# Patterns in Functional Programming

# 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"

# Binary Operations

- Common style of functions
  - Two arguments, one result
  - All same type

- Examples
  - Integer addition
  - String concatenation
  - List contatenation

- Already seen as function argument to fold method

# Binary Operations

- Category Theory defines an abstraction for binary operations
  - The Monoid
- Two parts
  - Operation append, with signature ( A, A ) => A
  - Single element zero, with type A
- A monoid is required to satisfy two laws
  - append operation is associative
  - zero is the identity of append

# Defining Monoid

- In Scala, monoid can be defined as a trait

```
trait Monoid [A] {
  def append( f1: A, f2: A ): A
  def zero: A
}
```

# Defining Monoid

- In Scala, monoid can be defined as a trait

```
trait Monoid [A] {
  def append( f1: A, f2: A ): A
  def zero: A
}
```

- Example: Integer addition

```
object IntAddMonoid extends Monoid[Int] {
  def append( i: Int, j: Int ): Int = i + j
  val zero = 0
}
```

```
scala> IntAddMonoid.append( 4, 5 )
res109: Int = 9

scala> IntAddMonoid.zero
res111: Int = 0
```

# Defining Monoid

- Is integer addition a monoid?
- Associative?

```
scala> IntAddMonoid.append( 1, IntAddMonoid.append(2, 3) )
res112: Int = 6

scala>  IntAddMonoid.append( IntAddMonoid.append(1, 2), 3 )
res114: Int = 6
```

# Defining Monoid

- Is integer addition a monoid?
- Associative?

```
scala> IntAddMonoid.append( 1, IntAddMonoid.append(2, 3) )
res112: Int = 6

scala>  IntAddMonoid.append( IntAddMonoid.append(1, 2), 3 )
res114: Int = 6
```

- Identity?

```
scala> IntAddMonoid.append( IntAddMonoid.zero, 2 )
res115: Int = 2
```

# Using Monoid

- Can be used with fold... operations
  - Identity provides the base case
  - Append provides the "inductive" case

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

scala> l1.foldLeft(IntAddMonoid.zero)(IntAddMonoid.append)
res116: Int = 6
```

# Using Monoid

- Can be used with fold… operations
  - Identity provides the base case
  - Append provides the "inductive" case

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

scala> l1.foldLeft(IntAddMonoid.zero)(IntAddMonoid.append)
res116: Int = 6
```

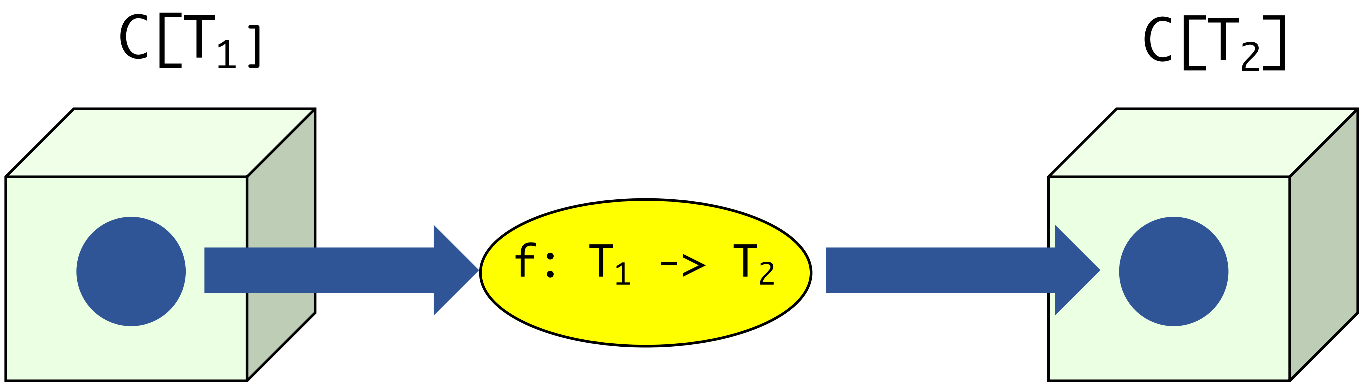
- Associative law means foldRight will yield the same result

```
scala> l1.foldRight(IntAddMonoid.zero)(IntAddMonoid.append)
res117: Int = 6
```

# 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 using trait

```
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

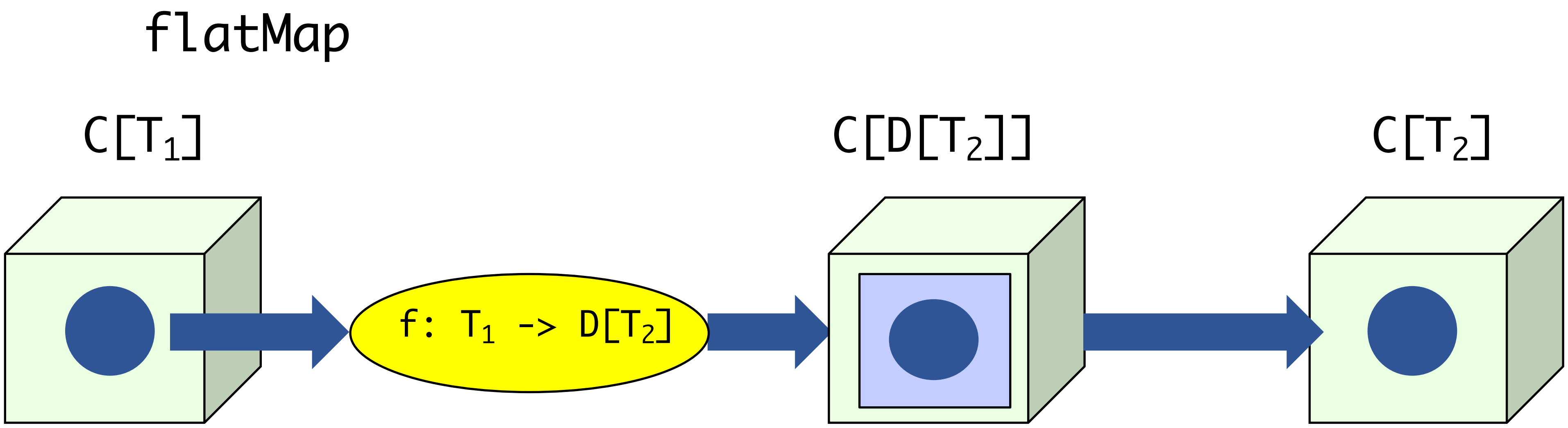
```
scala> val sb: Box[String] = Full("Hello")
sb: Box[String] = Full(Hello)

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 trait

```
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[_] ] trait

# 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)
```