

4.7

Updated: 2021-03-08

# Monoids, Functors and Monads


© 2015 Robin Hillyard



Northeastern  
University

# Monoids, Functors and Monads

- These terms come from category theory. But really, their definitions as far as Scala is concerned are quite simple:
  - for instance, a **monoid** is just the type of thing we were getting at with our *List.x2* method (a.k.a *sum*)—where List’s underlying type must be a *monoid*:
  - something that can be operated on dyadic-ally and something which has a “zero” (identity) value:

```
def sum: A = {  
  @tailrec def inner(as: List[A], x: A): A = as match {  
    case Nil => x  
    case Cons(hd, tl) => inner(tl, x++hd)   
  }  
  inner(this, 0)  
}
```

- You don’t really need to remember all this stuff. I will review what’s important at the end.

# Monoids

- A **monoid** consists of the following:
  - a type  $A$
  - an associative binary operation,  $op$ , that takes two values of type  $A$  and combines them into one such that:

$$op(op(x,y),z) == op(x,op(y,z))$$

for any  $x: A, y: A, z: A$ .

- an “identity” (zero) value:  $identity: A$  that is an identity for  $op$ , i.e.  
 $op(x,identity) == x$  for any  $x: A$ .
- For example:

```
trait Monoid[A] {  
  def op(a1: A, a2: A): A  
  def identity: A  
}  
object Monoid {  
  val stringMonoid = new Monoid[String] {  
    def op(a1: String, a2: String) = a1 + a2  
    val identity = ""  
  }  
  def listMonoid[A] = new Monoid[List[A]] {  
    def op(a1: List[A], a2: List[A]) = a1 ++ a2  
    val identity = Nil  
  }  
}
```

# Monoids (2)

- There are two important functions on collections that we met briefly in Recursion: *foldRight* and *foldLeft*:

```
def foldLeft[A,B](z: B)(f1: (B, A) => B): B
def foldRight[A,B](z: B)(fr: (A, B) => B): B
```

- These methods are equivalent providing that *f1* and *fr* are associative—they simply work through a container in different directions—but only *foldLeft* is tail-recursive for *List* so we'll concentrate on that.
- Here's *foldLeft* implemented for our own *List* class from week 2:

```
case class Cons[A](h: A, t: List[A]) extends List[A] {
  def foldLeft[B](z: B)(f: (B, A) => B): B = t.foldLeft(f(z,h))(f)
}

case object Nil extends List[Nothing] {
  def foldLeft[B](z: B)(f: (B, Nothing) => B): B = z
}
```

Let's reimplement sum on List in terms of foldLeft...

```
def sum[B](xs: List[A]): B = xs.foldLeft(B.zero)(B.plus)
```

- Just one snag: *B.zero* and *B.plus* are not defined. But if, for any *B*, *zero* and *plus* were defined we'd be all set.

# Monoids (2a)

- We can do it with implicits (and we can keep  $B$  the same as  $A$ )...

```
def sum[A: Numeric](xs: List[A]): A = {  
  val an = implicitly[Numeric[A]]  
  xs.foldLeft(an.zero)(an.plus)  
}
```

# Monoids (3)

- Let's go back to our *stringMonoid* and create an *intMonoid* too (we'll also make *op* explicit as *plus*):

```
object Monoid {  
  val stringMonoid = new Monoid[String] {  
    def plus(a1: String, a2: String) = a1 + a2  
    val zero = ""  
  }  
  def intMonoid[A] = new Monoid[Int] {  
    def plus(a1: Int, a2: Int) = a1 + a2  
    val zero = 0  
  }  
}
```

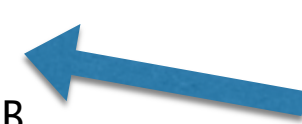
- So we would have:

```
def sum[B]: B = foldLeft(intMonoid.zero)(intMonoid.plus(_,_))
```

- Therefore, in general, a *Monoid* is something that is *foldable*. We could make this explicit by defining the following type constructor:

```
trait Foldable[F[_]] extends Functor[F] {  
  def foldLeft[A,B](z: B)(f: (B, A) => B): B  
  def foldRight[A,B](z: B)(f: (A, B) => B): B  
}  
def foldableList[A] = new Foldable[List] {  
  def map[A,B](as: List[A])(f: A => B): List[B] = as map f  
  def foldLeft[A,B](z: B)(f: (B, A) => B): B = ??? // tail recursive  
  def foldRight[A,B](z: B)(f: (A, B) => B): B = ??? // NOT tail recursive  
}
```

*Functor* defines *map*—we'll meet it next. “F[\_]” is a type constructor that takes an argument of a type, i.e. a higher-kinded type.



# Functors

- Functor is another term from category theory:
  - a *functor* is just a mapping between two categories (or types in Scala):

```
trait List[+A] { def map[B](f: A=>B): List[B] }  
trait Option[+A] { def map[B](f: A=>B): Option[B] }  
trait LazyList[+A] { def map[B](f: A=>B): LazyList[B] }  
etc. etc. etc.
```

- Notice anything?
  - All the definitions are identical—only the implementations differ.
  - Repetitive code like this is anathema to functional programmers!
  - Let's define a functor trait:

```
trait Functor[F[_]] {  
  def map[A, B](fa: F[A])(f: A => B): F[B]  
}  
object Functor {  
  def listFunctor = new Functor[List] {  
    def map[A,B](as: List[A])(f: A => B): List[B] = as map f  
  }  
}
```

# Monads

- Monad also comes from category theory. You've heard me mention monads already:
- A *Monad* isa *Functor* (i.e. it implements *map*):

```
trait List[+A] { def map[B](f: A=>B): List[B] }  
trait Option[+A] { def map[B](f: A=>B): Option[B] }  
trait LazyList[+A] { def map[B](f: A=>B): LazyList[B] }  
etc. etc. etc.
```

- Remember how *map2* was basically identical for several different container types? These were all *monads*! So let's define *map2* once and for all...

```
trait Monad[F[_]] extends Functor[F] {  
  def map2[A,B,C](ma: F[A], mb: F[B])(f: (A,B)=>C): F[C] =  
    flatMap(ma)(a => map(mb)(b => f(a,b)))  
}
```

- But *flatMap* isn't defined :( Except that since this definition is the canonical definition of *map2* and we are essentially defining monad as things which support *map2*, ergo *Monad* must define *flatMap* too:

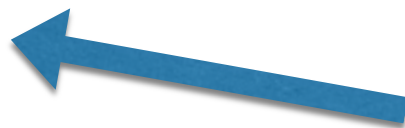
```
trait Monad[F[_]] extends Functor[F] {  
  def flatMap[A,B](ma: F[A])(f: A=>F[B]): F[B]  
  def map2[A,B,C](ma: F[A], mb: F[B])(f: (A,B)=>C): F[C] =  
    flatMap(ma)(a => map(mb)(b => f(a,b)))  
}
```



# Monads (2)

- Actually, there's a certain flexibility in exactly how we define the primitive methods of *Monad*. Let's think about *map*. If we had a method *unit* which took a value and simply wrapped it (like a single-element *List*) then we could actually write *map* in terms of *flatMap*:

```
trait Monad[F[_]] extends Functor[F] {  
  def unit[A](a: => A): F[A] // abstract  
  def flatMap[A,B](ma: F[A])(f: A=>F[B]): F[B] // abstract  
  def map[A,B](ma: F[A])(f: A=>B): F[B] = flatMap(ma)(a => unit(f(a)))  
  def map2[A,B,C](ma: F[A], mb: F[B])(f: (A,B)=>C): F[C] =  
    flatMap(ma)(a => map(mb)(b => f(a,b)))  
}
```



In this definition of *Monad*, *unit* and *flatMap* are abstract methods—they must be defined by implementers of *Monad*. The other two methods are concrete methods defined in terms of the first two.

- So, *unit+flatMap* is one possible form.
- But why do we care about all this stuff? Because of composability.

# Monads (3)

- Articles which (try to) explain monads:
  - <https://medium.com/@lettier/your-easy-guide-to-monads-applicatives-functors-862048d61610>
  - <https://medium.com/@sinisalouc/demystifying-the-monad-in-scala-cc716bb6f534>
  - <https://medium.com/@yuriigorbylov/monads-and-why-do-they-matter-9a285862e8b4>
  - <https://medium.com/zendesk-engineering/dont-fear-the-monad-f424260f29f6>
  - <https://medium.com/@evinsellin/teaching-monads-slightly-differently-2af62c4af8ce>

# Review: what do you need to remember?

- A type that implements *map* is a **functor**.
- A **monad** is a functor but not all functors are monads.
- For-comprehensions work on *monads*.
- The *reduce* method works when the underlying type is a **monoid**.
- Any type which defines both *map* and *flatMap* is a **monad**—you don't have to declare it explicitly as a monad!
- *map* can be defined as *flatMap(a => unit(f(a)))*