Functors, Monads and Transformer in Impure Functional Programming using Scala

Marco Mantovani

February 9, 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 essentially four choices:

1. Extending the class itself (not always possible, e.g. API classes)

Introduction: Type classes

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

1. Extending the class itself (not always possible, e.g. API classes)
2. Creating a new class that extends the original class and the new interface (not always possible, e.g. final classes)

Introduction: Type classes

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

1. Extending the class itself (not always possible, e.g. API classes)
2. Creating a new class that extends the original class and the new interface (not always possible, e.g. final classes)
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 essentially four choices:

1. Extending the class itself (not always possible, e.g. API classes)
2. Creating a new class that extends the original class and the new interface (not always possible, e.g. final classes)
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

```scala
trait Summable[T] {
  def sumElements(list: List[T]): T
}
```

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

```scala
trait Summable[T] {
  def sumElements(list: List[T]): T
}
```

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

Introduction: Type classes - Example (1)

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

```scala
trait Summable[T] {
  def sumElements(list: List[T]): T
}
```

and we want that Int and String classes implement it.
Of course we cannot change Int or String classes directly (they are into scala API), 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

```scala
trait Summable[T] {
  def sumElements(list: List[T]): T
}
```

and we want that Int and String classes implement it.
Of course we cannot change Int or String classes directly (they are into scala API), 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 Int and List of String:

```
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 Int and List of String:

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

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

Functors

Let's start with map function:

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

Functors

Let's start with map function:

```
1 def map[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 map[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 $map(map(obj)(f))(g) = map(obj)(g \circ f)$

Functors - Implementation (1)

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

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

Functors - Implementation (2)

Let's create List and Option functors.

```
implicit object listFunctor: Functor[List] {
  def map[A, B](fa: List[A])(f: A => B): List[B] = fa.map(f)
}
implicit object optionFunctor: Functor[Option] {
  def map[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.map(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 ToFunctor[F[_], A, B](a: F[A])(implicit
    functor: Functor[F]) {
  def applyMap(f: A => B): F[B] = functor.map(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
```

Functors Laws

Functors laws are expressed in scala in this way:

```scala
def identity(fa: F[A]): Boolean = {
  def id(x: A): A = x
  fa.applyMap(id) == fa
}
def composition(fa: F[A])(f: A => A, g: A => A): Boolean =
  fa.applyMap(f).applyMap(g) == fa.applyMap(g compose f)
```

We could test these laws through PBT ScalaCheck.

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 functor 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 functor 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 functor 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 $bind(bind(obj)(f))(g) = bind(obj)(x \Rightarrow flatMap(f(x))(g))$

Monads - Implementation (1a)

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

```
1 trait Monad[M[_]] extends Functor[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 (1b)

Monads are functors, so have *map* function. Often monads have also another method: *join*. There are standard implementations of *map* and *join* through *bind* and *unit*:

```
1    trait Monad[M[_]] extends Functor[M] {
2      def bind[A, B](fa: M[A])(f: A => M[B]): M[B]
3      def unit[A](a: A): M[A]
4      def join[A](ma: M[M[A]]): M[A] = {
5        bind(ma)(m => m)
6      }
7      override def map[A, B](fa: M[A])(f: A => B): M[B] = {
8        bind(fa)(a => unit(f(a)))
9      }
10   }
11
```

It's possible demonstrate that $bind(obj)(f)$ correspond to $join(map(obj)(f))$.

Monads - Implementation (2)

Let's create List and Option 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 bind(List(1,2,3))((x : Int) => List(x + 1)) //List(2, 3, 4)
4 bind(Some(1) : Option[Int])((x : Int) => Some(x + 1)) //
    Some(2)
5 bind(Set(true, true, false))((x : Boolean) => Set(!x)) //
    COMPILE ERROR
```

Monads - Implementation (4)

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

```
1  implicit class ToMonad1[M[_], A, B](a: M[A])(implicit monad:
       Monad[M]) {
2    lazy val applyUnit: A => M[A] = monad.unit
3    def applyBind(f: A => M[B]): M[B] = monad.bind(a)(f)
4    def applyMap(f: A => B): M[B] = monad.map(a)(f)
5  }
6  implicit class ToMonad2[M[_], A](a: M[M[A]])(implicit monad:
       Monad[M]) {
7    lazy val applyJoin: M[A] = monad.join(a)
8  }
9  List(1,2,3).bind((x : Int) => List(x + 1)) //List(2, 3, 4)
10 (Some(1) : Option[Int]).bind((x : Int) => Some(x + 1)) //
       Some(2)
11 Set(true, true, false).applyBind((x : Boolean) => Set(!x))
       // COMPILE ERROR
```

Monads Laws

Monads laws are expressed in scala in this way:

```
1 def leftIdentity(a: A)(f: A => M[A]): Boolean =
2   monad.unit(a).applyBind(f) == f(a)
3 def rightIdentity(t: M[A]): Boolean =
4   t.applyBind(t.applyUnit) == t
5 def associativity(t: M[A])(f: A => M[A], g: A => M[A]):
    Boolean =
6   t.applyBind(f).applyBind(g) == t.applyBind((x: A) => f(x).
    applyBind(g))
```

We could test these laws through PBT ScalaCheck.

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(22)
3 temp.applyMap(_ + 1) // Some(23)
```

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.

Functors and Monads in Scala (1)

Scala official API doesn't contain Functor and Monad trait, however cite then along the documentation. For example, in Option class documentation we could find:

> *The most idiomatic way to use an Option instance is to treat it as a collection or* ***monad*** *and use map, flatMap*[1] *[...]*

[1] in scala *bind* is called *flatMap*

Functors and Monads in Scala (2)

Scala offers syntactic sugar to simplify call to *map* and *flatMap* methods through *for – expression*.

```
val temp = for {
  i <- Option(1)
  j <- Option(2)
  k <- Option(3)
} yield i + j + k //Some(6)
```

is equivalent to

```
 val temp: Option[Int] = Some(1).flatMap {
  (i: Int) => Some(2).flatMap {
    (j: Int) => Some(3).map {
      (k: Int) => i + j + k
    }
  }
} //Some(6)
```

Functors compose with Functors

Let's start from an example

```
1 val v = List(Option(1), None, Option(3))
2 val f1: Int => String = _ + "a"
3 v.applyMap(x => x.applyMap(f1)) //List(Some(1a), None, Some
      (3a))
```

If we have a functors of a functor we can compose *map* in a simple (and intuitive) way: $map(v)(x => map(x)(f1))$.

Monads don't compose with monads (1)

Again, let's start from an example

```
1 val v = List(Option(1), None, Option(3))
2 val f2: Int => List[Option[String]] = x => List(Some(x + "a"
    ))
3 v.applyBind(x => x.applyBind(f2)) // COMPILE ERROR
```

If we have a monads of a monads we **cannot** compose *bind* in a simple (and intuitive) way: $bind(v)(x => bind(x)(f2))$. We would have type mismatch error.

Monads don't compose with monads (2)

A solution could be

```
v.applyBind {
  case None => default
  case Some(x) => f2(x)
} //List(Some(1a), None, Some(3a))
```

This conduces to boilerplate code (again...).
Note also that now Option could be or not a monad because we don't use bind or unit of Option. So the problem is related to monad which internal type is generic.

Transformer

Monad transformers are monad which internal type is generic (with generic instantiated with non-generic type), so they are monad of type $M1[M2[A]]$ with an additional method *lift*.

Transformer

Monad transformers are monad which internal type is generic (with generic instantiated with non-generic type), so they are monad of type $M1[M2[A]]$ with an additional method *lift*. They have the same laws of monad, plus:

1. Law 1: given an object *obj* we have that $lift(unit(obj)) = unit(obj)$ where unit call on the left is monad method and unit on the right is transformer method.

Transformer

Monad transformers are monad which internal type is generic (with generic instantiated with non-generic type), so they are monad of type $M1[M2[A]]$ with an additional method *lift*. They have the same laws of monad, plus:

1. Law 1: given an object *obj* we have that $lift(unit(obj)) = unit(obj)$ where unit call on the left is monad method and unit on the right is transformer method.
2. Law 2: given a function f and a monad m we have that $lift(bind(m)(f)) = bind(lift(m))(lift \circ f)$ where bind call on the left is monad method and bind on the right is transformer method.

Transformer - Implementation (1)

We'll use type classes to implement monad transformers.

```
1  trait Transformer[M1[_], M2[_]] {
2    implicit def monadM1: Monad[M1]
3    def bind[A, B](fa: M1[M2[A]])(f: A => M1[M2[B]]): M1[M2[B
       ]]
4    def unit[A](a: A): M1[M2[A]]
5    def lift[A](fa: M1[A]): M1[M2[A]]
6    def join[A](fa: M1[M2[M1[M2[A]]]]): M1[M2[A]] =
7      bind(fa)(m => m)
8    def map[A, B](fa: M1[M2[A]])(f: A => B): M1[M2[B]] =
9      bind(fa)(a => unit(f(a)))
10 }
```

Transformer - Implementation (2)

Let's create Option transformer, called OptionT.

```
trait OptionT[M1[_]] extends Transformer[M1, Option] {
  override def bind[A, B](fa: M1[Option[A]])(f: A => M1[
    Option[B]]): M1[Option[B]] = {
    monadM1.bind(fa) {
      case None => monadM1.unit(None)
      case Some(a) => f(a)
    }
  }
  override def lift[A](fa: M1[A]): M1[Option[A]] =
    monadM1.bind(fa)(a => unit(a))
  override def unit[A](a: A): M1[Option[A]] = monadM1.unit(
    Some(a))
}
```

We can instantiate it on list, provided implicit ListMonad, in this way:

```
implicit object ListOptionT extends OptionT[List] {
  override implicit val monadM1: Monad[List] = implicitly
}
```

Transformer - Implementation (3)

Through extension method we could also use transformer function as method in transformer:

```
1  implicit class ToTransformer1[M1[_], M2[_], A, B](a: M1[M2[A
       ]])(implicit transformer: Transformer[M1, M2]) {
2    lazy val applyUnitT: A => M1[M2[A]] = transformer.unit
3    def applyMapT(f: A => B): M1[M2[B]] = transformer.map(a)(f
       )
4    def applyBindT(f: A => M1[M2[B]]): M1[M2[B]] = transformer
       .bind(a)(f)
5  }
6  implicit class ToTransformer2[M1[_], M2[_], A, B](fa: M1[A])
       (implicit transformer: Transformer[M1, M2]) {
7    lazy val applyLiftT: M1[M2[A]] = transformer.lift(fa)
8  }
9  implicit class ToTransformer3[M1[_], M2[_], A, B](fa: M1[M2[
       M1[M2[A]]]])(implicit transformer: Transformer[M1, M2])
       {
10   lazy val applyJoinT: M1[M2[A]] = transformer.join(fa)
11 }
```

Transformer Laws

Transformer laws are expressed in scala in this way:

```
1  def law1(a: A): Boolean =
2    transformer.monadM1.unit(a).applyLiftT == transformer.unit
       (a)
3  def law2(m: M1[A])(f: A => M1[A]): Boolean =
4    transformer.lift(transformer.monadM1.bind(m)(f)) ==
       transformer.lift(m).applyBindT(transformer.lift[A] _
       compose f)
5  def leftIdentity(a: A)(f: A => M1[M2[A]]): Boolean =
6    transformer.unit(a).applyBindT(f) == f(a)
7  def rightIdentity(t: M1[M2[A]]): Boolean =
8    t.applyBindT(t.applyUnitT) == t
9  def associativity(t: M1[M2[A]])(f: A => M1[M2[A]], g: A =>
       M1[M2[A]]): Boolean =
10   t.applyBindT(f).applyBindT(g) == t.applyBindT((x: A) => f(
       x).applyBindT(g))
```

We could test these laws through PBT ScalaCheck.

Source codes

Source codes

- Scala sources are available at my GitHub.

The end

Thanks!