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
```

© J&G Services Ltd, 2016

Binary Operations

• Scala's Numeric typeclass makes this more concise

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

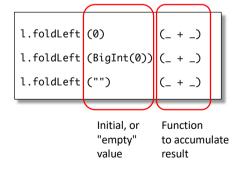
```
scala> def sum(l: List[String]) = l.foldLeft("")(_ + _)
sum: (l: List[String])String
scala> sum(List("one", "two"))
res61: String = onetwo
```

• But this will not work beside the Numeric typeclass

© J&G Services Ltd, 2016

Introducing the Monoid

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



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

© J&G Services Ltd, 2016

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
```

© J&G Services Ltd, 2016

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
```



© J&G Services Ltd, 2016

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( 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

Using Monoid

© J&G Services Ltd, 2016

- 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
```

© J&G Services Ltd, 2016

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

More Complex Monoids

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

© J&G Services Ltd, 2016

More Complex Monoids

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

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)
```