Patterns in Functional Programming

Part 2

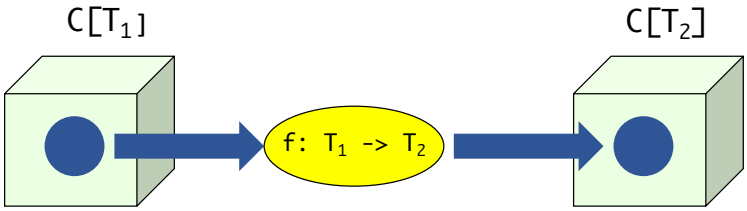
Functional Programming and Mathematics

- Functional programming derives many of its patterns/idioms directly from mathematics
  - Category theory
- Monoid
  - Representation of binary operations
- Functor
  - Processing data in enclosed in some container
- Monad
  - Providing a way of processing in a pipeline
  - A mechanism for dealing with "effects"

Computation on Containers

- Many types can be described as "container" types
  - Collections
  - Option[T], Try[T], Future[T], …
- These types expose several common patterns of computation

`map`



Computation on Containers

- Many types can be described as "container" types
  - Collections
  - Option[T], Try[T], Future[T], …
- These types expose several common patterns of computation

`map`

```
scala> val l1:List[Int] = List(1, 2, 3)
l1: List[Int] = List(1, 2, 3)

scala> l1 map ( el => el + 5 )
res118: List[Int] = List(6, 7, 8)
```

Implementing map on Containers

- Use structural recursion
- Simple container

```
sealed trait Box[A] {

  def fold[B](empty: B)(full: A => B) = // as seen earlier

  def map[B](f: A => B): Box[B] =
    this match {
      case Empty() => Empty[B]()
      case Full(v) => Full(f(v))
    }
}
...
```

```
scala> val e: Box[Int] = Empty()
e: Box[Int] = Empty()

scala> val f: Box[Int] = Full(2)
f: Box[Int] = Full(2)

scala> e.map( x => x * 2 )
res119: Box[Int] = Empty()

scala> f.map( x => x * 2 )
res120: Box[Int] = Full(4)
```


Implementing map on Containers

- Use structural recursion
- Recursively defined container

```
sealed trait MyList[A] {

  def fold[B] ( end: B ) ( f: (A, B) => B ): B = ???

  def map[B] ( f: A => B ) : MyList[B] =
    this match {
      case End() => End[B]()
      case Cons(hd, tl) => Cons ( f(hd), tl.map(f) )
    }
}
```

Implementing map on Containers

- Use structural recursion
- Recursively defined container

```
scala> val l1: MyList[Int] = Cons(1, Cons(3, Cons(5, End()) ) )
l1: MyList[Int] = Cons(1,Cons(3,Cons(5,End())))

scala> val s1: MyList[String] = Cons("Hello", Cons("world", End()) )
s1: MyList[String] = Cons(Hello,Cons(world,End()))

scala> l1.map( x=> x * x )
res121: MyList[Int] = Cons(1,Cons(9,Cons(25,End())))

scala> s1.map( _.length )
res122: MyList[Int] = Cons(5,Cons(5,End()))

scala> val e1: MyList[Int] = End()
e1: MyList[Int] = End()

scala> e1.map( x => x * x )
res123: MyList[Int] = End()
```


Functor

- Category Theory defines abstraction over the functionality provided by map
- Functor
- May be defined as a typeclass

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

Instance of container type

Function to be applied

Functor Examples

- We can define Functors based on existing types that satisfy the requirements

```
object SeqF extends Functor[Seq] {
  def map[A,B](seq: Seq[A])(f: A=>B): Seq[B] = seq map f
}
```

- Functor object now allows its operations to be applied to supplied objects

```
scala> val s1 = List("Foo", "Bar")
s1: List[String] = List(Foo, Bar)

scala> SeqF.map(s1)( _.toUpperCase )
res125: Seq[String] = List(FOO, BAR)
```

```
scala> val l1 = List(1,3,5 )
l1: List[Int] = List(1, 3, 5)

scala> val e1: List[Int] = Nil
e1: List[Int] = List()

scala> SeqF.map(e1)( _ * 4 )
res126: Seq[Int] = List()
```


Functor Examples

- Container types do not need to define map function themselves

```
object BoxF extends Functor[Box] {
  def map[A,B](b: Box[A])(f: A=>B): Box[B] = b match {
    case Empty() => Empty[B]()
    case Full(v) => Full(f(v))
  }
}
```

Functor Examples

- Container types do not need to define map function themselves

```
object BoxF extends Functor[Box] {
  def map[A,B](b: Box[A])(f: A=>B): Box[B] = b match {
    case Empty() => Empty[B]()
    case Full(v) => Full(f(v))
  }
}
```

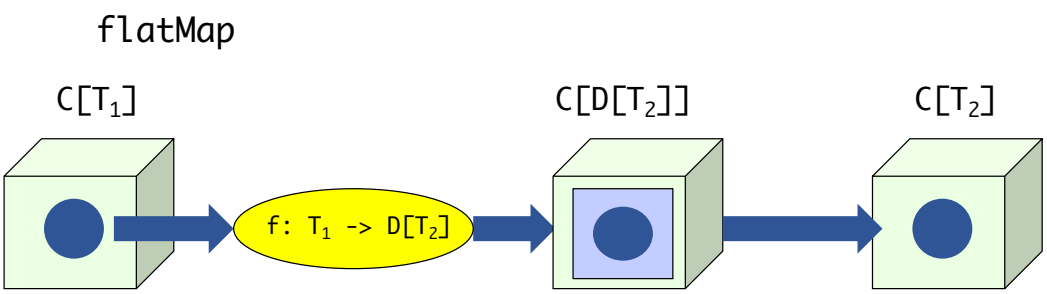
- Now map function is available for Box

```
scala> val sb: Box[String] = Full("Hello")
sb: Box[String] = Full(Hello)

scala> BoxF.map(sb)( _.length )
res130: Box[Int] = Full(5)
```

```
scala> val b: Box[Int] = Full(5)
b: Box[Int] = Full(5)

scala> val eb: Box[Int] = Empty()
eb: Box[Int] = Empty()

scala> BoxF.map(b)( _ * 2 )
res128: Box[Int] = Full(10)

scala> BoxF.map(eb)( _ * 2 )
res129: Box[Int] = Empty()
```


Computation on Containers: 2

- Sometimes the function applied to element in container itself returns an instance of a container type
- Requirement is to remove the value from this inner container



Computation on Containers: 2

- Sometimes the function applied to element in container itself returns an instance of a container type
- Requirement is to remove the value from this inner container

`flatMap`

```
scala> val l1 = List( 1, 3, 5 )
l1: List[Int] = List(1, 3, 5)

scala> l1.flatMap ( el => List( el - 1, el, el + 1 ) )
res131: List[Int] = List(0, 1, 2, 2, 3, 4, 4, 5, 6)
```


Implementing flatMap

- Structural recursion
- Focus on simple container
  - Recursive containers more complex

```
sealed trait Box[A] {

  def fold[B](empty: B)(full: A => B) = // as before

  def map[B](f: A => B): Box[B] =       // as before

  def flatMap[B](f: A => Box[B] ) : Box[B] =
    this match {
    case Empty() => Empty[B]()
    case Full(v) => f(v)
  }
}
...
```

Implementing flatMap

- Structural recursion
- Focus on simple container
  - Recursive containers more complex

```
scala> val eb: Box[Int] = Empty()
eb: Box[Int] = Empty()

scala> val fb: Box[Int] = Full(100)
fb: Box[Int] = Full(100)

scala> eb.flatMap ( x => Full(x * 2) )
res141: Box[Int] = Empty()

scala> fb.flatMap ( x => Full(x * 2) )
res142: Box[Int] = Full(200)
```

```
scala> eb.map ( x => x * 2 )
res139: Box[Int] = Empty()

scala> fb.map ( x => x * 2 )
res140: Box[Int] = Full(200)
```


Monad

- Abstraction concept from Category Theory
- Abstracts over flatMap operation
  - Sometimes referred to as bind
- Formal definition also requires point operation
  - Create instance of the type from a value
  - Sometimes referred to as unit
- Examples of types that meet these requirements
  - List
  - Set
  - Option

Monad

- Category Theory requires monads to satisfy certain laws
- Associativity

```
m flatMap f flatMap g == m flatMap ( x => f(x) flatMap g )
```

- Left Unit

```
unit(x) flatMap f == f(x)
```

- Left Unit

```
m flatMap unit == m
```

- Not all Scala types considered to be monads satisfy all of these
  - Try[T]

Defining Monad

- Scala allows monad to be defined as a typeclass

```
trait Monad[ M[_] ] {
  def flatMap[A, B](fa: M[A])(f: A => M[B]) : M[B]
  def unit[A](a: => A): M[A]
}
```

- Like Functor[F[_]] typeclass

Monad Examples

- Define monad instances based on existing "monadic" types

```
object SeqM extends Monad[Seq] {
  def flatMap[A,B](seq: Seq[A])(f: A => Seq[B]): Seq[B] = seq flatMap f

  def unit[A]( v: => A ): Seq[A] = Seq(v)
}
```

```
scala> val l1 = List(1,3,5)
l1: List[Int] = List(1, 3, 5)

scala> val el:List[Int] = Nil
el: List[Int] = List()

scala> SeqM.flatMap(l1)( i => 1 to i )
res146: Seq[Int] = List(1, 1, 2, 3, 1, 2, 3, 4, 5)

scala> SeqM.flatMap(e1)( i => 1 to i )
res147: Seq[Int] = List()
```

```
scala> val sl = List("Hello", "world")
sl: List[String] = List(Hello, world)

scala> SeqM.flatMap(sl)( _.toSeq )
res145: Seq[Char] =
    List(H, e, l, l, o, w, o, r, l, d)
```


Monad Examples

- flatMap method can be defined within the Monad object

```
object BoxM extends Monad[Box] {
  def flatMap[A, B](b: Box[A])(f: A => Box[B] ) : Box[B] =
    b match {
      case Empty() => Empty[B]()
      case Full(v) => f(v)
    }
  def unit[A]( v: => A): Box[A] = Full(v)
}
```

Monad Examples

- flatMap method can be defined within the Monad object

```
scala> val b: Box[Int] = Full(4)
b: Box[Int] = Full(4)

scala> val eb: Box[Int] = Empty()
eb: Box[Int] = Empty()

scala> val sb: Box[String] = Full("Foobar")
sb: Box[String] = Full(Foobar)
```

```
scala> BoxM.flatMap(b)( i => BoxM.unit(i * 2) )
res149: Box[Int] = Full(8)

scala> BoxM.flatMap(eb)( i => BoxM.unit(i * 2) )
res150: Box[Int] = Empty()

scala> BoxM.flatMap(sb)( i => BoxM.unit(i.toUpperCase) )
res151: Box[String] = Full(FOOBAR)
```


Monads and Functors

- Map function (from Functor) can be defined in terms of flatMap and unit methods

```
trait Monad[ M[_] ] {
  def unit[A](a: => A): M[A]
  def flatMap[A, B](fa: M[A])(f: A => M[B]) : M[B]

  def map[A,B](ma: M[A])(f: A => B) = flatMap(ma)(a => unit(f(a)))
}
```

```
trait Monad[ M[_] ] extends Functor[ M ] {
        ...
}
```

- Every Monad is a Functor
  - Not true in reverse

Working With Monads

- Monads allow for sequential processing of values *in some context*
  - Context defined by monad type
  - Value type defined by element type

```
def optAdd ( i: Opt[Int], j: Opt[Int] ): Opt[Int] = {
  for (
    i1 <- i;
    i2 <- j
  ) yield ( i1 + i2 )
}
```

```
def mkOptInt (s: String):Option[Int] = {
  try {
    Some(s.toInt)
  } catch {
    case e: Exception => None
  }
}
```

```
scala> optAdd( mkOptInt("1"), mkOptInt("2") )
res96: Option[Int] = Some(3)

scala> optAdd( mkOptInt("1"), mkOptInt("ouch") )
res95: Option[Int] = None
```


Working With Monads

- We can generalise this behaviour

```
def tryAdd ( i: Try[Int], j: Try[Int] ): Try[Int] =
       i flatMap ( i1 => j map ( i2 => i1 + i2 ) )
```

```
def optAdd ( i: Option[Int], j: Option[Int] ): Option[Int] =
       i flatMap ( i1 => j map ( i2 => i1 + i2 ) )
```

- Derive generic method that can be added to Monad typeclass

```
def map2[A,B,C] (ma: M[A], mb: M[B]) (f: (A,B) => C) : M[C] =
    flatMap(ma) ( a => map (mb) ( b => f(a,b) ) )
```

Generalised Monad Type Class

```
trait Monad[ M[_] ] extends Functor[ M ] {

  def flatMap[A, B](fa: M[A])(f: A => M[B]): M[B]

  def unit[A](a: => A): M[A]

  def map [A, B]( ma: M[A] )(f: A => B): M[B] =
    flatMap(ma)( a => unit(f(a)) )

  def map2[A,B,C] (ma: M[A], mb: M[B]) (f: (A,B) => C) : M[C] =
    flatMap(ma) ( a => map (mb) ( b => f(a,b) ) )
}
```

- Monad instance need only define unit and flatMap

Monad Combinators

- Monads do not always compose easily
  - No simple abstract means of achieving this in the general case
- It is possible to define *Monad Combinator* functions
  - Allow monads to be generated/transformed from other monads
- Two common and useful examples
- sequence
  - Transform a list of Monadic values into a Monadic List of values
- traverse
  - Apply monadic function to list of input values, then apply sequence to the resulting List

Monad Combinators

- Consider the Box[A] from before

```
sealed trait Box[A]

final case class Empty[A]() extends Box[A]
final case class Full[A](value: A) extends Box[A]
```

- Create instance of Monad typeclass for this

```
object BoxM extends Monad[Box] {
  def unit[A](v: => A): Box[A] = Full(v)
  def flatMap[A,B](b: Box[A])( f: A => Box[B] ): Box[B] =
    b match {
      case Empty() => Empty[B]()
      case Full(v) => f(v)
    }
}
```


Monad Combinators

- The combinators can be used for this Monad

```
scala> val l: List[Box[Int]] = List( Full(1), Full(2), Full(3) )
l: List[Box[Int]] = List(Full(1), Full(2), Full(3))

scala> BoxM.sequence(l)
res112: Box[List[Int]] = Full(List(1, 2, 3))
```

```
scala> def mkBox(i: Int): Box[Int] = Full(i)
mkBox: (i: Int)Box[Int]

scala> BoxM.traverse( List(1,2,3) )( mkBox(_) )
res113: Box[List[Int]] = Full(List(1, 2, 3))
```

Monad Combinators

- The concrete implementations of the combinators are may be include in the typeclass

```
trait Monad[ M[_] ] extends Functor[ M ] {
  def flatMap[A, B](fa: M[A])(f: A => M[B]): M[B]
  def unit[A](a: => A): M[A]
  def map [A, B]( ma: M[A] )(f: A => B): M[B] =
    flatMap(ma)( a => unit(f(a)) )
  def map2[A,B,C] (ma: M[A], mb: M[B]) (f: (A,B) => C) : M[C] =
    flatMap(ma) ( a => map (mb) ( b => f(a,b) ) )
  def traverse[A,B](la: List[A])(f: A => M[B]): M[List[B]] =
    la.foldRight( unit(List[B]()) ) ( (a, mlb) => map2(f(a), mlb)( _ :: _ ) )
  def sequence[A](lma: List[M[A]]): M[List[A]] =
    traverse(lma)(ma => ma)
}
```

Abstract, must be defined in M

Applicative

- A further FP pattern
  - Extension of Functor
- Useful alternative approach to building combinators such as traverse, sequence, …
  - With some subtle semantic differences
- Consider the traverse combinator function from before:

```
def traverse[A,B](as: List[A])(f: A => F[B]): F[List[B]] =
    as.foldRight( unit(List[B]())) ( (a, fbs) => map2(f(a), fbs)(_ :: _) )
```

In Monads, map2 is built using flatMap

Applicative

- Monad is built from primitives unit and flatMap
  - map2 derived from these
- Applicative Functor is built from primitives unit and map2
  - map2 is used to derive a further basic function, apply

```
trait Applicative[ F[_] ] extends Functor[ F ] {
  def unit[A](a : => A): F[A]
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[C]
  def apply[A,B](fab: F[A => B])(fa:F[A]): F[B] =
    map2(fab, fa)(_(_))
  def map[A,B](fa: F[A])(f: A=>B): F[B] =
    apply(unit(f))(fa)

}
```

map2 can be derived from apply if necessary

Applicative

- Combinators can be added

```
trait Applicative[ F[_] ] extends Functor[ F ] {
  def unit[A](a : => A): F[A]
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A,B) => C): F[C]
  def apply[A,B](fab: F[A => B])(fa:F[A]): F[B] =
    map2(fab, fa)(_(_))
  def map[A,B](fa: F[A])(f: A=>B): F[B] =
    apply(unit(f))(fa)
  def traverse[A,B](as: List[A])(f: A => F[B]): F[List[B]] =
    as.foldRight(unit(List[B]()))((a, fbs) => map2(f(a), fbs)(_ :: _))
  def sequence[A](fas: List[F[A]]): F[List[A]] =
    traverse(fas)(fa => fa)
}
```

Same implementation as in Monad example

Applicative Example

- Use with Box type

```
object BoxA extends Applicative[Box] {
  def unit[A](v: => A): Box[A] = Full(v)
  def map2[A,B,C](fa: Box[A], fb: Box[B])(f: (A,B) => C) =
    (fa, fb) match {
      case (Full(a), Full(b)) => Full ( f(a,b) )
      case _ => Empty()
    }
}
```

```
scala> BoxA.traverse(List(1,2,3))(mkBox(_))
res2: Box[List[Int]] = Full(List(1, 2, 3))
```


Applicative and Monad

- Both appear to produce the same results in traverse…
- What is the difference between them??
- Monad based solution is based on flatMap
  - Imposes a strict ordering on the computation
  - Each step depends on the result of the previous one
  - Any "error" or failure causes fail-fast behaviour
- Applicative based solution based on map2/apply
  - No requirement for ordering
  - Each step is independent of the other(s)
  - Error or failure details can be accumulated if required