Reading: Chapter 10-11 of [Chiusano and Bjarnason 2014]

Functional Design Patterns

Lecture 090 of Advanced Programming

November 1, 2018

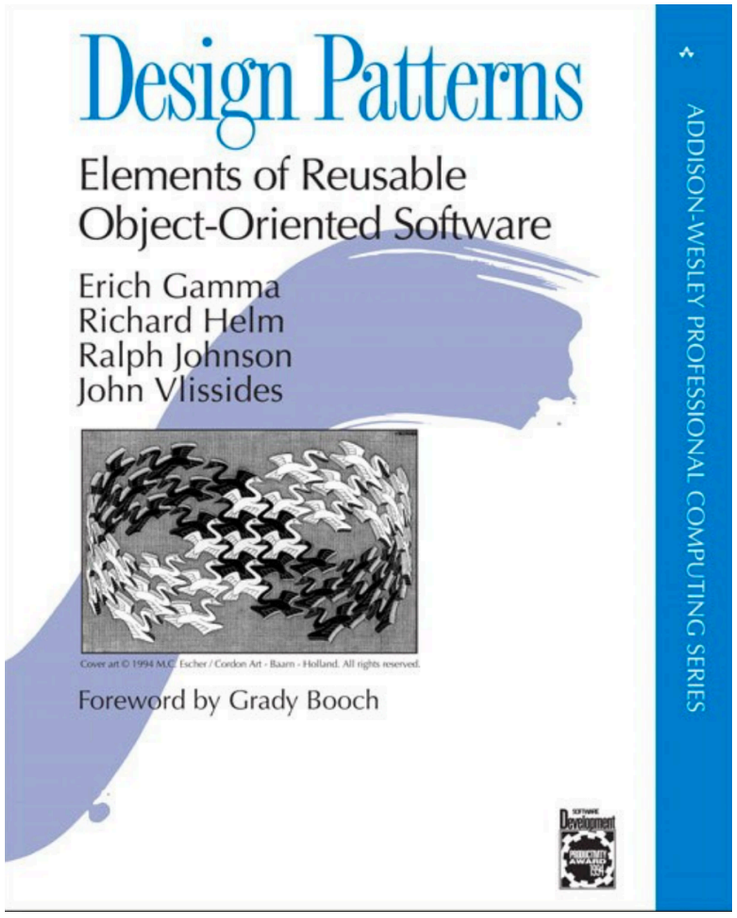
Andrzej Wasowski & Zhoulai Fu

IT University of Copenhagen

Good coders code; great coders reuse

Menu for Today

- Monoid
- Foldable
- Functor
- Monad



- These are FP concepts in the first place
- In this lecture, we shall focus on concepts, instead of code
- We do not implement; we prove

You may have already known these

- foldLeft
- foldRight
- map
- flatMap

Parameterized Types

- Type A[B] Type C[D, E]
- trait F[G[_]]
- class M[I, J[_], K]
  - e.g with a type error: new M [String, List[Int], Date]()

Examples taken from somewhere

Monoid

A monoid is a set that has a **closed, associative,** binary operation and has an **identity** element.

Associativity

```
scala> (1+2)+3
res60: Int = 6

scala> 1+(2+3)
res61: Int = 6
```

Closed binary operation

- Integers are closed with +,-,* but not /
- Positive number is not closed with subtraction

Associativity

```
scala> (2*3)*4
res64: Int = 24

scala> 2*(3*4)
res65: Int = 24
```

Associativity

```
scala> List(1,2)++(List(3,4)++List(5))
res75: List[Int] = List(1, 2, 3, 4, 5)
```

```
scala> (List(1,2)++List(3,4))++List(5)
res76: List[Int] = List(1, 2, 3, 4, 5)
```

Not Associative

```
scala> (8-2)-4
res77: Int = 2


scala> 8-(2-4)
res78: Int = 10
```

Associativity formally defined

Given a set S, a binary operator $\otimes$ is said to be associative if for any three elements a, b, c of S,

$$(a \otimes b) \otimes c = a \otimes (b \otimes c)$$

Identity

```
scala> 42+0
res79: Int = 42

scala> 0+42
res80: Int = 42
```

Identity

```
scala> 42*1
res68: Int = 42

scala> 1*42
res69: Int = 42
```

Identity

```
scala> List(2,3)++Nil
res72: List[Int] = List(2, 3)

scala> Nil++List(2,3)
res73: List[Int] = List(2, 3)
```

Identity formally defined

Given a set S and a binary operator $\otimes$, an element $id \in S$ is said to be an identity if

$$id \otimes s = s \otimes id = s$$

holds for any $s \in S$.

Quiz 1: To recognize

Type	B-Operation	Closed?	Associative?	Idenity?	So, is it a Mnoid
Int	Addition				
Int	Multiplication				
Int	Substraction				
Int	Division				
Int	Max				
Float	Addition				
Boolean	OR				
Boolean	AND				
String	Concatenation				

Quiz 2: To prove

Let

- $Digits = \{0, 1, ...9\}$, and
- binary operator $a \oplus b = (a + b) \mod 10$.

Prove $(Digits, \oplus, 0)$ is a monoid.

Why monoid matters?

- Monoid is about abstraction
- Abstraction makes it easier to reason about code
- Abstraction makes it easier to reuse code

Some Practical benefits of using monoid

Well behaved folds over sequences!

```
trait Seq[A] { …
  def fold(a: A)(f: (A, A)): A
}
fold(Monoid[A].id)(_ |+| _)
```

With a monoid, no need to guarantee order.

```
(((((a |+| b) |+| c) |+| d) |+| e) |+| f) |+| g
```

```
a |+| (b |+| (c |+| (d |+| (e |+| (f |+| g)))))
```

```
(a |+| b |+| c |+| d) |+| (e |+| f |+| g)
```

```
xs.par.fold(Monoid[A].id)(_ |+| _)
```

From: Life after Monoids

A Monoid Trait in Scala

```
trait Monoid[A] {
  def id: A
  def op(x: A, y: A): A
}

//Examples:

val intMonoid=new Monoid[Int] {
  def id=0
  def op(x:Int,y:Int):Int = x+y
}
```

Foldable

Almost all structures implemented in this course are foldable

- List(1,2,3).foldRight(1)(_*_)
- IndexSeq
- Stream
- Tree

The Foldable Trait In Scala

```
trait Foldable[F[_]] {
  def foldRight[A,B](as: F[A])(z: B)(f: (A,B) => B): B
  def foldLeft[A,B](as: F[A])(z: B)(f: (B,A) => B): B
  def foldMap[A,B](as: F[A])(f: A => B)(mb: Monoid[B]): B
  def concatenate[A](as: F[A])(m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)
}
```

- F[_] is the type constructor with one argument
- FoldLeft, FoldRight and FoldMap can be implemented in terms of each other, but that might not be the most efficient implementation.
-

Functor

The pattern

```
def map[A,B](ga: Gen[A])(f: A => B): Gen[B]

def map[A,B](pa: Parser[A])(f: A => B): Parser[B]

def map[A,B](oa: Option[A])(f: A => B): Option[A]
```

Functor

- When a value is wrapped in a structure, you cannot apply a normal function to that value. You need a *map* that knows how to apply functions to values that are wrapped in a structure.

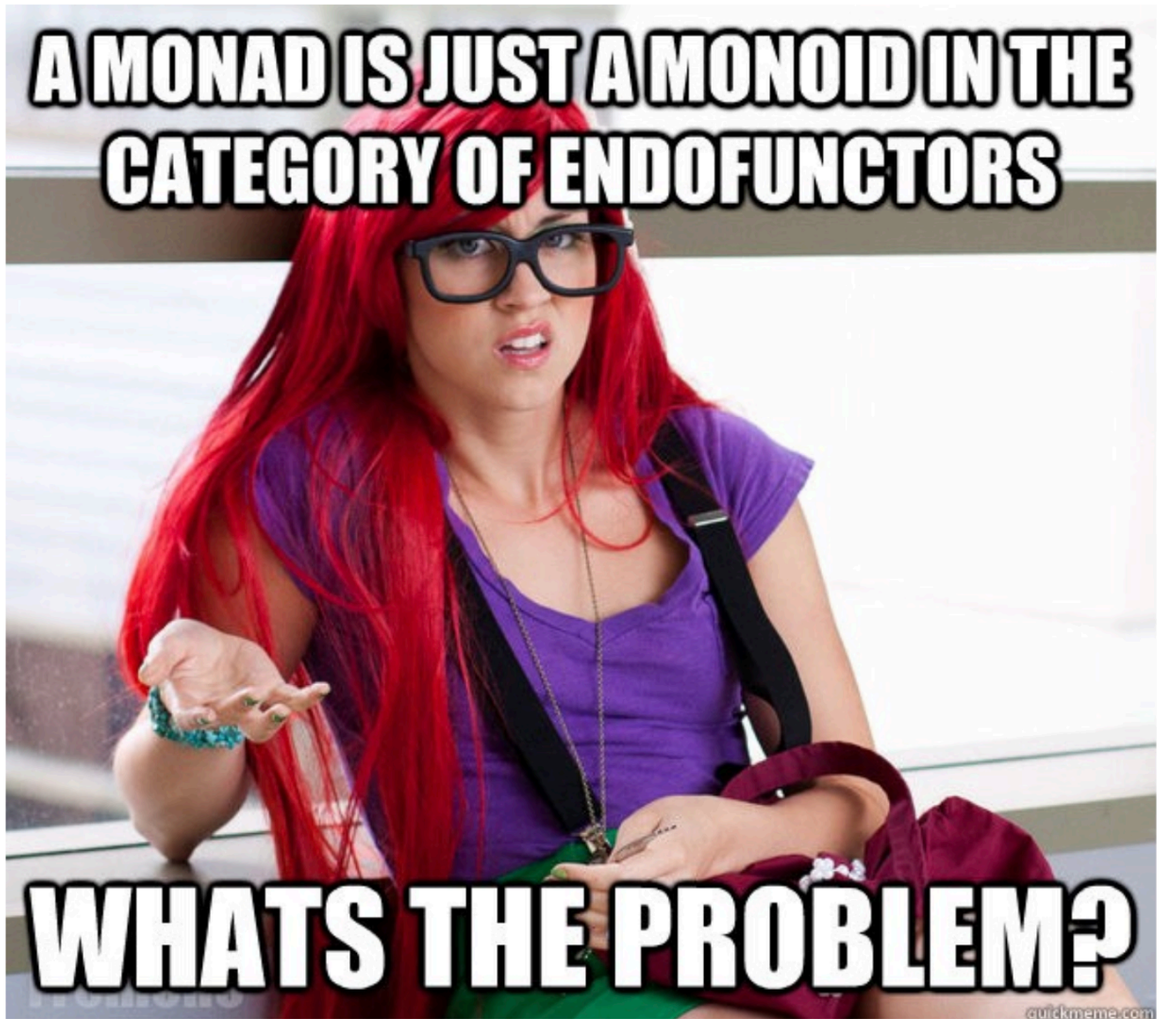
From: *Functors, Applicatives, and Monads*

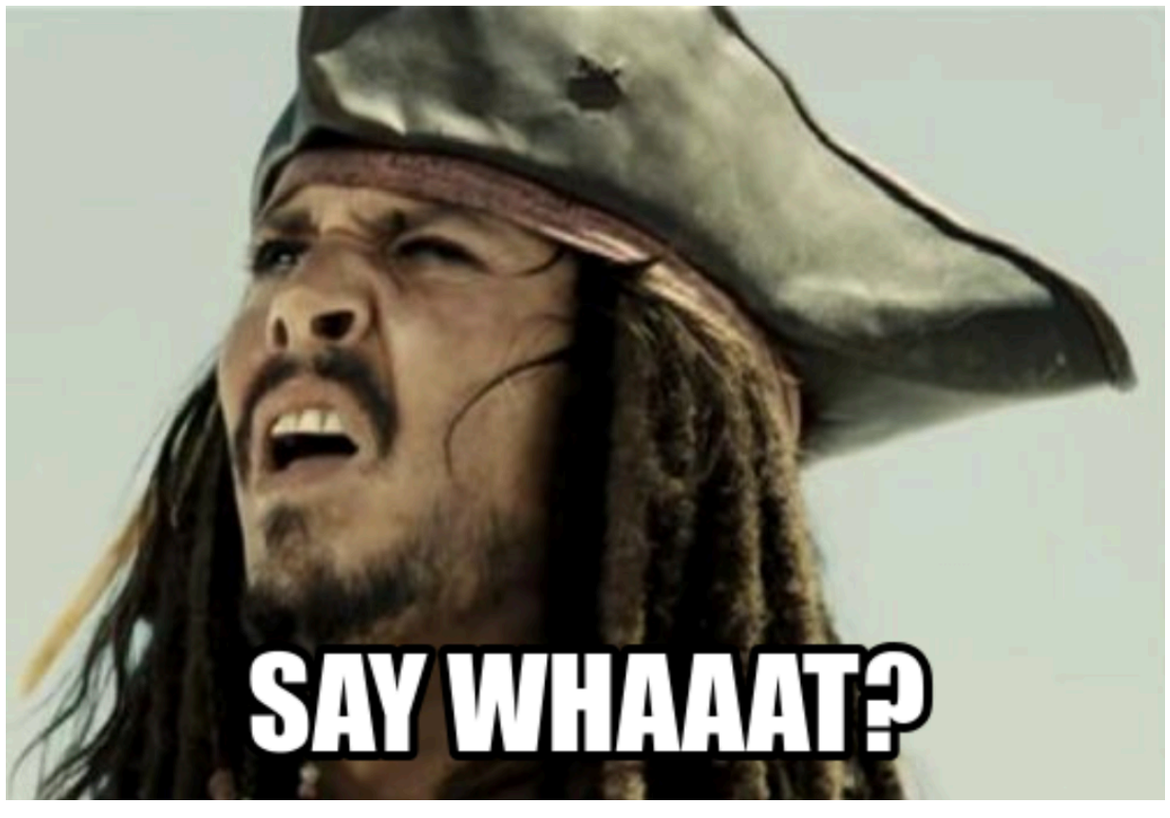
A Functor Trait In Scala

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

- Map is parameterized on the type constructor F, much like we did with Foldable.

Monad


A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTORS
WHATS THE PROBLEM?
quickmeme.com


SAY WHAAAT?

A monad is, first of all, a wrapper

- def unit[A] (x: A): Monad[A]
- Understand the type parameter as a label sticker on the wrapper, saying what kind of object you have inside

It is a FlatMappable wrapper

```
scala> List(2,3).map(x=>List("Hello",x))
res12: List[List[Any]] = List(List(Hello, 2), List(Hello, 3))

scala> List(2,3).map(x=>List("Hello",x)).flatten
res13: List[Any] = List(Hello, 2, Hello, 3)

scala> List(2,3).flatMap(x=>List("Hello",x))
res14: List[Any] = List(Hello, 2, Hello, 3)
```

- def flatMap[B] (f: A => Monad[B]): Monad[B]
- flatMap = first map, then flatten
- To flatMap an object of type M[A], you need to provide a function of type A-> M[B], FlatMap then uses this function to transform A into M[B], resulting in a M[M[B]], and then it will flatten the whole thing into M[B].

```
          map with A => M[B]                    flatten
M[A]  ------------------------->  M[M[B]]  ------------> M[B]
```

A Monad Trait in Scala

Listing 11.4 Creating our `Monad` trait

```
trait Monad[F[_]] extends Functor[F] {
  def unit[A](a: => A): F[A]
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]

  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)))
}
```

Since `Monad` provides a default implementation of `map`, it can extend `Functor`. All monads are functors, but not all functors are monads.

- Monad picks unit and flatMap as a minimal setting
- Other functions, such as “map”, or “map2” can be defined via flatMap
- You are going to see many monads in APIs. They may not call themselves monads, but knowing that they are actually monads make you life easier

Conclusions

- Monoid = Closed binary operation with identity
- Foldable
- Functor = Mappable
- Monad = FlatMappable wrapper
- Recognizing these patterns —- they are everywhere, give us a new power to understand code, reuse it and to reason about it