One Div Zero

An exploration of software development.

Tuesday, September 18, 2007

Monads are Elephants Part 1

Introductions to monads are bit of cottage industry on the Internet. So I figured, "why buck tradition?" But this article will present Scala's way of dealing with monads.

An ancient parable goes that several blind men were experiencing their first elephant. "It's a tree," one said while wrapping his arms around its legs. "A large snake," another said while holding its trunk. A third said...um, something about a broom or a fan or whatever.

From this parable we can conclude this: the ancients believed that the visually impaired like to fondle large mammals. Fortunately we live in a more enlightened age.

We're also supposed to learn something about how our limitations can prevent us from grasping the whole picture and that we're all blind in some way. It's so Zen.

I think there's a third lesson to be learned - the opposite of the main intent: that it's possible to learn much about the big picture by getting a series of limited explanations. If you had never seen an elephant and people told you things like "it has legs as thick as tree trunks," "a nose like a snake," "a tail like a broom," "ears like fans," etc. then you'd soon have a pretty good understanding. Your conception wouldn't be perfect, but when you finally saw an elephant it would fit neatly into the mental picture you had slowly built up. Just as the elephant was about to step on you, you'd think "wow, the legs really are like trees."

Monads are Container Types

One of the most commonly used container types is List and we'll spend some time with it. I also mentioned Option in a previous article. As a reminder, an Option is always either Some(value) or None. It might not be clear how List and Option are related, but if you consider an Option as a stunted List that can only have 0 or 1 elements, it helps. Trees and Sets can also be monads. But remember that monads are elephants, so with some monads you may have to squint a bit to see them as containers.

Monads are parameterized. List is a useful concept, but you need to know what's in the List. A List of Strings (List[String]) is pretty different from a List of Ints (List[Int]). Obviously it can be useful to convert from one to the other. Which leads us to the next point.

Monads Support Higher Order Functions

A higher order function is a function that takes a function as a parameter or returns a function as a result. Monads are containers which have several higher order functions defined. Or, since we're talking about Scala, monads have several higher order methods.

One such method is map. If you know any functional languages then you're probably familiar with map in one form or another. The map method takes a function and applies it to each element in the container to return a new container. For instance

```
def double(x: Int) = 2 * x
val xs = List(1, 2, 3)
val doubles = xs map double
// or val doubles = xs map {2 * _}
assert(doubles == List(2, 4, 6))
```

Map does not change the kind of monad, but may change its parameterized type...

```
val one = Some(1)
val oneString = one map {_.toString}
assert(oneString == Some("1"))
```

Here the {_.toString} notation means that the toString method should be called on the element.

Monads are Combinable

Now let's say we have a configuration library that let's us fetch parameters. For any parameter we'll get back an Option[String] - in other words we may or may not get a string depending on whether the parameter is defined. Let's say we also have a function, stringToInt, which takes a String and returns Some[Int] if the string is parseable as an integer or None if it's not. If we try to combine them using map we run into trouble.

```
val opString : Option[String] = config fetchParam "MaxThreads"
def stringToInt(string:String) : Option[Int] = ...
val result = opString map stringToInt
```

Unfortunately, since we started with an Option and mapped its contained element with a function that results in another Option, the variable "result" is now an Option that contains an Option, ie result is an Option[Option[Int]]. That's probably not terribly useful in most cases.

To motivate a solution, imagine if instead of Option we'd used List and ended up with List[List[Int]]] - in other words a list containing some number of lists. Given that, we just need "flatten" - a function which takes a list of lists (List[List[A]]) and returns a single list (List[A]) by concatenating everything together.[1]

A flatten function for Option[Option[A]] works a bit differently.

```
def flatten[A](outer:Option[Option[A]]) : Option[A] =
   outer match {
     case None => None
     case Some(inner) => inner
   }
```

If the outer option is None, then result is None. Otherwise the result is the inner Option.

These two flatten functions have similar signatures: they take an M[M[A]] and turn it into an M[A]. But the way they do it is quite different. Other monads would have their own ways of doing flatten - possibly quite sophisticated ways. This possible sophistication is why explanations of monads will often use "join" instead of "flatten." "Join" neatly indicates that some aspect of the outer monad may be combined (joined) with some aspect of the inner monad. I'll stick with "flatten," though, because it fits with our container analogy.

Now, Scala does not require you to write flatten explicitly. But it does require that each monad have a method called flatMap.[2]. What's flatMap? It's exactly what it sounds like: doing a map and then flattening the result.

```
class M[A] {
  private def flatten[B](x:M[M[B]]) : M[B] = ...
  def map[B](f: A => B) : M[B] = ...
  def flatMap[B](f: A => M[B]) : M[B] = flatten(map(f))
}
```

With that, we can revisit our problematic code...

```
val opString : Option[String] = config fetchParam "MaxThreads"
def stringToInt(string:String) : Option[Int] = ...
val result = opString flatMap stringToInt
```

Because of flatMap we end up with "result" being an Option[Int]. If we wanted, we could take result and flatMap it with a function from Int to Option[Foo]. And then we could faltMap that with a function from Foo to Option[Bar], etc.

If you're keeping score, many papers on monads use the word "bind" instead of "flatMap" and Haskell uses the ">>=" operator. It's all the same concept.

Monads Can Be Built In Different Ways

So we've seen how the flatMap method can be built using map. It's possible to go the other way: start with flatMap and create map based on it. In order to do so we need one more concept. In most papers on monads the concept is called "unit," in Haskell it's called "return." Scala is an object oriented language so the same concept might be called a single argument "constructor" or "factory." Basically, unit takes one value of type A and turns it into a monad of type M[A]. For List, unit(x) == List(x) and for Option, unit(x) == Some(x).

Scala does not require a separate "unit" function or method, and whether you write it or not is a matter of taste. In writing this version of map I'll explicitly write "unit" just to show how it fits into things.

```
class M[A](value: A) {
  private def unit[B] (value : B) = new M(value)
  def map[B](f: A => B) : M[B] = flatMap {x => unit(f(x))}
  def flatMap[B](f: A => M[B]) : M[B] = ...
}
```

In this version flatMap has to be built without reference to map or flatten - it will have to do both in one go. The interesting bit is map. It takes the function passed in (f) and turns it into a new function that is appropriate for flatMap. The new function looks like {x => unit(f(x))} meaning that first f is applied to x, then unit is applied to the result.

Conclusion for Part I

Scala monads must have map and flatMap methods. Map can be implemented via flatMap and a constructor or flatMap can be implemented via map and flatten.

flatMap is the heart of our elephantine beast. When you're new to monads, it may help to build at least the first version of a flatMap in terms of map and flatten. Map is usually pretty straight forward. Figuring out what makes sense for flatten is the hard part.

As you move into monads that aren't collections you may find that flatMap should be implemented first and map should be implemented based on it and unit.

In part 2 I'll cover Scala's syntactic sugar for monads. In part 3 I'll present the elephant's DNA: the monad laws. Finally, in part 4 I'll show a monad that's only barely a container. In the meantime, here's a cheat sheet for translating between computer science papers on monads, Haskell, and Scala.

Generic	Haskell	Scala
M	data M a or newtype M a or instance Monad (M a)	class M[A] or case class M[A] or trait M[A]
M a	M a	M[A]
unit v	return v	new M(v) or M(v)
map f m	fmap f m	m map f
bind f m	m >>= f or f =<< m	m flatMap f
join	join	flatten
	do	for

Footnotes

[1]. The Scala standard library includes a flatten method on List. It's pretty slick, but to explain it I would have to go into implicit conversions which would be a significant distraction. The slick part is that flatten makes sense on List[List[A]] but not on List[A], yet Scala's flatten method is defined on all Lists while still being statically type checked.

[2]. I'm using a bit of shorthand here. Scala doesn't "require" any particular method names to make a monad. You can call your methods "germufaBitz" or "frizzleMuck". However, if you stick with map and flatMap then you'll be able to use Scala's "for comprehensions"

James Iry at 6:14 PM

‹ Home ›

View web version

About Me

James Iry

San Francisco, CA, United States

If cars were built like software then...well, I don't know squat about building cars so who knows. It might be kinda cool. But probably not.

View my complete profile