One Div Zero

An exploration of software development.

Tuesday, October 2, 2007

Monads are Elephants Part 2

In part 1, I introduced Scala's monads through the parable of the blind men and the elephant. Normally we're supposed to take the view that each blind man doesn't come close to understanding what an elephant is, but I presented the alternative view that if you heard all the blind men describe their experience then you'd quickly build a surprisingly good understanding of elephants.

In part 2 I'm going to poke at the beast some more by exploring Scala's monad related syntactic sugar: "for comprehensions."

A Little "For"

A very simple "for" looks like this

```
val ns = List(1, 2)
val qs = for (n <- ns) yield n * 2
assert (qs == List(2, 4))
```

The "for" can be read as "for [each] n [in] ns yield n * 2." It looks like a loop, but it's not. This is our old friend map in disguise.

```
val qs = ns map {n => n * 2}
```

The rule here is simple

```
for (x <- expr) yield resultExpr
```

Expands to[1]

```
expr map {x => resultExpr}
```

And as a reminder, that's equivalent to

```
expr flatMap {x => unit(resultExpr)}
```

More "For"

One expression in a "for" isn't terribly interesting. Let's add some more

```
val ns = List(1, 2)
val os = List (4, 5)
val qs = for (n <- ns; o <- os)
   yield n * o
assert (qs == List (1*4, 1*5, 2*4, 2*5))
```

This "for" could be read "for [each] n [in] ns [and each] o [in] os yield n * o. This form of "for" looks a bit like nested loops but it's just a bit of map and flatMap.

```
val qs = ns flatMap {n =>
   os map {o => n * o }}
```

It's worth while to spend a moment understanding why this works. Here's how it gets computed (red italics gets turned into bold green):

1. `val qs = ns flatMap {n =>`
   `os map {o => n * o }}`
2. `val qs = ns flatMap {n =>`
   `List(n * 4, n * 5)`
3. `val qs =`
   `List(1 * 4, 1 * 5, 2 * 4, 2 * 5)`

Now With More Expression

Let's kick it up a notch.

```
val qs =
   for (n <- ns; o <- os; p <- ps)
      yield n * o * p
```

This "for" gets expanded into

```
val qs = ns flatMap {n =>
   os flatMap {o =>
      {ps map {p => n * o * p}}}}
```

That looks pretty similar to our previous "for." That's because the rule is recursive

```
for(x1 <- expr1;...x <- expr)
   yield resultExpr
```

expands to

```
expr1 flatMap {x1 =>
   for(...;x <- expr) yield resultExpr
}
```

This rule gets applied repeatedly until only one expression remains at which point the map form of expansion is used. Here's how the compiler expands the "val qs = for..." statement (again red italics turns to bold green)

1. `val qs =`
   `for (n <- ns; o <- os; p <- ps)`
   `yield n * o * p`

```
2. val qs =
      ns flatMap {n => for(o <- os; p <- ps)
      yield n * o * p}

3. val qs =
      ns flatMap {n => os flatMap {o =>
      for(p <- ps) yield n * o * p}}

4. val qs =
      ns flatMap {n => os flatMap {o =>
      {ps map {p => n * o * p}}}
```

An Imperative "For"

"For" also has an imperative version for the cases where you're only calling a function for its side effects. In it you just drop the yield statement.

```
val ns = List(1, 2)
val os = List (4, 5)
for (n <- ns; o <- os)  println(n * o)
```

The expansion rule is much like the yield based version but foreach is used instead of flatMap or map.

```
ns foreach {n => os foreach {o => println(n * o) }}
```

Now, you don't have to implement foreach if you don't want to use the imperative form of "for", but foreach is trivial to implement since we already have map.

```
class M[A] {
   def map[B](f: A=> B) : M[B] = ...
   def flatMap[B](f: A => M[B]) : M[B] = ...
   def foreach[B](f: A=> B) : Unit = {
       map(f)
       ()
   }
}
```

In other words, foreach can just call map and throw away the results. That might not be the most runtime efficient way of doing things, though, so Scala allows you to define foreach your own way.

Filtering "For"

So far our monads have built on a few key concepts. These three methods - map, flatMap, and forEach - allow almost all of what "for" can do.

Scala's "for" statement has one more feature: "if" guards. As an example

```
val names = List("Abe", "Beth", "Bob", "Mary")
val bNames = for (bName <- names;
   if bName(0) == 'B'
) yield bName + " is a name starting with B"

assert(bNames == List(
```

```
   "Beth is a name starting with B",
   "Bob is a name starting with B"))
```

"if" guards get mapped to a method called filter. Filter takes a predicate function (a function that takes on argument and returns true or false) and creates a new monad without the elements that don't match the predicate. The for statement above gets translated into something like the following.

```
val bNames =
   (names filter { bName => bName(0) == 'B' })
   .map { bName =>
      bName + " is a name starting with B"
   }
```

First the list is filtered for names that start with B. Then that filtered list is mapped using a function that appends " is a name..."

Not all monads can be filtered. Using the container analogy, filtering might remove all elements and some containers can't be empty. For such monads you don't need to create a filter method. Scala won't complain as long as you don't use an "if" guard in a "for" expression.

I'll have more to say about filter, how to define it in purely monadic terms, and what kinds of monads can't be filtered in the next installment

Conclusion for Part 2

"For" is a handy way to work with monads. Its syntax is particularly useful for working with Lists and other collections. But "for" is more general than that. It expands into map, flatMap, foreach, and filter. Of those, map and flatMap should be defined for any monad. The foreach method can be defined if you want the monad to be used imperatively and it's trivial to build. Filter can be defined for some monads but not for others.

"m map f" can be implemented as "m flatMap {x => unit(x)}. "m foreach f" can be implemented in terms of map, or in terms of flatMap "m flatMap {x => unit(f(x));()}. Even "m filter p" can be implemented using flatMap (I'll show how next time). flatMap really is the heart of the beast.

Remember, monads are elephants. The picture I've painted of monads so far emphasizes collections. In part 4, I'll present a monad that isn't a collection and only a container in an abstract way. But before part 4 can happen, part 3 needs to cover some properties that are true of all monads: the monadic laws.

In the mean time, here's a cheat sheet showing how Haskell's do and Scala's for are related.

Haskell	Scala

`do var1<- expn1`
`  var2 <- expn2`
`  expn3`
	`for {var1 <- expn1;`
	`  var2 <- expn2;`
	`  result <- expn3`
	`} yield result`

`do var1 <- expn1` `  var2 <- expn2`	`for {var1 <- expn1;` `  var2 <- expn2;`

`  return expn3`	`} yield expn3`

`do var1 <- expn1 >> expn2`
`  return expn3`
	`for {_ <- expn1;`
	`  var1 <- expn2`
	`} yield expn3`

Footnotes

[1]. The Scala spec actually specifies that "for" expands using pattern matching. Basically, the real spec expands the rules I present here to allow patterns on the left side of the <-. It would just muddy this article too much to delve too deeply into the subject.

James Iry at 9:03 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