Free monads

Alex Gryzlov
Adform
2015-04-29

Declarativity

- Making the computer do the work for you
- Reusing concepts and avoiding commitments for as long as possible
- We can reuse concepts from algebra with strong typing

Monoid

A data structure with an "empty" element and binary operation

```
trait Monoid[A] {
  val zero: A
  def append(a1: A, a2: A): A
}
```

Some laws:

```
// identity
A ⊕ 0 == 0 ⊕ A == A
// associativity
A ⊕ (B ⊕ C) == (A ⊕ B) ⊕ C
```

Functor

A data structure which knows how to apply pure functions inside itself

```
trait Functor[F[_]] {

  def map[A, B](fa: F[A])(f: A => B): F[B]

}
```

Very similar laws:

```
// identity
F map {x => x} == F
// associativity
F map f map g == F map (f compose g)
```

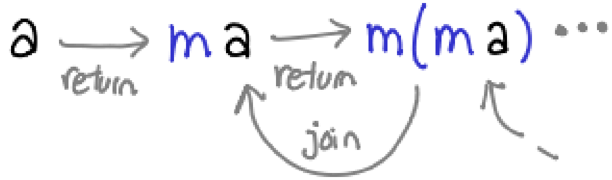
Monoid + Functor =

These are already pretty useful

e.g., Mapreduce (Hadoop) assumes your data forms a monoid (so it can shuffle the reduce steps)

But as any kid with 2 toys, let’s try to make them interact

Monoid of functors




```scala
trait Monad[F[_]] {

   def map[A,B](fa: F[A])(f: A => B): F[B]
   def point[A](a: A): F[A]
   // aka flatten
   def join[A](ffa: F[F[A]]): F[A]
   // aka flatMap, we can derive it
   def bind[A,B](fa: F[A])(f: A => F[B]): F[B] =
      join(map(fa)(f))
}
```

A monad is just a monoid in the category of (endo)functors!

Monoid laws

Let's try them on *List*:

```
// left identity (I*A == A)
scala> point(List(1,2))
res0: List[List[Int]] = List(List(1, 2))
scala> res0.flatten
res1: List[Int] = List(1, 2)
```

```
// right identity (A*I == A)
scala> List(1,2).map(point)
res2: List[List[Int]] = List(List(1), List(2))
scala> res2.flatten
res3: List[Int] = List(1, 2)
```

```
// associativity (A*(B*C) == (A*B)*C)
scala> val l3 = List(List(List(1), List(2)), List(List(3), List(4)))
scala> l3.flatten
res4: List[List[Int]] = List(List(1), List(2), List(3), List(4))
scala> l3.map(_.flatten)
res5: List[List[Int]] = List(List(1, 2), List(3, 4))
scala> res4.flatten == res5.flatten
res6: Boolean = true
```

Monad

- A functor with additional restrictions
- Flattens nested structures
- Chains together actions with effects

```scala
trait Monad[F[_]] {

   def point[A](a: A): F[A]
   // aka flatMap
   def bind[A,B](fa: F[A])(f: A => F[B]): F[B]
   // symmetrically, we can derive flatten
   def join[A](ffa: F[F[A]]): F[A] =
      bind(ffa)(a => a)
   // map is redundant now
   def map[A,B](fa: F[A])(f: A => B): F[B] =
      bind(fa)(a => point(f(a)))
}
```

Monadic sugar

For-notation:

```
Future(BigDecimal(2).pow(50)).flatMap( a =>
   Future(a+1).flatMap( b =>
      Future(a-1).map( c =>
         b*c
      )
   )
)

for {
  a <- Future(BigDecimal(2).pow(50))
  b <- Future(a + 1)
  c <- Future(a - 1)
} yield b*c
```

Monadic state

State monad in *Scalaz*:

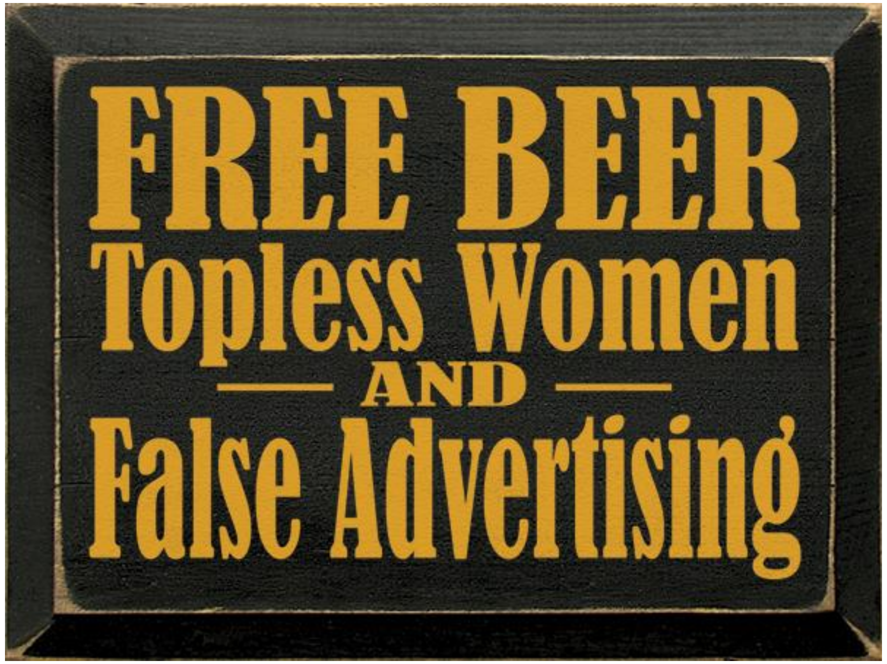
```
// State[S, A] is basically a tuple (s, a)

import scala.util.Random

def dice() = State[Random, Int](r => (r, r.nextInt(6) + 1))
def twoDice() = for {
  r1 <- dice()
  r2 <- dice()
} yield (r1, r2)

// start with a known seed
TwoDice().eval(new Random(1L))
```

Free objects



Free objects

A simplest possible way of getting *Foo* without providing any additional restrictions or structure

There are a lot:

- Free monoids
- Free groups
- Free functors
- Free categories
- …

Free monoid

A monoid which just keeps intact everything fed to it
(like a monoidal AST)

```
sealed trait FreeMonoid[+A] // to handle Append(smth, Zero)
final case object Zero extends FreeMonoid[Nothing]
final case class Value[A](a: A) extends FreeMonoid[A]
final case class Append[A](l: FreeMonoid[A], r: FreeMonoid[A]) extends FreeMonoid[A]
```

Not very useful, let's introduce a normalized version:

- Zero at the end
- Associativity allows us to chain all appends right to left

```
(A ⊕ (B ⊕ (C ⊕ 0)))
```

Free monoid v2

A normalized version

```
sealed trait FreeMonoid[+A]
final case object Zero extends FreeMonoid[Nothing]
final case class Append[A](l: A, r: FreeMonoid[A]) extends FreeMonoid[A]
```

This is actually the same as *List*!

(*List* is a monoid under concatenation)

Free monads idea

What's wrong with regular monads?

- They don't compose
- They force us to compute at the time of construction

The most important part is *join*

It loses structure (i.e., does <u>computation</u>)

What if we just promise to define it later (declarativity)?

The computation splits into <u>AST</u> and <u>interpreter</u>

Free monads

So, monad is a monoid of functors

Then free monad is a free monoid of functors (i.e., a list of functors)

```scala
def point[A](a: A): F[A]
def join[A](ffa: F[F[A]]): F[A]
def map[A,B](fa: F[A])(f: A => B): F[B]
```

```scala
sealed trait Free[F[_], A]
final case class Point[F[_], A](a: A) extends Free[F, A]
final case class Join[F[_], A](ff: F[Free[F, A]]) extends Free[F, A]
// we don't need to represent the map, just require that F is a functor
```

Instead of *Point* and *Join*, usually *Return* and *Suspend* are used

Free monad is a monad

Let’s see if it is:

```scala
sealed trait Free[F[_], A] {

  def point[F[_]](a: A): Free[F, A] = Point[F, A](a)

  def flatMap[B](f: A => Free[F, B])
               (implicit functor: Functor[F]): Free[F, B] =
    this match {
      case Point(a) => f(a)
      case Join(ff) => Join(ff map (_ flatMap f))
    }

  def map[B](f: A => B)
           (implicit functor: Functor[F]): Free[F, B] =
    flatMap(a => Point(f(a)))

}
```

Free monad helpers

You can lift things into a free monad (think of it as just taking A and returning *A :: Nil*):

```
def liftF[F[_], A](value: => F[A])(implicit F: Functor[F]): Free[F, A] =
  Suspend(F.map(value)(Return[F, A]))
```

And you can fold the sequence of steps to obtain the value:

```
trait ~>[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
}
```

```
def foldMap[F[_], M[_], A](fm: Free[F, A])(f: F ~> M)
           (implicit FI: Functor[F], MI: Monad[M]): M[A] =
 fm match {
   case Return(a) => MI.pure(a)
   case Suspend(ff) => MI.bind(f(ff))(foldMap(_)(f))
 }
```

Trampolines

No tail call optimization on JVM

Scala has *@tailrec*, but this still is a problem for, e.g. mutual recursion:

```
def isOdd(n: Int): Boolean = {
  if (n == 0) false
  else isEven(n - 1)
}
def even1(n: Int): Boolean = {
  if (n == 0) true
  else isOdd(n - 1)
}
isOdd(200000)   // boom
```

Trampolines

Let’s wrap the recursive calls:

```scala
def isOddT(n: Int): Bounce[Boolean] = {
  if (n == 0) Done(false)
  else Call(() => isEvenT(n - 1))
}
def isEvenT(n: Int): Bounce[Boolean] = {
  if (n == 0) Done(true)
  else Call(() => isOddT(n - 1))
}
```

```scala
sealed trait Bounce[A]
case class Done[A](result: A) extends Bounce[A]
case class Call[A](thunk: () => Bounce[A]) extends Bounce[A]

def trampoline[A](bounce: Bounce[A]): A = bounce match {
  case Call(thunk) => trampoline(thunk())
  case Done(x) => x
}
```

```scala
scala> trampoline(isOddT(200000))
res0: Boolean = false
```

Trampolines

Turns out our trampoline is just a special case of *Free*:

```
type Bounce[A] = Free[Function0, A]
```

And the *trampoline()* function is our first example of a free monad interpreter

Interpreters 1

We are defining *algebras* (a set of operations)

Let’s make one for an imaginary NoSQL DB

First, define your grammar

```scala
sealed trait KVS[+Next] // we need Next to have a functor
case class Put[T, Next](key: String, value: T, next: Next) extends KVS[Next]
case class Get[T, Next](key: String, onResult: T => Next) extends KVS[Next]
case class Delete[Next](key: String, next: Next) extends KVS[Next]
```

Interpreters 2

Second, prove it’s a functor (over the *Next* value)

```scala
implicit val functor: Functor[KVS] = new Functor[KVS] {
  def map[A, B](kvs: KVS[A])(f: A => B): KVS[B] = kvs match {
    case Put(key, value, next) => Put(key, value, f(next))
    // we need to help Scala with types here
    case g: Get[t, A] => Get[t, B](g.key, g.onResult andThen f)
    case Delete(key, next) => Delete(key, f(next))
  }
}
```

Interpreters 3

Third, make smart constructors for your grammar:

```scala
def put[T](key: String, value: T): KVS[Unit] = liftF(Put(key, value, ()))

def get[T](key: String): KVS[T] = liftF(Get[T, T](key, identity))

def delete(key: String): KVS[Unit] = liftF(Delete(key, ()))

// we can add more operations
def update[T](key: String, f: T => T): KVS[Unit] = for {
  v <- get[T](key)
  _ <- put[T](key, f(v))
} yield ()
```

Interpreters 4

Now you can write your program:

```
def program: KVS[Int] = for {
  _   <- put("foo", 1)
  _   <- update[Int]("foo", (_ + 2))
  id  <- get[Int]("foo")
} yield (id)
```

It does nothing, since it’s just a description of steps

Interpreters 5

Write a compiler (a *natural transformation*):

```scala
val db = scala.collection.mutable.Map.empty[String, Any]

type Id[A] = A   // a trivial functor/monad, we'll skip the definition

def mapCompiler = new (KVS ~> Id) {
  def apply[A](fa: KVS[A]): Id[A] = fa match {
    case p@Put(key, value, next) =>
          println(s"OP:$p")
          db += key -> value
          next
    case g: Get[t, A] =>
          println(s"OP:$g")
          g.onResult(db(g.key).asInstanceOf[t])
    case d@Delete(key, next) =>
          println(s"OP:$d")
          db -= key
          next
  }
}
```

Interpreters 6

Run it!

```
scala> val result: Id[Int] = program.foldMap(mapCompiler)
OP:Put(foo, 1, <...>)
OP:Get(foo, <function1>)
OP:Put(foo, 3, <...>)
OP:Get(foo, <function1>)
result: Id[Int] = 3
```

Performance

The problem with free monads is their quadratic complexity:

The longer the sequence, the longer is the *flatMap*

One of the solutions is to introduce a *Gosub* case that encodes *flatMap*:

```
case class Gosub[S[+_]: Functor, A, +B](a: Free[S, A], f: A => Free[S, B]) extends Free[S, B]
```

It represents a call to a subroutine

When it is finished, it continues the computation by calling the function f with the result.

Performance

Another problem is quadratic complexity when observing internal state

There are some solutions but it’s generally an ongoing work

Subsuming DI

So monads generally do not compose

But free monads sorta do!

Just wrap individual grammars in *Free* and combine them into a giant *Free*

This means we can do away with monadic transformers and dependency injection

Subsuming DI

Declarativity!

```
// naive
def transfer1(amount: Long, from: Account, to: Account, user: User, auth: Authorization,
 log: Logger, err: ErrorHandler, store: Storage): Unit

// context parameter, could be implicit etc
def transfer2(amount: Long, from: Account, to: Account, user: User, context:
Authorization with Logger with ErrorHandler with Storage): Unit

// return some instructions that include all of the above
def transferF(amount: Long, from: Account, to: Account, user: User): Free[Instruction, Unit]
```

You'll need things like *Inject* and *Coproduct* to achieve that

But ideally it'll all be hidden in library code

More

- Yoneda/Coyoneda
- Free applicatives
- Cofree comonads

Real projects using Free

- https://github.com/ethul/redis-algebra
- https://github.com/tpolecat/doobie
- https://github.com/xuwei-k/httpz

Links

- http://timperrett.com/2013/11/25/understanding-state-monad/
- http://polygonalhell.blogspot.cz/2014/12/scalaz-getting-to-grips-free-monad.html
- http://blog.higher-order.com/blog/2013/11/01/free-and-yoneda/
- http://mandubian.com/2015/04/09/freer/
- http://www.paolocapriotti.com/blog/2013/11/20/free-monads-part-1/ (3 parts)
- https://github.com/mandubian/injective