WorldState{def nextState:WorldState}

abstract class IOApplication {
  private class WorldStateImpl(id:BigInt)
      extends WorldState {
    def nextState = new WorldStateImpl(id + 1)
  }
  final def main(args:Array[String]):Unit = {
    val ioaction = iomain(args)
    ioaction(new WorldStateImpl(0));
  }
  def iomain(args:Array[String]):IOAction[_]
}
```

RTConsole stays mostly the same, but I've added a putLine method as an analog to println. I've also changed getString to be a val. Why not? It's always the same action.

```
//file RTConsole.scala
object RTConsole {
  val getString = IOAction(Console.readLine)
  def putString(s: String) =
    IOAction(Console.print(s))
  def putLine(s: String) =
    IOAction(Console.println(s))
}
```

And now a HelloWorld application to exercise some of this new functionality. sayHello creates an action from a string. If the string is a recognized name then the result is an appropriate (or inappropriate) greeting. Otherwise it's a failure action.

Ask is a convenience method that creates an action that will display a specified string then get one. The >> operator ensures that the action's result will be the result of getString.

processsString takes an arbitrary string and, if it's 'quit' then it creates an action that will say goodbye and be done. On any other string sayHello is called. The result is combined with another action using 'or' in case sayHello fails. Either way the action is sequenced with the loop action.

Loop is interesting. It's defined as a val just because it can be - a def would work just as well. So it's not quite a loop in the sense of being a recursive function, but it is a recursive value since it's defined in terms of processString which in turn is defined based on loop.

The iomain function kicks everything off by creating an action that will display an intro then do what the loop action specifies.

Warning: because of the way the library is implemented this loop will eventually blow the stack. Do not use it in production code. Read the comments to see why.

```
object HelloWorld extends IOApplication {
  import IOAction._
  import RTConsole._

  def sayHello(n:String) = n match {
    case "Bob" => putLine("Hello, Bob")
    case "Chuck" => putLine("Hey, Chuck")
    case "Sarah" => putLine("Helloooo, Sarah")
    case _ => fail("match exception")
  }

  def ask(q:String) =
    putString(q) >> getString

  def processString(s:String) = s match {
    case "quit" => putLine("Catch ya later")
    case _ => (sayHello(s) or
        putLine(s + ", I don't know you.")) >>

        loop
  }

  val loop:IOAction[Unit] =
    for {
      name <- ask("What's your name? ");
      _ <- processString(name)
    } yield ()

  def iomain(args:Array[String]) = {
    putLine(
        "This is an example of the IO monad.") >>
    putLine("Enter a name or 'quit'") >>
    loop
```

```
  }
}
```

Conclusion for Part 4

In this article I've called the IO monad 'IOAction' to make it clear that instances are actions that are waiting to be performed. Many will find the IO monad of little practical value in Scala. That's okay, I'm not here to preach about referential transparency. However, the IO monad is one of the simplest monads that's clearly not a collection in any sense.

Still, instances of the IO monad can be seen as containers. But instead of containing values they contain expressions. flatMap and map in essence turn the embedded expressions into more complex expressions.

Perhaps a more useful mental model is to see instances of the IO monad as computations or functions. flatMap can be seen as applying a function to the computation to create a more complex computation.

In the last part of this series I'll cover a way to unify the container and computation models. But first I want to reinforce how useful monads can be by showing an application that uses an elephantine herd of monads to do something a bit more complicated.

James Iry at 7:50 PM