Monads in Functional Programming using Scala

Marco Mantovani

January 31, 2021

Table of contents

Introduction: Type classes

When we want to extend an existing class with a new interface (and so, possibly, with new functionalities) we have four choices:

1. Extending the class itself, but often we can't modify it (it could be taken from an API)

Introduction: Type classes

When we want to extend an existing class with a new interface (and so, possibly, with new functionalities) we have four choices:

1. Extending the class itself, but often we can't modify it (it could be taken from an API)
2. Creating a new class that extends the original class and the new interface (not always possible)

Introduction: Type classes

When we want to extend an existing class with a new interface (and so, possibly, with new functionalities) we have four choices:

1. Extending the class itself, but often we can't modify it (it could be taken from an API)
2. Creating a new class that extends the original class and the new interface (not always possible)
3. Creating a wrapping class that contains the original class and extends the new interface

Introduction: Type classes

When we want to extend an existing class with a new interface (and so, possibly, with new functionalities) we have four choices:

1. Extending the class itself, but often we can't modify it (it could be taken from an API)
2. Creating a new class that extends the original class and the new interface (not always possible)
3. Creating a wrapping class that contains the original class and extends the new interface
4. Using type classes

The last choice is a classical pattern of FP.

Introduction: Type classes - Example (1)

Let's introduce type classes with an example. We have this scala interface

```
1 trait Summable[T] {
2   def sumElements(list: List[T]): T
3 }
```

and we want that Int and String classes implement it.

Introduction: Type classes - Example (1)

Let's introduce type classes with an example. We have this scala interface

```
1 trait Summable[T] {
2   def sumElements(list: List[T]): T
3 }
```

and we want that Int and String classes implement it.
Of course we cannot change Int or String classes directly (they are into scala's library), so the first option is discarded.

Introduction: Type classes - Example (1)

Let's introduce type classes with an example. We have this scala interface

```
1 trait Summable[T] {
2   def sumElements(list: List[T]): T
3 }
```

and we want that Int and String classes implement it.
Of course we cannot change Int or String classes directly (they are into scala's library), so the first option is discarded.
We could create new classes MyInt and MyStr that extend respectively Int and String but those classes are final, so we cannot work in this way.

Introduction: Type classes - Example (1)

Let's introduce type classes with an example. We have this scala interface

```
1 trait Summable[T] {
2   def sumElements(list: List[T]): T
3 }
```

and we want that Int and String classes implement it.
Of course we cannot change Int or String classes directly (they are into scala's library), so the first option is discarded.
We could create new classes MyInt and MyStr that extend respectively Int and String but those classes are final, so we cannot work in this way.
Surely we can create wrapping classes MyInt and MyStr, but in this way we have now two definitions of Int and of String and it could be messy.

Introduction: Type classes - Example (2)

Let's try type classes! We start creating two implicit objects that implement the interface as follows:

```
1 implicit object IntSummable extends Summable[Int] {
2   def sumElements(list: List[Int]): Int = list.sum
3 }
4 implicit object StringSummable extends Summable[String] {
5   def sumElements(list: List[String]): String =
6     list.mkString("")
7 }
```

Introduction: Type classes - Example (2)

Let's try type classes! We start creating two implicit objects that implement the interface as follows:

```
implicit object IntSummable extends Summable[Int] {
  def sumElements(list: List[Int]): Int = list.sum
}
implicit object StringSummable extends Summable[String] {
  def sumElements(list: List[String]): String =
    list.mkString("")
}
```

In this way we can now define a generic method that uses sumElements

```
def sumAll[T](l: List[T])(implicit summable: Summable[T]): T
    =
  summable.sumElements(l)
```

Introduction: Type classes - Example (3)

The advantage of this approach is that we can use sumAll method only with Int and String (all the other types will create a compile error) and, because of implict, we don't have to pass summable object to sumAll.

```
1 sumAll(List(1,2,3)) // 6
2 sumAll(List("Scala ", "is ", "awesome")) // "Scala is
      awesome"
3 sumAll(List(true, true, false)) // COMPILE ERROR
```

Introduction: Type classes - Example (4)

Through extension method we could also use sumAll as method in List of summable:

```
1 implicit class ListSummable[A](l: List[A])(implicit summable
      : Summable[A]) {
2   def sumAll: A = summable.sumElements(l)
3 }
```

Introduction: Type classes - Example (4)

Through extension method we could also use sumAll as method in List of summable:

```
1 implicit class ListSummable[A](l: List[A])(implicit summable
      : Summable[A]) {
2   def sumAll: A = summable.sumElements(l)
3 }
```

```
1 List(1,2,3).sumAll //6
2 List("Scala ", "is ", "awesome").sumAll //"Scala is awesome"
3 List(true, true, false).sumAll // COMPILE ERROR
```

Functors

Let's start with map function:

```
1 def fmap[A, B](fa: F[A])(f: A => B): F[B]
```

Functors

Let's start with map function:

```
1 def fmap[A, B](fa: F[A])(f: A => B): F[B]
```

Definition

A functor is a class that has the map method which respects the following laws:

1. **Identity:** given the identity function *id* and a functor *obj* we have that $map(obj)(id) = obj$.

Functors

Let's start with map function:

```
1 def fmap[A, B](fa: F[A])(f: A => B): F[B]
```

Definition

A functor is a class that has the map method which respects the following laws:

1. **Identity:** given the identity function *id* and a functor *obj* we have that $map(obj)(id) = obj$.
2. **Composition:** given two functions f and g and a functor *obj* we have that $map(map(obj)(f))(g) = map(obj)(g \circ f)$

We could test these laws through PBT ScalaCheck.

Functors - Implementation (1)

We'll use type classes to implement functors.
Functors can be represented with the following interface

```
1 trait Functor[F[_]] {
2   def fmap[A, B](fa: F[A])(f: A => B): F[B]
3 }
```

Functors - Implementation (2)

Let's extend List and Option classes to be functors.

```
implicit object listFunctor: Functor[List] {
  def fmap[A, B](fa: List[A])(f: A => B): List[B] = fa.map(f
    )
}
implicit object optionFunctor: Functor[Option] {
  def fmap[A, B](fa: Option[A])(f: A => B): Option[B] =
  fa match {
    case None => None
    case Some(x) => Some(f(x))
  }
}
```

The first case is easy because List class has already map function (to be honestly, Option has map method too).

Functors - Implementation (3)

As before, we can create a generic method that applies map:

```
def applyMap[F[_], A, B](a: F[A])(f: A => B)(implicit
    functor: Functor[F]): F[B] = {
  functor.fmap(a)(f)
}
applyMap(List(1, 2, 3))(_ + 1) //List(2, 3, 4)
applyMap(Some(1) : Option[Int])(_ + 1) // Some(2)
applyMap(Set(true, true, false))(!_) // COMPILE ERROR
```

Through extension method we could also use applyMap as method of functors:

```
implicit class FunctorMethod[F[_], A, B](a: F[A])(implicit
    functor: Functor[F]) {
  def applyMap(f: A => B): F[B] = functor.fmap(a)(f)
}
List(1, 2, 3).applyMap(_ + 1) //List(2, 3, 4)
(Some(1):Option[Int]).applyMap(_ + 1) // Some(2)
Set(true, true, false).applyMap(!_) // COMPILE ERROR
```

Why are functors useful?

Functors allow to avoid boilerplate code. Let's take Option type. Normally before applying a function on Option we should check if that value is either Some or None through pattern matching:

```
val a : Option[Int] = Some(20)
def f(a : Int) : Int = if (a % 2 == 0) a / 2 else a - 1
val ris = (a match {
  case None => None
  case Some(b) => Some(f(b))
}) //Some(10)
```

Functors reduce code:

```
val a : Option[Int] = Some(10)
def f(a : Int) : Int = if (a % 2 == 0) a / 2 else a - 1
val ris = a.applyMap(f) //Some(10)
```

Monads

Again, let's start from bind (also called flatMap) and unit methods:

```
1 def bind[A, B](fa: M[A])(f: A => M[B]): M[B]
2 def unit[A](a : A) : M[A]
```

Monads

Again, let's start from bind (also called flatMap) and unit methods:

```
1 def bind[A, B](fa: M[A])(f: A => M[B]): M[B]
2 def unit[A](a : A) : M[A]
```

Definition

A monad is a class that has bind and unit methods which respect the following laws:

1. **Left identity:** given a function f and an object obj we have that $bind(unit(obj))(f) = f(obj)$.

Monads

Again, let's start from bind (also called flatMap) and unit methods:

```
1 def bind[A, B](fa: M[A])(f: A => M[B]): M[B]
2 def unit[A](a : A) : M[A]
```

Definition

A monad is a class that has bind and unit methods which respect the following laws:

1. **Left identity:** given a function f and an object obj we have that $bind(unit(obj))(f) = f(obj)$.
2. **Right identity:** given a monad obj we have that $bind(obj)(unit) = obj$

Monads

Again, let's start from bind (also called flatMap) and unit methods:

```
1 def bind[A, B](fa: M[A])(f: A => M[B]): M[B]
2 def unit[A](a : A) : M[A]
```

Definition

A monad is a class that has bind and unit methods which respect the following laws:

1. **Left identity:** given a function f and an object obj we have that $bind(unit(obj))(f) = f(obj)$.
2. **Right identity:** given a monad obj we have that $bind(obj)(unit) = obj$
3. **Associativity:** given two functions f and g and a monad obj we have that $bind(bind(obj)(f))(g) = bind(obj)(x \Rightarrow flatMap(f(x))(g))$

We could test these laws through PBT ScalaCheck.

Monads - Implementation (1)

We'll use type classes to implement monads.
Monads can be represented with the following interface

```
1 trait Monad[M[_]] {
2   def bind[A, B](fa: M[A])(f: A => M[B]): M[B]
3   def unit[A](a : A) : M[A]
4 }
```

Monads - Implementation (2)

Let's extend List and Option classes to be monads.

```
1  implicit object ListMonad extends Monad[List] {
2    override def bind[A, B](fa: List[A])(f: A => List[B]):
       List[B] =
3      fa.flatMap(f)
4    override def unit[A](a: A): List[A] = List(a)
5  }
6  implicit object OptionMonad extends Monad[Option] {
7    override def bind[A, B](fa: Option[A])(f: A => Option[B]):
       Option[B] =
8      fa match {
9        case None => None
10       case Some(a) => f(a)
11     }
12   override def unit[A](a: A): Option[A] = Some(a)
13 }
```

The first case is easy because List class has already flatMap function (to be honestly, Option has flatMap method too).

Monads - Implementation (3)

As before, we can create a generic method that applies bind:

```
1 def bind[M[_], A, B](a: M[A])(f: A => M[B])(implicit monad:
      Monad[M]): M[B] = {
2   monad.bind(a)(f)
3 }
4 bind(List(1,2,3))((x : Int) => List(x + 1)) //List(2, 3, 4)
5 bind(Some(1) : Option[Int])((x : Int) => Some(x + 1)) //
      Some(2)
6 bind(Set(true, true, false))((x : Boolean) => Set(!x)) //
      COMPILE ERROR
```

Monads - Implementation (4)

Through extension method we could also use bind as method in monads:

```
1 implicit class MonadMethods[M[_], A, B](a: M[A])(implicit
      monad: Monad[M]) {
2   def bind(f: A => M[B]): M[B] = monad.bind(a)(f)
3   def unit : A => M[A] = monad.unit
4 }
5 List(1,2,3).bind((x : Int) => List(x + 1)) //List(2, 3, 4)
6 (Some(1) : Option[Int]).bind((x : Int) => Some(x + 1)) //
      Some(2)
7 Set(true, true, false).applyBind((x : Boolean) => Set(!x))
      // COMPILE ERROR
```

Differences between Functors and Monads (1)

Why should we use monads instead of functors?
Let's start from a simple example: concatenation of map function.

```
1 val a : Option[Int] = Some(20)
2 val temp = a.applyMap(_ + 1) //Some(2)
3 temp.applyMap(_ + 1) // Some(3)
```

This example works nicely. We start from an Option of Int, after the first step we have an Option of Int on which we apply again the same map function and we finally get an Option of Int.

Differences between Functors and Monads (2)

But, let's try with a new function:

```
1 val a : Option[Int] = Some(20)
2 def f(x:Int) : Option[Int] = if (x == 0) None else Some(2 /
      x)
3 val temp1 = a.applyMap(f) // return Some(Some(0))
4 temp1.applyMap(f)// COMPILE ERROR
```

Here we stuck because, after the first step, we got an Option of Option of Int. We could manage this problem and adjust the second function to work with Option of Option of Int, but in this way we would get an Option of Option of Option of Int as result type and we could continue getting more and more layers of Option.

Differences between Functors and Monads (2)

But, let's try with a new function:

```
1 val a : Option[Int] = Some(20)
2 def f(x:Int) : Option[Int] = if (x == 0) None else Some(2 /
      x)
3 val temp1 = a.applyMap(f) // return Some(Some(0))
4 temp1.applyMap(f)// COMPILE ERROR
```

Here we stuck because, after the first step, we got an Option of Option of Int. We could manage this problem and adjust the second function to work with Option of Option of Int, but in this way we would get an Option of Option of Option of Int as result type and we could continue getting more and more layers of Option.
Here monads come to help!

```
1 val temp = a.bind(f) // return Some(0)
2 temp.bind(f) //None
```

We could continue to apply bind endless, without change the return type.

Sitography and Source codes

Sitography

- Rock the JVM - Type classes
- Rock the JVM - Functor
- dcapwell - Functor
- adit.io - Functor & Monad

Source codes

- Scala sources are available at my GitHub.

The end

Thanks!