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

Functional Design Patterns

Lecture 130 of Advanced Programming

November 28, 2019

Andrzej Wasowski & Zhoulai Fu

IT University of Copenhagen

Background: Two ways to define a set

- Description-based
  - Days_in_a_week={Monday,Tuesday,…}
- Law-based (which subsumes the one above)
  - Naturals: 0 is an natural; n is a natural=> n+1 is a natural
  - Odd_int: Any integer n is odd is n/2 is still an integer
-

Today: Functional design patterns

- Part 1: Monoid
- Part 2: Foldable, Functor, and Monad

- We focus on types that follow certain laws.
- We prove rather than implement.

Type constructors and parameters

- List is a *type constructor,* not a type. We can apply it to the type Int, to produce the type List[Int]
- Thus, List needs a *parameter* to build a type
- We write A[_] for a type constructor A
- We write A[B] for a type

Quiz: discuss in groups of 2-3 people. Then, try it (3 min)

- Suppose that we have defined this class: A [I, J[_]]
- Is A a type? If not, what are its parameters?
- Does “new A[Int, List[Int]]” compile?

Demo

Monoid

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

“close”

- Int is closed with +, -, * but not /
- Double is closed all +, -, *, and /
- Positive number is not closed with subtraction

Demo

“associative”

```
scala> (1+2)+3==1+(2+3)
res55: Boolean = true

scala> (1*2)*3==1*(2*3)
res56: Boolean = true
```

“associative”

```
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
```

Not“associative”

```
scala> 0.1+(0.2+0.3)
res53: Double = 0.6

scala> (0.1+0.2)+0.3
res54: Double = 0.6000000000000001
```

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 $s \in S$.

Monoid

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

Quiz 1: Discuss in 2-3 and fill in the table (6 min)

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

Quiz 2: prove that the latter three are monoid

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

Why monoid matters?

- It makes programming simpler.
- It makes sense of parallel computing, by giving us the freedom to break our problem into chunks that can be computed in parallel.

“It makes programming simpler”

```
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

“It makes sense of parallel computing”

```
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

“It makes sense of parallel computing”

- With integers, (a + b + c + d) + (e + f + g + h) + (i + j + k + l) can be done with 3 CPUs.
- Not with floats!

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.
-

With monoids, foldLeft = foldRight

```
scala> val words = List("Hic", "Est", "Index")
words: List[String] = List(Hic, Est, Index)

scala> val s = words.foldRight(stringMonoid.zero)(stringMonoid.op)
s: String = "HicEstIndex"

scala> val t = words.foldLeft(stringMonoid.zero)(stringMonoid.op)
t: String = "HicEstIndex"
```

Further reading: Beautiful folds.
https://www.youtube.com/watch?v=6a5Ti0r8Q2s

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]
```

There is a typo on this slide. Can you find it?

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*

Illustration with boxes and arrows

- A box is a type constructor
- A box with a label is a type
- An arrow is a function

Then, the “map” in a functor looks like

A Functor Trait In Scala

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

- Functor is parameterized on the type constructor F, much like we did with Foldable.
- Functor law: map (x) (a => a) = x

Monad

A monad is, first of all, a wrapper

- def unit[A] (x: A): Monad[A]

In addition, a monad is flatMappable

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

- flatMap = first map, then flatten
- To flatMap an object of type M[A], we need a function of type A-> M[B]. FlatMap uses this function to transform A into M[B], resulting in a M[M[B]], and then it flattens the whole thing into 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 has unit and flatMap for a minimal setting
- Other functions, such as “map”, or “map2” can be defined via flatMap
- We have seen, and will see many monads. They may not call themselves monads, but knowing that they are can be useful

Illustration of monads functions with boxes and arrows

Function	Inputs	Output
map		
unit		
mpa2		
flatmap		

- Exercise 1: Derive map from map2
- Exercise 2: Derive map2 from flatmap

Whitebox demo

Conclusions

- Monoid has a closed, associative binary operation and an identity
- Foldable has foldMappable
- Functor has a map
- Monad has unit + flatMap
- These patterns are everywhere in function programming. Recognizing them gives us a new power to write concise, clear, less error-prone, and more reusable code.