7 — Monads

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2020

Lehrstuhl für Informatik VI, Uni Würzburg

The Functor (Recap - Don’t do it again)

Implement the `Functor` for `Option`.

```scala
trait Functor[F[_]]:
  extension [A](fa: F[A])
    def map[B](f: A => B): F[B]
```

The Functor (Recap - Don't do it again)

Implement the `Functor` for `Option`.

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

Solution:

```
given Functor[Option] with
  extension [A](fa: Option[A])
    def map[B](f: A => B): Option[B] =
      fa match
        case None    => None
        case Some(a) => Some(f(a))
```

The Functor (Recap)

We can now define functions purely in terms of functor.

```scala
trait Functor[F[_]]:
  extension [A](fa: F[A])
    def map[B](f: A => B): F[B]

    /* map, but also keep old value */
    def fproduct[B](f: A => B): F[(A, B)] =
      fa.map(a => (a, f(a)))

  extension [A,B](fab: F[(A, B)])
    /* split a functor with a tuple */
    def unzip: (F[A], F[B]) =
      (map(fab)(_._1), map(fab)(_._2))
```

Both of these functions “just work” as long as there is a `map` method.

Motivating Example (Parsers)

This is a real example from my company.

We need to parse a schedule to control when the lights go on (and off) in gyms.

```
+ MO 08:00
- MO 14:00
+ MO 16:00
- MO 22:00
+ DI 08:00
- DI 14:00
+ DI 16:00
- DI 22:00
+ MI 08:00
- MI 14:00
+ MI 16:00
- MI 22:00
+ DO 08:00
- DO 14:00
```

```
+ DO 16:00
- DO 22:00
+ FR 08:00
- FR 14:00
+ FR 16:00
- FR 22:00
+ SA 08:00
- SA 14:00
+ SA 16:00
- SA 22:00
+ SO 08:00
- SO 14:00
+ SO 16:00
- SO 22:00
```

Motivating Example (Parsers)

And here's the code:

```
def parse(s: String) =
  (list <* endOfInput).parseOnly(s)

def list: Parser[List[Entry]] =
  entry.sepBy(string("\n"))

def entry: Parser[Entry] = for
  dir     <- direction
  _       <- ws
  day     <- weekDay
  _       <- ws
  hours   <- int
  _       <- string(":")
  minutes <- int
yield Entry(dir, day, hours, minutes)
```

```
def direction: Parser[Boolean] =
  string("+").as(true) <+>
  string("-").as(false)

def weekDay: Parser[DayOfWeek] = List(
  ("MO", DayOfWeek.MONDAY),
  ("DI", DayOfWeek.TUESDAY),
  ("MI", DayOfWeek.WEDNESDAY),
  ("DO", DayOfWeek.THURSDAY),
  ("FR", DayOfWeek.FRIDAY),
  ("SA", DayOfWeek.SATURDAY),
  ("SO", DayOfWeek.SUNDAY)
).foldMap({ case (s, d) =>
    string(s).as(d)})

def ws: Parser[Unit] =
  horizontalWhitespace.many1.void
```

Motivating Example (Parsers)

What have we seen here?

- The for comprehension isn't part of the parser library
- The `<*` operator is not part of the parser library
- The `as` method is not part of the parser library
- The `<+>` operator is not part of the parser library
- The `.void` method is not part of the parser library
- The `.foldMap` method is not part of the parser library

All those methods and operators come from the `Monad` and `Alternative` type classes. Which means you can write a large portion of your program without having to learn special combinators for the `Parser` data type.

Motivating Example (Probability)

We want to do a small RPG simulation.

We have two players. Both players first do an attack roll, the one who rolls higher can hit the other. The player who hits then rolls an attack and subtracts that from the hitpoints of the other player. Here are the hit and damage dice:

	Attack	Damage
Player1	1W3 + 1W2	2W3 + 2
Player2	1W6 + 1	4

We want to simulate who of the players wins, given some initial hit points for both players.

Motivating Example (Probability)

Here’s the game state:

```
final case class State(hp1: Int, hp2: Int):
  def hitFirst(dmg: Int): State = copy(hp1 = hp1 - dmg)
  def hitSecond(dmg: Int): State = copy(hp2 = hp2 - dmg)
  def winner: Option[Boolean] =
    if      hp1 <= 0 then Some(false)
    else if hp2 <= 0 then Some(true)
    else                  None
```

Motivating Example (Probability)

And here's the simulation:

```
def fight(s: State): Prob[Boolean] =
  for
    oneHits  <- playerOneHits

    newState <-
      if oneHits then
        for dmg <- rollDamage1
        yield s.hitSecond(dmg)
      else
        for dmg <- rollDamage2
        yield s.hitFirst(dmg)

    result <- newState.winner match
        case Some(w) => w.pure[Prob]
        case None    => fight(newState)
  yield result

def playerOneHits =
  (rollHit1, rollHit2).mapN(_ >= _)
```

```
def rollHit1 = for
  r1 <- dice(3)
  r2 <- dice(2)
yield r1 + r2

def rollHit2 =
  dice(6).map(_ + 1)

def rollDamage1 =
  rollMupltiple(2, 3).map(_ + 2)

def rollMupltiple(n: Int, d: Int) =
  List.fill(n)(dice(d))
    .sequence.map(_.sum)

def rollDamage2 =
  4.pure[Prob]
```

Motivating Example (Probability)

And this is the output:

```
false -> 0.4284250023956541, true -> 0.5715749976043474
```

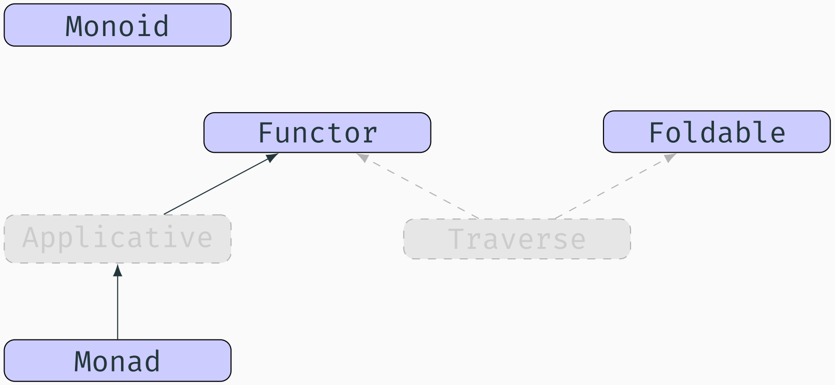
What have we seen here?

- The for comprehension isn't part of the prob library
- The `.mapN` method is not part of the prob library
- The `.map` method is not part of the prob library
- The `.sequence` method is not part of the prob library

All those methods and operators come from the `Monad` and `Traversable` type classes. Which means you can write a large portion of your program without having to learn special combinators for the `Prob` data type.

Monads

Typeclass hierarchy



The Monad

The functor abstraction we saw last week is great and it let's us abstract over a lot of data structures.

But there are also a lot of functions we can't write when we are given only a functor instance. One example is combining multiple values `F[A], F[B], …` into a single `F`.

Let's introduce a more powerful abstraction: the monad.

```
trait Monad[F[_]]:
  def pure[A](a: A): F[A]

  extension [A](fa: F[A])
    def flatMap[B](f: A => F[B]): F[B]
```

With a monad we can lift an expression into the monad (`pure`).

We can also combine the current monad together with a function which takes its value and yields a new monadic value to get a new monad (`flatMap`).

The Monad

Monads are strictly more powerful than functors. How would we prove that?

The Monad

Monads are strictly more powerful than functors. How would we prove that?

```scala
def functorFromMonad[F[_]](using m: Monad[F]): Functor[F] = new Functor[F]:
  extension [A](fa: F[A])
    def map[B](f: A => B): F[B] =
```

We prove it by providing a function which gives us a `Functor` for every type for which we know it’s a `Monad`.

The Monad

Monads are strictly more powerful than functors. How would we prove that?

```
def functorFromMonad[F[_]](using m: Monad[F]): Functor[F] = new Functor[F]:
  extension [A](fa: F[A])
    def map[B](f: A => B): F[B] =
```

We prove it by providing a function which gives us a `Functor` for every type for which we know it's a `Monad`.

Solution:

```
def functorFromMonad[F[_]](using m: Monad[F]): Functor[F] = new Functor[F]:
  extension [A](fa: F[A])
    def map[B](f: A => B): F[B] =
      fa.flatMap(a => m.pure(f(a)))
```

The Monad — Exercise: map2

But is a monad powerful enough to let us combine multiple values inside `F`?

Go back to the code and implement the `map2` function.

```
extension [A](fa: F[A])
  // ...
  def map2[B, C](fb: F[B])(f: (A, B) => C): F[C] = ???
```

The Monad — Exercise: map2

But is a monad powerful enough to let us combine multiple values inside `F`?

Go back to the code and implement the `map2` function.

```scala
extension [A](fa: F[A])
  // ...
  def map2[B, C](fb: F[B])(f: (A, B) => C): F[C] = ???
```

Solution:

```scala
def map2[B, C](fb: F[B])(f: (A, B) => C): F[C] =
  fa.flatMap(a => fb.flatMap(b => pure(f(a, b))))
```

The Monad

Which data structures we already encountered are monads?

The Monad

Which data structures we already encountered are monads?

- Option
- Either
- List (and other collections like LazyList, Vector, …)
- Parser
- Prob

The Monad — Exercise: sequence

We have seen **`sequence`** pop up at various places. For example to transform a list of potential errors (`List[Either[E, A]]`) into either an error or a list of values (`Either[E, List[A]]`).

Sequence can be implemented for any monad.

Go back to the code and add the function

```
extension [A](fas: List[F[A]])
  def sequence: F[List[A]] = ???
```

You may use `map2` from our monad, and `foldRight` on `List` (courtesy of the standard library).

The Monad — Exercise: sequence

We have seen **`sequence`** pop up at various places. For example to transform a list of potential errors (`List[Either[E, A]]`) into either an error or a list of values (`Either[E, List[A]]`).

Sequence can be implemented for any monad.

Go back to the code and add the function

```
extension [A](fas: List[F[A]])
  def sequence: F[List[A]] = ???
```

You may use `map2` from our monad, and `foldRight` on `List` (courtesy of the standard library). Solution:

```
extension [A](fas: List[F[A]])
  def sequence: F[List[A]] =
    fas.foldRight(pure(List.empty[A]))((a, b) => a.map2(b)(_::_))
                    // ↑ or Nil: List[A]
```

The Monad

Sequencing does very different things for different monads.

Option	`List[Option[A]]` -> `Option[List[A]]`. Yields either an empty option or a list of all the values.
Either	`List[Either[E, A]]` -> `Either[E, List[A]]`. Yields either the first error or a list of all the values.
List	`List[List[A]]` -> `List[List[A]]`. Creates the Cartesian product of all the lists.
Parser	`List[Parser[A]]` -> `Parser[List[A]]`. Parses all the items in sequence.
Prob	`List[Prob[A]]` -> `Prob[List[A]]`. Creates a probability distribution of a list.

The Monad Laws

Now that we know a bit about monads, let’s talk laws.

A monad has to obey two laws:

- The associative law
- The identity law

Let’s take them one by one.

The Monad Laws

Look at those two programs

```
val progA: Option[Order] = for
  name     <- getName
  price    <- getPrice
  quantity <- getQuantity
yield Order(Item(name, price), quantity)
```

```
val progB: Option[Item] = for
  name     <- getName
  price    <- getPrice
yield Item(name, price)

val progC: Option[Order] = for
  item     <- progB
  quantity <- getQuantity
yield Order(item, quantity)
```

We expect them to be equal. In both cases, the result of `progA` and `progC` should be the same.

The Monad Laws

But they don’t expand to the same code

```scala
val progA: Option[Order] =
  getName.flatMap(name =>
  getPrice.flatMap(price =>
  getQuantity.map(quantity =>
      Order(Item(name, price), quantity))))
```

```scala
val progC: Option[Order] =
  getName.flatMap(name =>
  getPrice.map(price =>
      Item(name, price))).flatMap(i =>
  getQuantity.map(quantity =>
      Order(i, quantity)))
```

For example, we have one `map` call on the left side, but two on the right side. This is a problem.

The Monad Laws

To ensure that both of the code examples discussed just now always behave the same, every monad has to follow the associative law:

```
x.flatMap(f).flatMap(g) == x.flatMap(a => f(a).flatMap(g))
```

The Monad Laws

We can prove that this law holds for `Option`, for example, by applying the substitution model given to us by referential transparency.

```
x.flatMap(f).flatMap(g) == x.flatMap(a => f(a).flatMap(g))
```

For None:

```
None.flatMap(f).flatMap(g)  ==  None.flatMap(a => f(a).flatMap(g))
          None.flatMap(g)  ==  None
                     None  ==  None
```

The Monad Laws

We can prove that this law holds for `Option`, for example, by applying the substitution model given to us by referential transparency.

```
x.flatMap(f).flatMap(g) == x.flatMap(a => f(a).flatMap(g))
```

For `Some(v)`:

```
Some(v).flatMap(f).flatMap(g)   ==   Some(v).flatMap(a => f(a).flatMap(g))
             f(v).flatMap(g)   ==   (a => f(a).flatMap(g))(v)
             f(v).flatMap(g)   ==   f(v).flatMap(g)
```

The Monad Laws

So the associative law holds for `Option`. It is not clear, though, that this is an *associative* law at all.

Associativity laws for an operator $\oplus$ normaly take the form

$$a \oplus (b \oplus c) == (a \oplus b) \oplus c)$$

which looks quite different from

```
x.flatMap(f).flatMap(g) == x.flatMap(a => f(a).flatMap(g))
```

The Monad Laws

We can look at it through the lens of function composition.

Instead of functions of the type `A => B` we look at functions of the type `A => F[B]`, where `F` is a monad.

Those functions are called **Kleisli Arrows** and they can be composed like normal functions as long as we have a monad instance for `F`.

With that in mind, we can restate our associative law this way:

```
compose(f, compose(g, h)) == compose(compose(f, g), h)
```

The Monad Laws

Let’s try to implement `compose`.

Go back to the code and add the function

```scala
def compose[A, B, C](f: A => F[B])(g: B => F[C]): A => F[C] = ???
```

Note that we have two parameter lists here. They are only here to aid type inference, but the function is equivalent to the theoretical compose function above. You should be able to implement it just by using `flatMap`

The Monad Laws

Let's try to implement `compose`.

Go back to the code and add the function

```
def compose[A, B, C](f: A => F[B])(g: B => F[C]): A => F[C] = ???
```

Note that we have two parameter lists here. They are only here to aid type inference, but the function is equivalent to the theoretical compose function above. You should be able to implement it just by using `flatMap`

Solution:

```
def compose[A, B, C](f: A => F[B])(g: B => F[C]): A => F[C] =
  a => f(a).flatMap(g)
```

The Monad Laws

The other law was the identity law, which can be stated as follows:

- `flatMap(x)(pure) == x`
- `flatMap(pure(y))(f) == f(y)`

or, alternatively

- `compose(f, pure) == f`
- `compose(pure, f) == f`

What Does “Monad” Mean?

It’s hard to describe what a monad really is, because monads don’t have shared funcationality. They can pass state along, do error handling, build cartesian products or do nothing at all.

They are described purely by an algebraic interface with laws:

> *A monad is an implementation of one of the minimal sets of monadic combinators, satisfying the laws of associativity and identity.*

The “minimal set of monad combinators” we used are `flatMap` and `pure`, but there are others.

What Does “Monad” Mean?

We already have seen many monads which do different things. Now let’s look at a monad which doesn’t do anything:

```scala
final case class Id[A](value: A)
```

Implement the monad instance for `Id`

What Does “Monad” Mean?

We already have seen many monads which do different things. Now let’s look at a monad which doesn’t do anything:

```scala
final case class Id[A](value: A)
```

Implement the monad instance for `Id` Solution:

```scala
given Monad[Id] with
  def pure[A](a: A): Id[A] = Id(a)

  extension [A](fa: Id[A])
    def flatMap[B](f: A => Id[B]): Id[B] = f(fa.value)
```

What Does “Monad” Mean?

The only way to really understand what monads are and what they are good for is to use many of them.

You will learn something new many times.

Summary

We have seen:

- What a monad is (laws and signatures)
- That `sequence` and `map2` work for every monad
- That `Either`, `Option`, `List`, `Prob`, `Parser` and `Id` are all monads, even though they do comletely different things
- What kleisli arrows are and how to compose them

From @impurepics / impurepics.com