INFO0054-1
Programmation Fonctionnelle

Chapter 08: Functors and Monads

Christophe Debruyne

(c.debruyne@uliege.be)

References

- Chapter 11: Monads.
  Paul Chiusano and Runar Bjarnason. Functional Programming in Scala, Manning Publications, 2015.

DISCLAIMER: Unlike languages such as Haskell, Scala does not provide types for Foldable, Monoid, Functor, and Monad. But as they are abstract concepts representing objects that follow certain laws, we can easily represent those in Scala.

In the next few lessons, we will learn more about these abstract types. There are libraries for Scala that do provide these types. Examples include:

- **Scalaz https://github.com/scalaz/scalaz**
- **Cats https://typelevel.org/cats/**

Overview

- Introduction – The Option and List ADTs
- Functors
- Monads

Part 1: Introduction

Introduction

- Monoids was our first instance of a completely abstract and purely algebraic interface which were defined in terms op operations that complied with certain laws.

- There are other such interfaces, namely functors and monads, which we will cover in this class.

- In this book, they exemplify these with the various libraries they have built for objects of the type `List`, `Gen`, `Parser`, etc. As we have not covered those in this course, we will apply those to `List` and `Option`.

- Let's first look a implementations of `List` and `Option`. These implementation do not include various utility functions for the purpose of this chapter.

Reminder: the List ADT

```
enum List[+A]:
    case Nil
    case Cons(head: A, tail: List[A])

    def foldRight[B](z: B, f: (A, B) => B): B = this match
        case Nil => z
        case Cons(h, t) => f(h, t.foldRight(z, f))

    def map[B](f: A => B): List[B] =
        flatMap(a => List(f(a)))

    def flatMap[B](f: A => List[B]): List[B] =
        foldRight(Nil: List[B], (a, acc) => f(a).append(acc))
```

L'objet compagnon n'est pas inclut

Reminder: the Option ADT

```
enum Option[+A]:
    case Some(get: A)
    case None

    def getOrElse[B>:A](default: => B): B = this match
        case None => default
        case Some(a) => a

    def map[B](f: A => B): Option[B] =
        flatMap(a => Some(f(a)))

    def flatMap[B](f: A => Option[B]): Option[B] = this match
        case None => None
        case Some(a) => f(a)
```

Part 2:
Functors

What is a Functor? I

- In category theory, a branch of mathematics, a functor is a mapping (!) between categories. In other words, it is a mapping between algebraic objects (*). These functors need to abide some laws, which we will come back to later.
  - (*) where did we mention algebraic objects before?

- In CS, and more specifically FP, we have adopted the term functor for describing the technique of generic ADTs (i.e., not bound to a specific type) to apply a function to its data without changing the structure of the ADT. **I.e., we can transform the values inside our ADT without transforming the ADT nor its structure.**

```
scala> List(1,2,3).map(_ % 2 == 0)
val res2: List[Boolean] = Cons(false,Cons(true,Cons(false,Nil)))
```

La structure est préservée

```
scala> Some(1).map(_ + 2)
val res3: Option[Int] = Some(3)
```

What is a Functor? II

```
enum List[+A]:
   ...
    def map[B](f: A => B): List[B] =
        foldRight(Nil: List[B], (h, t) => Cons(f(h), t))

enum Option[+A]:
    ...
    def map[B](f: A => B): Option[B] = this match
      case None => None
      case Some(a) => Some(f(a))
```

- Notice that the <u>signatures</u> for both `map`s are very similar.
- <u>The only difference is in the data type they return</u>.

What is a Functor? III

```
enum List[+A]:
    ...
    def map[B](f: A => B): List[B] =
        foldRight(Nil: List[B], (h, t) => Cons(f(h), t))

enum Option[+A]:
    ...
    def map[B](f: A => B): Option[B] = this match
      case None => None
      case Some(a) => Some(f(a))
```

- There is apparently a common approach to transform the values inside these ADTs without changing the structure; the function map.
- The question that we can ask ourselves now. Can we create abstractions that allow us to use, reason, build, etc. functions without specifying the ADT? The answer is, unsurprisingly, yes: with functors. :-)

Higher-kinded types

- Remember that type parameters are enclosed in square brackets, while value parameters are enclosed in parentheses. So far, we have only seen type parameters that abstract over some type, e.g., `def map[A,B](l: List[A], f: A => B): List[B]`.

- We have also seen type constructors such as `List` and `Option`. We cannot have values of type `List`, for instance. However, we can apply type constructors to a type. For instance, we can apply the type constructor `List` to the type `Int` to create objects of the type `List[Int]`.

- In Scala, you have the possibility to declare higher-kinded types. Higher-kinded types are types that abstract over some type X and that type X abstracts over another type Y. In other words, we can abstract over type constructors.

Creating a trait Functor

- Let's create a trait functor

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

- **<u>In the scope of our functor</u>**, we extend all objects that match this type parameter with a method `map`.
- That `map` function needs to be specified for all type constructors.

- Notice that functors that "contain" objects of type `A`, no matter the type constructor `F`, these objects must implement a `map` function.
- We will see that specific implementations will delegate this to the map functions of our ADTs.
- Let's create functors for `List` and `Option`.

Creating a functor object

```
import List._
import Option._

given of: Functor[Option] with
    extension [A](o: Option[A])
        def map[B](f: A => B): Option[B] = o.map(f)

given lf: Functor[List] with
    extension [A](l: List[A])
        def map[B](f: A => B): List[B] = l.map(f)


----------

scala> val l1 = List(1,2,3,4)
val l1: List[Int] = Cons(1,Cons(2,Cons(3,Cons(4,Nil))))

scala> l1.map(_ + 1)
val res0: List[Int] = Cons(2,Cons(3,Cons(4,Cons(5,Nil))))

scala> lf.map(l1)(_ + 1)
val res1: List[Int] = Cons(2,Cons(3,Cons(4,Cons(5,Nil))))
```

- Be certain to import our ADTs. Otherwise, you will be referring to objects in Scala.
- `Functor[Option]` is a functor object that operates on `Option`. If given an object `o` of the type `Option[A]`, then calling its `map` function will call `o`'s `map` function.
- `Functor[List]` is a functor object that operates on `List`. If given an object `l` of the type `List[A]`, then calling its `map` function will call `l`'s `map` function.

Ok… But what's the point? Well, we can now think of new operations that operate on the interface of functor!

map and scope.

- We will add println statements to observe what is going on.
- Remember, we have declared an extension of objects that match F[A].
- That extension declares a map function. The map function needs to be implemented for every type constructor such as List.
- A List[_] object already has a map defined. Which one will be executed, that depends on the context.

```scala
import List._
import Option._

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

given lf: Functor[List] with
    extension [A](l: List[A])
        def map[B](f: A => B): List[B] =
            println("We execute map of functor")
            l.map(f)

----------

scala> val x = List(1)
val x: List[Int] = Cons(1,Nil)

scala> x.map(_ + 1)
val res0: List[Int] = Cons(2,Nil)

scala> lf.map(x)(_ + 1)
We execute map of functor
val res1: List[Int] = Cons(2,Nil)
```

Here, we declare the signature of map, but there is no definition. Definitions need to be specified by specializations.

Here, we apply the map function defined by List.

Operations with functors I

- We can discover and implement useful functions on the functor's interface <u>in an algebraic way</u>. For instance, unzip!

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

    extension [A, B](fab: F[(A, B)])
        def unzip: (F[A], F[B]) =
            (fab.map(_(0)), fab.map(_(1)))
```

The function `unzip` is called `distribute` in the book.

- Given a type constructor that contain tuples of two objects of types A and B, we return a tuple containing two objects of that type constructor. The first contains all the As, and the second contains all the Bs.
- Notice that the structure of ADT is preserved and notice that the abstraction does not mention any specific type constructors.

`unzip` and scope.

- We will add `println` statements to observe what is going on.
- Remember, we have declared an extension of objects that match `F[(A, B)]`. That means, we extend objects of type constructor `F` containing tuples of arity two that contain an `A` and a `A`. These objects will have access to a function `unzip` that takes no argument.
- Since our ADTs have not implemented a function `unzip`, that implementation of the function will be called <u>no matter the context</u>!

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

    extension [A, B](fab: F[(A, B)])
        def unzip: (F[A], F[B]) =
            println("We execute unzip of functor")
            (fab.map(_(0)), fab.map(_(1)))

----------

scala> val x = List((1,2))
val x: List[(Int, Int)] = Cons((1,2),Nil)

scala> lf.unzip(x)
We execute unzip of functor
val res0: (List[Int], List[Int]) = (Cons(1,Nil),Cons(2,Nil))

scala> x.unzip
We execute unzip of functor
val res1: (List[Int], List[Int]) = (Cons(1,Nil),Cons(2,Nil))
```

`List[(Int, Int)]` matches with `F[(A, B)]` and is thus extended with an `unzip` function.

Operations with functors II

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

    extension [A, B](fab: F[(A, B)])
        def unzip: (F[A], F[B]) =
            (fab.map(_(0)), fab.map(_(1)))
```

- We have just defined a new and useful combinator that is based purely on the abstract interface of `Functor`. For objects of the type `F[(A, B)]`.

- Or, as Chiusano and Bjarnason (2022) state: "Since we know nothing about other than that it's a functor, the law assures us that the returned F values will have the same shape as the arguments."

Operations with functors III

```
scala> val l = List((1,2),(3,4),(5,6))
val l: List[(Int, Int)] = Cons((1,2),Cons((3,4),Cons((5,6),Nil)))

scala> l.unzip
val res2: (List[Int], List[Int]) =
(Cons(1,Cons(3,Cons(5,Nil))),Cons(2,Cons(4,Cons(6,Nil))))



scala> val o = Some((1,2))
val o: Option[(Int, Int)] = Some((1,2))

scala> o.unzip
val res3: (Option[Int], Option[Int]) = (Some(1),Some(2))
```

Operations with functors IV

```
trait Functor[F[_]]:
    // ...
    extension [A](fa: F[A])
        def computeTuples[B](f: A => B): F[(A,B)] =
            fa.map((x) => (x, f(x)))

    extension [A, B](e: Either[F[A], F[B]])
        def codistribute: F[Either[A, B]] = e match
            case Left(a) => a.map(Left(_))
            case Right(b) => b.map(Right(_))
----------
scala> List(1,2,3).computeTuples(_ * 2)
val res0: List[(Int, Int)] = Cons((1,2),Cons((2,4),Cons((3,6),Nil)))

scala> Some(1).computeTuples(_ * 2)
val res1: Option[(Int, Int)] = Some((1,2))

scala> Left(List(2)).codistribute
val res2: List[Either[Int, Nothing]] = Cons(Left(2),Nil)
```

Functor laws

A type constructor with a `map` function is considered a functor only if its `map` function obeys two laws:

- The law of composition: `foo.map(f).map(g)` and `foo.map(f.andThen(g))` should be equal.
  - In other words, two passes over the ADT two apply first `f` and then `g` should return the same result as a single pass with composition of `f` and `g`.
- The law of identity: applying `map` on the identify function `(x) => x` should return an object that is equal to the original.
  - In other words, it should return "the same object." I use parentheses as we will build a new object, but its structure and values are the same.

Functors are thus "mappable" and generic ADTs whose map functions obey the laws of composition and identity.

By the way, law of composition I

In languages such as Haskell, the law of composition is defined as follows:

- "Functors preserve composition of morphisms
  `fmap (f . g)  ==  fmap f . fmap g`
  If two sequential mapping operations are performed one after the other using two functions, the result should be the same as a single mapping operation with one function that is equivalent to applying the first function to the result of the second." (https://wiki.haskell.org/Functor)

- Here, `fmap` is "equal" to `map` (you should read it as "functor map").
- `f . g` means "apply first g, then f"
- `fmap f . fmap g` means "apply first the map of g, then the map of f"

- While we described this law with `andThen`, it boils down to the same.

By the way, law of composition II

We defined it in terms of andThen since Scala uses OOP notation, which may complicate things. Languages such as Haskell and Scheme do not have this and (f)map is a function that takes two arguments; a function and an ADT.

```
def map[A,B](f: A => B) = (l: List[A]) => l.map(f)
def f(n: Int): Int = n + 3
def g(n: Int): Int = n * 2

----------

scala> map(f.compose(g))(List(1,2,3,4))
val res0: List[Int] = List(5, 7, 9, 11)

scala> map(f).compose(map(g))(List(1,2,3,4))
val res1: List[Int] = List(5, 7, 9, 11)
```

Hold on a minute I

- We have seen `Foldable` data structures, which have a function `foldMap`.
- Now we see `Functor`s, which are also structures containing data, and they have a function `map`.

- Are Foldable and Functor the same? No.

- `Foldable` → are containers that can be "folded" (i.e., combined) into one value. Values can be combined using monoids, and we can use this to transform the data contained in a foldable into a list. The use foldMap can thus produce a new structure.
  - This implies that we can enumerate them.
  - We used the function `foldMap`, in other books they use `fold`, `foldr`, `foldRight`, …
- `Functor` → are containers that allow one to `map` a function to all its elements and the application of `Functor` must preserve the structure of the container.

Hold on a minute II

- Many (interesting) foldables are also functors.
- But that implies there are foldables that aren't functors!

- A straight-forward example are HashSets (in Scala).

```
scala> val foo =            Set(1, 2, 3, 4, 5)
val foo: Set[Int] =   HashSet(5, 1, 2, 3, 4)

scala> foo.fold(0)(_ + _)
val res0: Int = 15

scala> foo.map(_ + 5)
val res1: Set[Int] = HashSet(10, 6, 9, 7, 8)
```

A Set is foldable …

… but isn't a functor.

Part 3

Part 3:

Monads

Preambule

- In the book, the authors introduced a function `map2` for many of the ADTs they develop. This is not a function that is available (as such) in Scala's standard library but will examine the function `map2` for the point the authors want to illustrate.

- The function `map2` combines two values of a datatype using a binary function. With this function, we never need to modify any functions to have them operate on two objects of the same ADT.

map2 for List

```
enum List[+A]:
    case Nil
    case Cons(head: A, tail: List[A])

    // ...
    def map[B](f: A => B): List[B] =
        flatMap(a => List(f(a)))

    def flatMap[B](f: A => List[B]): List[B] =
        foldRight(Nil: List[B], (a, acc) => f(a).append(acc))

    def map2[B,C](lb: List[B])(f: (A,B) => C): List[C] =
        flatMap(a => lb.map(b => f(a,b)))
----------
scala> List(1,2).map2(List(3,4))(_ + _)
val res4: List[Int] = Cons(4,Cons(5,Cons(5,Cons(6,Nil))))
```

map2 for Option

```scala
enum Option[+A]:
    case Some(get: A)
    case None

    // ...
    def map[B](f: A => B): Option[B] =
        flatMap(a => Some(f(a)))

    def flatMap[B](f: A => Option[B]): Option[B] = this match
        case None => None
        case Some(a) => f(a)

    def map2[B,C](ob: Option[B])(f: (A, B) => C): Option[C] =
        flatMap(a => ob.map(b => f(a, b)))
----------
scala> Some(1).map2(Some(2))(_ + _)
val res5: Option[Int] = Some(3)
```

Identifying a pattern

- Notice that the two functions are very similar:

```
def map2[B,C](lb: List[B])(f: (A,B) => C): List[C] =
    flatMap(a => lb.map(b => f(a,b)))

def map2[B,C](ob: Option[B])(f: (A, B) => C): Option[C] =
    flatMap(a => ob.map(b => f(a, b)))
```

Unit. Create a function to create "elementary "objects of our ADT?

- Also recall that we can implement `flatMap` using `map`:

```
def map[B](f: A => B): List[B] = flatMap(a => List(f(a)))
def map[B](f: A => B): Option[B] = flatMap(a => Some(f(a)))
```

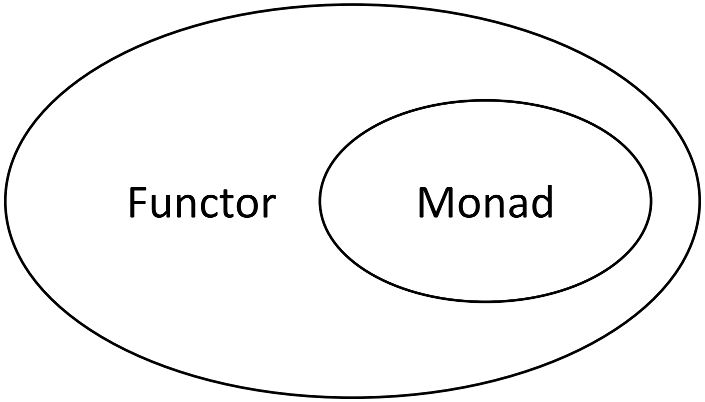
Do we have a pattern emerging? Yes!

If we have a `flatMap` and a `unit`, we can create an abstractions for `map`, `map2`, etc.

Hold on a minute… III

We have already identified part of the Monad, but we have not yet covered the monadic laws, but …

1. The functions `flatMap` and `unit` allow us to implement `map`.
2. Does `map` correspond with `map` from the previous part? Yes!
3. Indeed, <u>every Monad is a Functor, but not every Functor is a Monad</u>.



Monadic laws

"A monad is an ***implementation*** of one of the minimal sets of monadic combinators, satisfying the laws of associativity and identity." (Chiusano and Bjarnason, 2015)

- Set 1: `unit` and `flatMap`.
- ~~Set 2: `unit` and `compose`.~~
- ~~Set 3: `unit`, `map`, and `join`.~~

A monad consists of 3 things:

- A type constructor M for on a type A.
- A `unit` function that embeds an object of type A in a monad M.
- A `flatMap` function that takes the monadic variable and applies is to a function that yields a new monadic value.

The identity laws

1) Right identity law

unit is a right-identity for flatMap: `x.flatMap(unit) = x`

```
scala> List(1,2,3).flatMap(List(_))
val res2: List[Int] = Cons(1,Cons(2,Cons(3,Nil)))

scala> Some(1).flatMap(Some(_))
val res3: Option[Int] = Some(1)
```

2) Left identity law

unit is a left-identity for flatMap: `unit(y).flatMap(f) = f(y)`

```
scala> List(3).flatMap((a: Int) => List(a + 3))
val res0: List[Int] = Cons(6,Nil)

scala> ((a: Int) => List(a + 3))(3)
val res1: List[Int] = Cons(6,Nil)
```

The law of associativity I

The book demonstrates how one can prove the law of associativity for the Option Monad by using the substitution model (for both Some and None). This is beyond the scope of this course, but you are invited to read this chapter to get an intuitive understanding of that process.

<u>3) flatMap obeys the associative law</u>.

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

For example:

```
val f = (x: Int) => Some(x + 3)
val g = (x: Int) => Some(x * 2)
----------
scala> Some(3).flatMap(f).flatMap(g)
val res1: Option[Int] = Some(12)

scala> Some(3).flatMap(a => f(a).flatMap(g))
val res2: Option[Int] = Some(12)
```

Let's rewrite our code I

```scala
enum List[+A]:
    case Nil
    case Cons(head: A, tail: List[A])

    def foldRight[B](z: B, f: (A, B) => B): B = this match
        case Nil => z
        case Cons(h, t) => f(h, t.foldRight(z, f))

    def append[B>:A](a2: List[B]): List[B] = this match
        case Nil => a2
        case Cons(h,t) => Cons(h, t.append(a2))

    def flatMap[B](f: A => List[B]): List[B] =
        foldRight(Nil: List[B], (a, acc) => f(a).append(acc))

object List:
    def apply[A](as: A*): List[A] =
        if(as.isEmpty) Nil
        else Cons(as.head, apply(as.tail: _*))

enum Option[+A]:
    case Some(get: A)
    case None

    def getOrElse[B>:A](default: => B): B = this match
      case None => default
      case Some(a) => a

    def flatMap[B](f: A => Option[B]): Option[B] = this match
        case None => None
        case Some(a) => f(a)
```

Notice that I only have a concrete definition for `flatMap` (and functions specific to these ADTs).

These ADTs are assumed to reside in the same file. Feel free to declare them in separate packages.

Let's rewrite our code II

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

    // other functions omitted for brevity

trait Monad[F[_]] extends Functor[F]:
  def unit[A](a: => A): F[A]

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

    def map[B](f: A => B): F[B] =
      fa.flatMap(a => unit(f(a)))

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

We are now defining map using `flatMap`!

Since all Monads are Functors, and Functors require a `map`, we have ensured that all monads have an implementation of `map`.

Let's rewrite our code III

```
object Monad:
    import Option._
    given optionMonad: Monad[Option] with
        def unit[A](a: => A) = Some(a)

        extension [A](fa: Option[A])
            def flatMap[B](f: A => Option[B]) =
                fa.flatMap(f)

    import List._
    given listMonad: Monad[List] with
        def unit[A](a: => A) = List(a)

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

We are now defining map using `flatMap`!

Since all Monads are Functors, and Functors require a `map`, we have ensured that all monads have an implementation of `map`.

Using our monads

```
scala> val l1: List[Int] = List(1,2)
val l1: List[Int] = Cons(1,Cons(2,Nil))

scala> val l2: List[Int] = List(5,6)
val l2: List[Int] = Cons(5,Cons(6,Nil))

scala> l1.map(_ + 2)
val res0: List[Int] = Cons(3,Cons(4,Nil))

scala> l1.flatMap((x: Int) => List(x + 2))
val res1: List[Int] = Cons(3,Cons(4,Nil))

scala> val res = l1.map2(l2)(_ + _)
val res: List[Int] =
Cons(6,Cons(7,Cons(7,Cons(8,Nil))))
```

```
scala> val o1 = Some(1)
val o1: Option[Int] = Some(1)

scala> val o2 = Some(2)
val o2: Option[Int] = Some(2)

scala> val o3: Option[Int] = None
val o3: Option[Int] = None

scala> val x = Some(1).map2(o2)(_ + _)
val x: Option[Int] = Some(3)

scala> val y = Some(1).map2(o3)(_ + _)
val y: Option[Int] = None
```

The law of associativity II

Now we can write some code to exemplify this law.

```
trait Monad[F[_]] extends Functor[F]:
    // OMITTED
    def compose[A, B, C](f: A => F[B], g: B => F[C]): A => F[C] = a => f(a).flatMap(g)

val f = (x: Int) => Some(x + 3)
val g = (x: Int) => Some(x * 2)
val h = (x: Int) => Some(x + 5)

import optionMonad.compose

val com1 = compose(compose(f, g), h)
val com2 = compose(f, compose(g, h))

----------

scala> com1(1)
val res0: Option[Int] = Some(13)

scala> com2(1)
val res1: Option[Int] = Some(13)
```

Moverover, we have:

```
scala> compose(f, unit)(1)
val res11: Option[Int] = Some(4)

scala> compose(unit, f)(1)
scala> compose((x: Int) => unit(x), f)(1)
val res12: Option[Int] = Some(4)
```

So, the `compose`, `unit`, and monadic functions constitute a monoid!

Writing functions I

Much like we did for foldable data structures, we can now define functions for monads only once! We have already defined one such function; map2. We will now define two useful functions: sequence and traverse.

```
def sequence[A](fas: List[F[A]]): F[List[A]] =
    fas.foldRight(
      unit(List[A]()),
      (fa, acc) => fa.map2(acc)(Cons(_,_)))

----------

scala> import optionMonad.sequence

scala> sequence(List(Some(1),Some(2),Some(3)))
val res0: Option[List[Int]] = Some(Cons(1,Cons(2,Cons(3,Nil))))
```

Writing functions II

```scala
def traverse[A, B](as: List[A])(f: A => F[B]): F[List[B]] =
    as.foldRight(
        unit(List[B]()),
        (a, acc) => f(a).map2(acc)(Cons(_,_)))

----------

scala> import optionMonad.traverse

scala> traverse(List(1,2,3))((x: Int) => Some(x + 3))
val res1: Option[List[Int]] = Some(Cons(4,Cons(5,Cons(6,Nil))))
```

Summary

- A functor implements `map` such that the structure of the data is preserved. A functor must obey the laws of composition and identity.
- A monad implements flatMap and unit and the implementation must satisfy the laws of associativity and identity.
- "The Monad contract doesn't specify is happening between the lines, only that what whatever happening satisfies the laws of associativity and identity." (Chiusano and Bjarnason, 2015)

- All monads are functors, but not all functors are monads.
- Many functors are also foldable, but there are foldable objects that aren't functors.

Lexicon

- Combinator – combinateur
- Foldable – "pliable"
- Functor – foncteur
- Monad – monade