One Div Zero

An exploration of software development.

Thursday, October 18, 2007

Monads are Elephants Part 3

In this series I've presented an alternative view on the old parable about the blind men and the elephant. In this view, listening to the limited explanations from each of the blind men eventually leads to a pretty good understanding of elephants.

So far we've been looking at the outside of monads in Scala. That's taken us a great distance, but it's time to look inside. What really makes an elephant an elephant is its DNA. Monads have a common DNA of their own in the form of the monadic laws.

This article is a lot to digest all at once. It probably makes sense to read it in chunks. It can also be useful to re-read by substituting a monad you already understand (like List) into the laws.

Equality for All

Before I continue, I have to semi-formally explain what I mean when I use triple equals in these laws as in "f(x) ≡ g(x)." What I mean is what a mathematician might mean by "=" equality. I'm just avoiding single "=" to prevent confusion with assignment.

So I'm saying the expression on the left is "the same" as the expression on the right. That just leads to a question of what I mean by "the same."

First, I'm not talking about reference identity (Scala's eq method). Reference identity would satisfy my definition, but it's too strong a requirement. Second, I don't necessarily mean == equality either unless it happens to be implemented just right.

What I do mean by "the same" is that two objects are indistinguishable without directly or indirectly using primitive reference equality, reference based hash code, or isInstanceOf.

In particular it's possible for the expression on the left to lead to an object with some subtle internal differences from the object on the right and still be "the same." For example, one object might use an extra layer of indirection to achieve the same results as the one on the right. The important part is that, from the outside, both objects must behave the same.

One more note on "the same." All the laws I present implicitly assume that there are no side effects. I'll have more to say about side effects at the end of the article.

Breaking the Law

Inevitably somebody will wonder "what happens if I break law x?" The complete answer depends on what laws are broken and how, but I want to approach it holistically first. Here's a reminder of some

laws from another branch of mathematics. If a, b, and c are rational numbers then multiplication (*) obeys the following laws:

1. a * 1 ≡ a
2. a * b ≡ b * a
3. (a * b) * c ≡ a * (b * c)

Certainly it would be easy to create a class called "RationalNumber" and implement a * operator. But if it didn't follow these laws the result would be confusing to say the least. Users of your class would try to plug it into formulas and would get the wrong answers. Frankly, it would be hard to break these laws and still end up with anything that even looks like multiplication of rational numbers.

Monads are not rational numbers. But they do have laws that help define them and their operations. Like arithmetic operations, they also have "formulas" that allow you to use them in interesting ways. For instance, Scala's "for" notation is expanded using a formula that depends on these laws. So breaking the monad laws is likely to break "for" or some other expectation that users of your class might have.

Enough intro. To explain the monad laws, I'll start with another weird word: functor.

WTF - What The Functor?

Usually articles that start with words like "monad" and "functor" quickly devolve into soup of Greek letters. That's because both are abstract concepts in a branch of mathematics called category theory and explaining them completely is a mathematical exercise. Fortunately, my task isn't to explain them completely but just to cover them in Scala.

In Scala a functor is a class with a map method and a few simple properties. For a functor of type M[A], the map method takes a function from A to B and returns an M[B]. In other words, map converts an M[A] into an M[B] based on a function argument. It's important to think of map as performing a transformation and not necessarily having anything to do with loops. It might be implemented as a loop, but then again it might not.

Map's signature looks like this

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

First Functor Law: Identity

Let's say I invent a function called identity like so

```
def identity[A](x:A) = x
```

This obviously has the property that for any x

identity(x) ≡ x

It doesn't do much and that's the point. It just returns its argument (of whatever type) with no change. So here's our first functor law: for any functor m

- F1. m map identity ≡ m // or equivalently *
- F1b. m map {x => identity(x)} ≡ m // or equivalently
- F1c. m map {x => x} ≡ m

In other words, doing nothing much should result in no change. Brilliant! However, I should remind you that the expression on the left can return a different object and that object may even have a different internal structure. Just so long as you can't tell them apart.

If you were to create a functor that didn't follow this law then the following wouldn't hold true. To see why that would be confusing, pretend m is a List.

F1d. for (x <- m) yield x ≡ m

Second Functor Law: Composition

The second functor law specifies the way several "maps" compose together.

F2. m map g map f ≡ m map {x => f(g(x))}

This just says that if you map with g and then map with f then it's exactly the same thing as mapping with the composition "f of g." This composition law allows a programmer to do things all at once or stretch them out into multiple statements. Based on this law, a programmer can always assume the following will work.

```
val result1 = m map (f compose g)
val temp = m map g
val result2 =  temp map f
assert result1 == result2
```

In "for" notation this law looks like the following eye bleeder

F2b. for (y<- (for (x <-m) yield g(x)) yield f(y) ≡ for (x <- m) yield f(g(x))

Functors and Monads, Alive, Alive Oh

As you may have guessed by now all monads are functors so they must follow the functor laws. In fact, the functor laws can be deduced from the monad laws. It's just that the functor laws are so simple that it's easier to get a handle on them and see why they should be true.

As a reminder, a Scala monad has both map and flatMap methods with the following signatures

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

Additionally, the laws I present here will be based on "unit." "unit" stands for a single argument constructor or factory with the following signature

```
def unit[A](x:A):M[A] = ...
```

"unit" shouldn't be taken as the literal name of a function or method unless you want it to be. Scala doesn't specify or use it but it's an important part of monads. Any function that satisfies this signature and behaves according to the monad laws will do. Normally it's handy to create a monad M as a case class or with a companion object with an appropriate apply(x:A):M[A] method so that the expression M(x) behaves as unit(x).

The Functor/Monad Connection Law: The Zeroth Law

In the very first installment of this series I introduced a relationship

> FM1. m map f ≡ m flatMap {x => unit(f(x))}

This law doesn't do much for us alone, but it does create a connection between three concepts: unit, map, and flatMap.

This law can be expressed using "for" notation pretty nicely

> FM1a. for (x <- m) yield f(x) ≡ for (x <- m; y <- unit(f(x))) yield y

Flatten Revisited

In the very first article I mentioned the concept of "flatten" or "join" as something that converts a monad of type M[M[A]] into M[A], but didn't describe it formally. In that article I said that flatMap is a map followed by a flatten.

> FL1. m flatMap f ≡ flatten(m map f)

This leads to a very simple definition of flatten

1. flatten(m map identity) ≡ m flatMap identity // substitute identity for f
2. FL1a. flatten(m) ≡ m flatMap identity // by F1

So flattening m is the same as flatMapping m with the identity function. I won't use the flatten laws in this article as flatten isn't required by Scala but it's a nice concept to keep in your back pocket when flatMap seems too abstract.

The First Monad Law: Identity

The first and simplest of the monad laws is the monad identity law

- M1. m flatMap unit ≡ m // or equivalently
- M1a. m flatMap {x => unit(x)} ≡ m

Where the connector law connected 3 concepts, this law focuses on the relationship between 2 of them. One way of reading this law is that, in a sense, flatMap undoes whatever unit does. Again the reminder that the object that results on the left may actually be a bit different internally as long as it behaves the same as "m."

Using this and the connection law, we can derive the functor identity law

1. m flatMap {x => unit(x)} ≡ m // M1a
2. m flatMap {x => unit(identity(x))}≡ m // identity
3. F1b. m map {x => identity(x)} ≡ m // by FM1

The same derivation works in reverse, too. Expressed in "for" notation, the monad identity law is pretty straight forward

M1c. for (x <- m; y <- unit(x)) yield y ≡ m

The Second Monad Law: Unit

Monads have a sort of reverse to the monad identity law.

- M2. unit(x) flatMap f ≡ f(x) // or equivalently
- M2a. unit(x) flatMap {y => f(y)} ≡ f(x)

The law is basically saying that unit(x) must somehow preserve x in order to be able to figure out f(x) if f is handed to it. It's in precisely this sense that it's safe to say that any monad is a type of container (but that doesn't mean a monad is a collection!).

In "for" notation, the unit law becomes

M2b. for (y <- unit(x); result <- f(y)) yield result ≡ f(x)

This law has another implication for unit and how it relates to map

1. unit(x) map f ≡ unit(x) map f // no, really, it does!
2. unit(x) map f ≡ unit(x) flatMap {y => unit(f(y))} // by FM1
3. M2c. unit(x) map f ≡ unit(f(x)) // by M2a

In other words, if we create a monad instance from a single argument x and then map it using f we should get the same result as if we had created the monad instance from the result of applying f to x. In for notation

M2d. for (y <- unit(x)) yield f(y) ≡ unit(f(x))

The Third Monad Law: Composition

The composition law for monads is a rule for how a series of flatMaps work together.

- M3. m flatMap g flatMap f ≡ m flatMap {x => g(x) flatMap f} // or equivalently
- M3a. m flatMap {x => g(x)} flatMap {y => f(y)} ≡ m flatMap {x => g(x) flatMap {y => f(y) }}

It's the most complicated of all our laws and takes some time to appreciate. On the left side we start with a monad, m, flatMap it with g. Then that result is flatMapped with f. On the right side, we create an

anonymous function that applies g to its argument and then flatMaps that result with f. Finally m is flatMapped with the anonymous function. Both have same result.

In "for" notation, the composition law will send you fleeing in terror, so I recommend skipping it

> M3b. for (a <- m;b <- g(a);result <- f(b)) yield result ≡ for(a <- m; result <- for(b < g(a); temp <- f(b)) yield temp) yield result

From this law, we can derive the functor composition law. Which is to say breaking the monad composition law also breaks the (simpler) functor composition. The proof involves throwing several monad laws at the problem and it's not for the faint of heart

1. m map g map f ≡ m map g map f // I'm pretty sure
2. m map g map f ≡ m flatMap {x => unit(g(x))} flatMap {y => unit(f(y))} // by FM1, twice
3. m map g map f ≡ m flatMap {x => unit(g(x)) flatMap {y => unit(f(y))}} // by M3a
4. m map g map f ≡ m flatMap {x => unit(g(x)) map {y => f(y)}} // by FM1a
5. m map g map f ≡ m flatMap {x => unit(f(g(x))} // by M2c
6. F2. m map g map f ≡ m map {x => f(g(x))} // by FM1a

Total Loser Zeros

List has Nil (the empty list) and Option has None. Nil and None seem to have a certain similarity: they both represent a kind of emptiness. Formally they're called monadic zeros.

A monad may have many zeros. For instance, imagine an Option-like monad called Result. A Result can either be a Success(value) or a Failure(msg). The Failure constructor takes a string indicating why the failure occurred. Every different failure object is a different zero for Result.

A monad may have no zeros. While all collection monads will have zeros (empty collections) other kinds of monads may or may not depending on whether they have a concept of emptiness or failure that can follow the zero laws.

The First Zero Law: Identity

If mzero is a monadic zero then for any f it makes sense that

> MZ1. mzero flatMap f ≡ mzero

Translated into Texan: if t'ain't nothin' to start with then t'ain't gonna be nothin' after neither.

This law allows us to derive another zero law

1. mzero map f ≡ mzero map f // identity
2. mzero map f ≡ mzero flatMap {x => unit(f(x)) // by FM1
3. MZ1b. mzero map f ≡ mzero // by MZ1

So taking a zero and mapping with any function also results in a zero. This law makes clear that a zero is different from, say, unit(null) or some other construction that may appear empty but isn't quite empty

enough. To see why look at this

> unit(null) map {x => "Nope, not empty enough to be a zero"} ≡ unit("Nope, not empty enough to be a zero")

The Second Zero Law: M to Zero in Nothing Flat

The reverse of the zero identity law looks like this

> MZ2. m flatMap {x => mzero} ≡ mzero

Basically this says that replacing everything with nothing results in nothing which um...sure. This law just formalizes your intuition about how zeros "flatten."

The Third and Fourth Zero Laws: Plus

Monads that have zeros can also have something that works a bit like addition. For List, the "plus" equivalent is ":::" and for Option it's "orElse." Whatever it's called its signature will look this

```
class M[A] {
   ...
   def plus(other:M[B >: A]): M[B] = ...
}
```

Plus has the following two laws which should make sense: adding anything to a zero is that thing.

- MZ3. mzero plus m ≡ m
- MZ4. m plus mzero ≡ m

The plus laws don't say much about what "m plus n" is if neither is a monadic zero. That's left entirely up to you and will vary quite a bit depending on the monad. Typically, if concatenation makes sense for the monad then that's what plus will be. Otherwise, it will typically behave like an "or," returning the first non-zero value.

Filtering Revisited

In the previous installment I briefly mentioned that filter can be seen in purely monadic terms, and monadic zeros are just the trick to seeing how. As a reminder, a filterable monad looks like this

```
class M[A] {
   def map[B](f: A => B):M[B] = ...
   def flatMap[B](f: A=> M[B]): M[B] = ...
   def filter(p: A=> Boolean): M[A] = ...
}
```

The filter method is completely described in one simple law

> FIL1. m filter p ≡ m flatMap {x => if(p(x)) unit(x) else mzero}

We create an anonymous function that takes x and either returns unit(x) or mzero depending on what the predicate says about x. This anonymous function is then used in a flatMap. Here are a couple of

results from this

1. m filter {x => true} ≡ m filter {x => true} // identity
2. m filter {x => true} ≡ m flatMap {x => if (true) unit(x) else mzero} // by FIL1
3. m filter {x => true} ≡ m flatMap {x => unit(x)} // by definition of if
4. FIL1a. m filter {x => true} ≡ m // by M1

So filtering with a constant "true" results in the same object. Conversely

1. m filter {x => false} ≡ m filter {x => false} // identity
2. m filter {x => false} ≡ m flatMap {x => if (false) unit(x) else mzero} // by FIL1
3. m filter {x => false} ≡ m flatMap {x => mzero} // by definition of if
4. FIL1b. m filter {x => false} ≡ mzero // by MZ1

Filtering with a constant false results in a monadic zero.

Side Effects

Throughout this article I've implicitly assumed no side effects. Let's revisit our second functor law

m map g map f ≡ m map {x => (f(g(x)) }

If m is a List with several elements, then the order of the operations will be different between the left and right side. On the left, g will be called for every element and then f will be called for every element. On the right, calls to f and g will be interleaved. If f and g have side effects like doing IO or modifying the state of other variables then the system might behave differently if somebody "refactors" one expression into the other.

The moral of the story is this: avoid side effects when defining or using map, flatMap, and filter. Stick to foreach for side effects. Its very definition is a big warning sign that reordering things might cause different behavior.

Speaking of which, where are the foreach laws? Well, given that foreach returns no result, the only real rule I can express in this notation is

m foreach f ≡ ()

Which would imply that foreach does nothing. In a purely functional sense that's true, it converts m and f into a void result. But foreach is meant to be used for side effects - it's an imperative construct.

Conclusion for Part 3

Up until now, I've focused on Option and List to let your intuition get a feel for monads. With this article you've finally seen what really makes a monad a monad. It turns out that the monad laws say nothing about collections; they're more general than that. It's just that the monad laws happen to apply very well to collections.

In part 4 I'm going to present a full grown adult elephant er monad that has nothing collection-like about it and is only a container if seen in the right light.

Here's the obligatory Scala to Haskell cheet sheet showing the more important laws

	Scala	Haskell
FM1	m map f ≡ m flatMap {x => unit(f(x))}	fmap f m ≡ m >>= \x -> return (f x)
M1	m flatMap unit ≡ m	m >>= return ≡ m
M2	unit(x) flatMap f ≡ f(x)	(return x) >>= f ≡ f x
M3	m flatMap g flatMap f ≡ m flatMap {x => g(x) flatMap f}	(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)
MZ1	mzero flatMap f ≡ mzero	mzero >>= f ≡ mzero
MZ2	m flatMap {x => mzero} ≡ mzero	m >>= (\x -> mzero) ≡ mzero
MZ3	mzero plus m ≡ m	mzero 'mplus' m ≡ m
MZ4	m plus mzero ≡ m	m 'mplus' mzero ≡ m
FIL1	m filter p ≡ m flatMap {x => if(p(x)) unit(x) else mzero}	mfilter p m ≡ m >>= (\x -> if p x then return x else mzero)

James Iry at 8:31 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