Patterns in Functional Programming

Part 1

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

- Recall the fold operation derived for container types
  - E.g. List[Int]

```
scala> def sum(l: List[Int]): Int = l.foldLeft(0)(_ + _)
sum: (l: List[Int])Int

scala> sum(List(1,2,3))
res49: Int = 6
```

- We would like to generalise this over other numeric types

```
scala> def sum(l: List[BigInt]): BigInt = l.foldLeft(BigInt(0))(_ + _)
sum: (l: List[BigInt])BigInt

scala> sum(List(BigInt(1), BigInt(2)))
res57: BigInt = 3
```


Binary Operations

- Scala's Numeric typeclass makes this more concise

```
scala> def sum[A : Numeric](list: List[A]): A = {
     |   val ev = implicitly[Numeric[A]]
     |   list.foldLeft(ev.zero)((sum, e) => ev.plus(sum,e))
     | }
sum: [A](list: List[A])(implicit evidence$1: Numeric[A])A

scala> sum( List(1,2,3) )
res58: Int = 6

scala> sum( List(1.0, 2.0, 3.0) )
res59: Double = 6.0

scala> sum( List(BigInt(1), BigInt(2), BigInt(3)) )
res60: scala.math.BigInt = 6
```

Binary Operations

- But…
- Perhaps we would like to define a similar function to operate on Strings
  - String concatenation can be viewed as a form of "addition"
  - Empty string can behave as a "zero" value

```
scala> def sum(l: List[String]) = l.foldLeft("")(_ + _)
sum: (l: List[String])String

scala> sum(List("one", "two"))
res61: String = onetwo
```

- But this will not work beside the Numeric typeclass

Introducing the Monoid

- The calls to l.foldLeft all have a similar "shape"
  - Despite the element types being different

```
l.foldLeft (0)         (_ + _)
l.foldLeft (BigInt(0)) (_ + _)
l.foldLeft ("")        (_ + _)
```

Initial, or "empty" value

Function to accumulate result

Introducing the Monoid

- Category Theory defines an abstraction for such 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 typeclass

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

Defining Monoid

- In Scala, monoid can be defined as a typeclass

```
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
```


Why Associativity Is Important

- Given a List of elements a, b, c, d and some Monoid to reduce the List
- foldRight

```
append( a, append( b, append( c, d ) ) )
```

- foldLeft

```
append( append ( append( a, b ) ), c ), d )
```

- Associativity guarantees same result for both

Why Associativity Is Important

- In many cases a "balanced fold" can be more desirable

```
append( append ( a, b ), append ( c, d ) )
```

- Inner calls to append() are independent of each other
  - Possibility of parallelization

- Also potentially more efficient if cost of append is proportional to size of operands

```
scala> sl
res67: List[String] = List(one, two, three)

scala> sl.foldLeft( "" )( _ + _ )
res68: String = onetwothree
```

Requires allocation of new String and copying of the two argument Strings

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
```


Using Monoid

- If the original element type is not one for which we have a suitable Monoid, we can convert to an appropriate type as part of the fold operation

```
scala> def foldMap[A,B](as: List[A], m: Monoid[B])(f: A => B) =
     |    as.foldLeft(m.zero)((b,a) => m.op(b, f(a)) )
foldMap: [A, B](as: List[A], m: Monoid[B])(f: A => B)B

scala> sl
res75: List[String] = List(one, two, three)

scala> foldMap(sl, IntAddMonoid)(_.length)
res78: Int = 11
```

Convert Strings to their lengths, then use the IntAddMonoid to calculate the sum of these lengths

More Complex Monoids

- If types A and B are Monoids, then (A,B) is also a Monoid

```
scala> def productMonoid[A,B](A: Monoid[A], B: Monoid[B]): Monoid[(A,B)] =
     |   new Monoid[(A,B)] {
     |     def zero = (A.zero, B.zero)
     |     def op( x: (A,B), y: (A,B) ) =
     |       ( A.op(x._1, y._1), B.op(x._2, y._2) )
     |   }
productMonoid: [A, B](A: Monoid[A], B: Monoid[B])Monoid[(A, B)]
```


More Complex Monoids

- Merging Maps
  - If type of values can form a monoid

```
def mapMergeMonoid[K,V]( V: Monoid[V]): Monoid[Map[K,V]] =
  new Monoid[Map[K,V]] {
    def zero = Map[K,V]()
    def op(a: Map[K,V], b: Map[K,V]) =
        (a.keySet ++ b.keySet).foldLeft(zero) { (acc, k) =>
            acc.updated(k, V.op(a.getOrElse(k, V.zero),
                                b.getOrElse(k, V.zero) ) )
         }
  }
```

More Complex Monoids

- Merging Maps
  - If type of values can form a monoid

```
scala> val m1 = Map("k1" -> 2, "k2" -> 3, "k3" -> 1)
m1: scala.collection.immutable.Map[String,Int] = Map(k1 -> 2, k2 -> 3, k3 -> 1)

scala> val m2 = Map("k2" -> 2)
m2: scala.collection.immutable.Map[String,Int] = Map(k2 -> 2)

scala> val mm: Monoid[Map[String, Int]] = mapMergeMonoid(IntMonoid)
mm: Monoid[Map[String,Int]] = $anon$1@1bac997f

scala> mm.op(m1, m2)
res85: Map[String,Int] = Map(k1 -> 2, k2 -> 5, k3 -> 1)
```