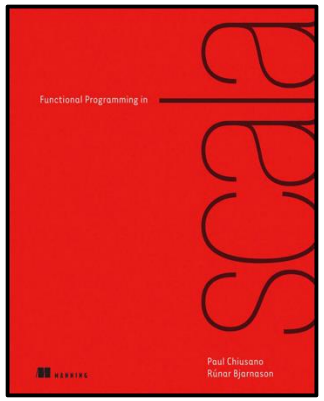
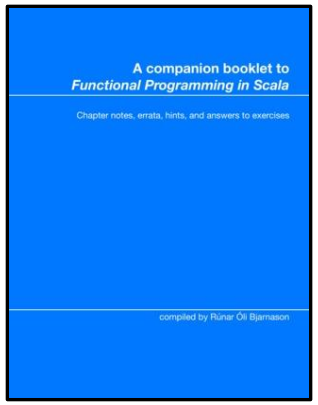
Monads do not Compose

not in a generic way - there is no general way of composing monads

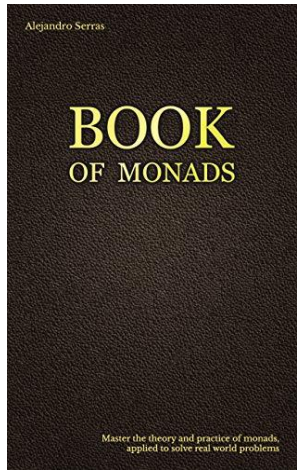
inspired by (and with excerpts from)



Functional Programming in Scala



A companion booklet to Functional Programming in Scala



slides by  @philip_schwarz

A **Monad** is both a **Functor** and an **Applicative**.

If you want to know more about **Applicative** then see the following

slideshare @philip_schwarz

https://www.slideshare.net/pjschwarz/ https://www.slideshare.net/pjschwarz/applicative-functor-116035644

A **Monad** is both a **Functor** and an **Applicative**

Here we define a **Monad** in terms of **unit** and **flatMap**

Yes, **sequence** and **traverse** really belong on a **Traverse** trait, but they are not the main focus here, just examples of functions using **Applicative** functions **unit** and **map2**.

Functor

map

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

def map[A, B](m: F[A], f: A ⇒ B): F[B]
```

Applicative

unit
map2

traverse
sequence

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

def unit[A](a: ⇒ A): F[A]
def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) ⇒ C): F[C]

override def map[A,B](fa: F[A])(f: A ⇒ B): F[B] = map2(fa,unit(()))((a,_) ⇒ f(a))

def traverse[A,B](as: List[A])(f: A ⇒ F[B]): F[List[B]] = as.foldRight(unit(List[B]()))((a, fbs) ⇒ map2(f(a), fbs)(_::_)
def sequence[A](lfa: List[F[A]]): F[List[A]] = traverse(lfa)(fa ⇒ fa)
```

Monad

flatMap

compose
join

```
trait Monad[F[_]] extends Applicative[F]

def flatMap[A,B](ma: F[A])(f: A ⇒ F[B]): F[B]

override def map[A,B](m: F[A])(f: A ⇒ B): F[B] = flatMap(m)(a ⇒ unit(f(a)))
override def map2[A,B,C](ma: F[A], mb: F[B])(f: (A, B) ⇒ C): F[C] = flatMap(ma)(a ⇒ map(mb)(b ⇒ f(a, b)))

def compose[A,B,C](f: A ⇒ F[B], g: B ⇒ F[C]): A ⇒ F[C] = a ⇒ flatMap(f(a))(g)
def join[A](mma: F[F[A]]): F[A] = flatMap(mma)(ma => ma)
```

listMonad

```
val listMonad = new Monad[List] {

override def unit[A](a: ⇒ A)= List(a)
override def flatMap[A,B](ma: List[A])(f: A ⇒ List[B]) = ma flatMap f
```

Functors compose.

On the next slide we look at a couple of examples.

If you would like to know more about **Functor composition** then see the following

slideshare @philip_schwarz

https://www.slideshare.net/pjschwarz/functor-composition

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

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

  def compose[G[_]](G:Functor[G]):Functor[λ[α=>F[G[α]]]] = {
    val self = this
    new Functor[λ[α => F[G[α]]]] {
      override def map[A, B](fga:F[G[A]])(f:A=>B):F[G[B]] =
        self.map(fga)(ga => G.map(ga)(f))
    }
  }
}
```

Functors compose

The map function of the composite Functor is the composition of the map functions of the functors being composed.

using https://github.com/non/kind-projector
allows us to simplify type lambda ({type f[α] = F[G[α]]})#f
to this: λ[α => F[G[α]]]

```scala
implicit val listFunctor = new Functor[List] {
  def map[A, B](fa: List[A])(f: A => B): List[B] = fa map f
}

implicit val optionFunctor = new Functor[Option] {
  def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa map f
}
```

We first compose the List Functor with the Option Functor. This allows us to map a function over lists of options. Then we do the opposite: we compose the Option Functor with the List Functor. This allows us to map a function over an optional list.

```scala
// Functor[List[Option]] = Functor[List] compose Functor[Option]
val optionListFunctor = listFunctor compose optionFunctor

assert(optionListFunctor.map(List(Some(1),Some(2),Some(3)))(double) == List(Some(2),Some(4),Some(6)))
```

```scala
// Functor[Option[List]] = Functor[Option] compose Functor[List]
val listOptionFunctor = optionFunctor compose listFunctor

assert(listOptionFunctor.map(Some(List(1,2,3)))(double) == Some(List(2,4,6)))
```

```scala
val double: Int => Int = _ * 2
```

@philip_schwarz

Applicatives also **compose**.

We can compose **Applicatives** using a similar technique to the one we used to compose **Functors**.

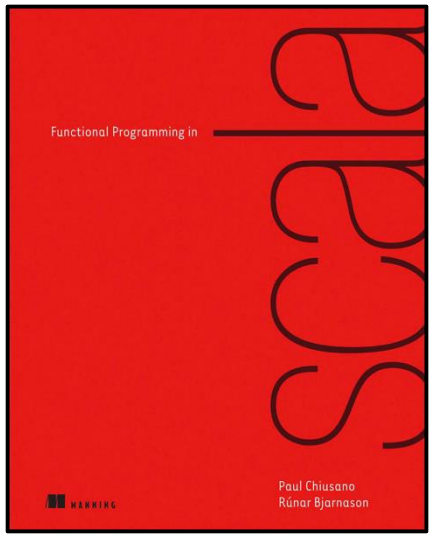
Let's look at **FPiS** to see how it is done.

EXERCISE 12.9

Hard: **Applicative functors** also compose another way! If `F[_]` and `G[_]` are **applicative functors**, then so is `F[G[_]]`.

Implement this function:

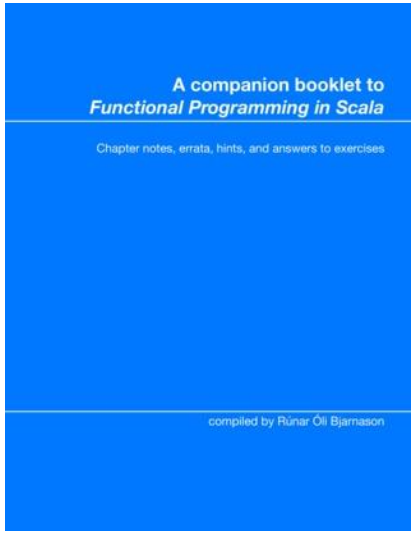
```
  def compose[G[_]](G: Applicative[G]): Applicative[({type f[x] = F[G[x]]})#f]
```



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

ANSWER TO EXERCISE 12.9

```
def compose[G[_]](G: Applicative[G]): Applicative[({type f[x] = F[G[x]]})#f] = {
  val self = this
  new Applicative[({type f[x] = F[G[x]]})#f] {
    def unit[A](a: => A) = self.unit(G.unit(a))
    override def map2[A,B,C](fga: F[G[A]], fgb: F[G[B]])(f: (A,B) => C) =
        self.map2(fga, fgb)(G.map2(_,_)(f))
  }
}
```



(by Runar Bjarnason)
@runarorama

In the next two slides we look at a couple of examples of **composing Applicatives**.

```scala
trait Applicative[F[_]] extends Functor[F] {

  def unit[A](a: => A): F[A]
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]

  def map[A,B](fa: F[A])(f: A => B): F[B] =
    map2(fa, unit(()))((a, _) => f(a))

  def compose[G[_]](G: Applicative[G]): Applicative[λ[α => F[G[α]]]] = {
    val self = this
    new Applicative[λ[α => F[G[α]]]] {
      def unit[A](a: => A): F[G[A]] = self.unit(G.unit(a))
      override def map2[A,B,C](fga:F[G[A]],fgb:F[G[B]])(f:(A,B)=>C):F[G[C]] =
        self.map2(fga, fgb)(G.map2(_,_)(f))
    }
  }
}
```

Applicatives compose

The **unit** function of the **composite Applicative** first lifts its parameter into **inner Applicative G** and then lifts the result into **outer Applicative F**.

The **map2** function of the **composite Applicative** is the **composition** of the **map2** functions of the **applicatives** being **composed**. It uses the **map2** function of **F** to break through the **outer Applicative** and the **map2** function of **G** to break through the **inner Applicative**. This is how it manages to break through the **two layers** of **Applicative** in the **composite**.

Let's create an **Applicative** instance for **Option** and one for **List**.

```scala
val optionApplicative = new Applicative[Option] {
  def unit[A](a: => A): Option[A] = Some(a)
  def map2[A,B,C](fa:Option[A],fb:Option[B])
          (f:(A,B)=>C):Option[C] =
    (fa, fb) match {
      case (Some(a), Some(b)) => Some(f(a,b))
      case _ => None
    }
}
```

```scala
val listApplicative = new Applicative[List] {
  def unit[A](a: => A): List[A] = List(a)
  def map2[A, B, C](fa:List[A],fb:List[B])
          (f:(A,B)=>C):List[C] =
    for {
      a <- fa
      b <- fb
    } yield f(a, b)
}
```

Applicatives compose

```
// Applicative[List] compose Applicative[Option] = Applicative[List[Option]
val optionListApplicative = listApplicative compose optionApplicative

assert( optionApplicative.map2( Option(1), Option(2) )(add)
        == Option(3) )

assert( listApplicative.map2( List(1,2,3), List(4,5,6))(add)
        == List(5,6,7,6,7,8,7,8,9))

assert(
  (optionListApplicative map2(
    List( Option(1), Option(2) ),
    List( Option(3), Option(4) )
  )(add)
  == List( Option(4), Option(5), Option(5), Option(6) ) )

assert( optionListApplicative.map( List( Option(1), Option(2) ) )(double)
        == List( Option(2), Option(4) ) )
```

Let's create an Applicative that is the composition of our List Applicative and our Option Applicative. Let's then have a go at mapping a binary function over two Lists of Options.

```
val add: (Int,Int) => Int = _ + _
```

```
val double: Int => Int = _ * 2
```

```
// Applicative[Option] compose Applicative[List] = Applicative[Option[List]]
val listOptionApplicative = optionApplicative compose listApplicative

assert( listOptionApplicative.map2( Option( List( 1,3,5 ) ), Option( List(2,4,6) ) )(add)
        == Option( List(3,5,7,5,7,9,7,9,11) ) )

assert( listOptionApplicative.map( Option( List(1,2,3) ) )(double)
        == Option( List(2,4,6) ) )
```

Now lets create the **opposite** composite Applicative and have a go at mapping a binary function over two Optional Lists.

What about a **Monad**? Is it possible for the **Monad** trait to have a **compose** function that takes any other **Monad** and returns a **composite Monad**?

Let's try implementing such a **compose** function.

Do Monads compose ?

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

  def flatMap[A, B](ma: F[A])(f: A => F[B]): F[B]

  override def map[A, B](m: F[A])(f: A => B): F[B] =
    flatMap(m)(a => unit(f(a)))

  override def map2[A, B, C](ma: F[A], mb: F[B])(f: (A, B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))

  def compose[G[_]](G: Monad[G]): Monad[λ[α => F[G[α]]]] = {
    val self = this
    new Monad[λ[α => F[G[α]]]] {
      def unit[A](a: => A): F[G[A]] = self.unit(G.unit(a))
      def flatMap[A, B](fga: F[G[A]])(f: A => F[G[B]]): F[G[B]] = {
        self.flatMap(fga) { ga =>
          G.flatMap(ga) { a =>
            val fgb: F[G[B]] = f(a)
            ??? // this inner flatMap must return G[_] but all we have is an F[G[_]
            ??? // to obtain a G[_] we'd have to swap F and G and return G[F[_]]
          }
          ??? // this outer flatMap must return F[G[_]] but all we have is a G[_]
          ??? // had we been able to swap F and G in the inner flatMap we would have a G[F[_]]
          ??? // so we would have to swap G and F again to get an F[G[_]]
        }
      }
    }
  }
```

The **unit** function of the **composite Monad** first lifts its parameter into **inner Monad G** and then lifts the result into **outer Monad F**.

The **flatMap** function of the **composite Monad** needs to be the **composition** of the **flatMap** functions of the **monads** being **composed**. It needs to use the **flatMap** function of **F** to break through the **outer Monad** and the **flatMap** function of **G** to break through the **inner Monad**. It would then be able to break through the **two layers** of **Monad** in the **composite**.

But we are not able to write suitable functions to pass to the flatMap functions of the inner and outer Monads.

@philip_schwarz

Do Monads compose ?

```
trait Monad[F[_]] extends Applicative[F] {

  def join[A](mma: F[F[A]]): F[A]

  def flatMap[A, B](ma: F[A])(f: A => F[B]): F[B] = join(map(ma)(f))

  override def map[A, B](m: F[A])(f: A => B): F[B] =
    flatMap(m)(a => unit(f(a)))

  override def map2[A, B, C](ma: F[A], mb: F[B])(f: (A, B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))

  def compose[G[_]](G: Monad[G]): Monad[λ[α => F[G[α]]]] = {
    val self = this
    new Monad[λ[α => F[G[α]]]] {
      def unit[A](a: => A): F[G[A]] = self.unit(G.unit(a))
      def join[A](fgfga: F[G[F[G[A]]]]): F[G[A]] = {
        self.join(G.join(fgfga))  // does not compile - it is impossible to flatten fgfga
      }
    }
  }
}
```

It is easier to see what the problem is if we change the **Monad** trait so that it is defined in terms of **map**, **join** and **unit** rather than in terms of **flatMap** and **unit**.

We then have to implement a **join** function for the **composite Monad** (rather than a **flatMap** function).

The **join** function has to turn an `F[G[F[G[A]]]]` into an `F[G[A]]`, which is not possible using the **join** functions of the **Monad**s being **composed**.

Do Monads compose ?

```
trait Monad[F[_]] extends Applicative[F] {

  def join[A](mma: F[F[A]]): F[A]

  def flatMap[A, B](ma: F[A])(f: A => F[B]): F[B] = join(map(ma)(f))

  override def map[A, B](m: F[A])(f: A => B): F[B] =
    flatMap(m)(a => unit(f(a)))

  override def map2[A, B, C](ma: F[A], mb: F[B])(f: (A, B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))

  def compose[G[_]](G: Monad[G]): Monad[λ[α => F[G[α]]]] = {
    val self = this
    new Monad[λ[α => F[G[α]]]] {
      def unit[A](a: => A): F[G[A]] = self.unit(G.unit(a))
      def join[A](fgfga: F[G[F[G[A]]]]): F[G[A]] = {
        self.join(G.join(fgfga)) // does not compile - it is impossible to flatten fgfga
        val ffgga: F[F[G[G[A]]]] = ???  // if it were possible to rearrange fgfga into ffgga
        val fgga: F[G[G[A]]] = self.join(ffgga) // then we could flatten ffgga to fgga
        val fga: F[G[A]] = self.map(fgga)(gga => G.join(gga)) // and then flatten fgga to fga
        fga
      }
    }
  }
```

If **join** were able to turn its parameter F[G[F[G[A]]]] into F[F[G[G[A]]]] then it would be able to **flatten** the two Fs and the 2 Gs and be left with the desired F[G[A]].

But is doing something like that possible and in which cases?

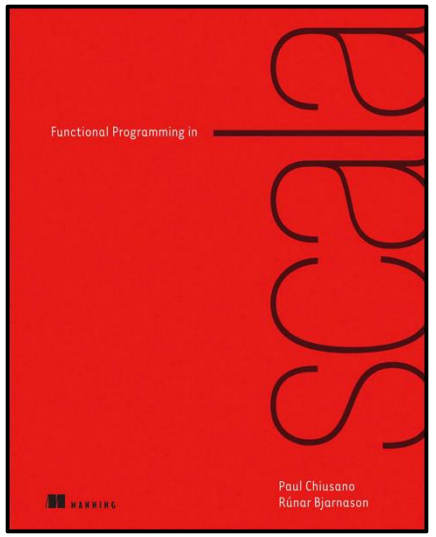
@philip_schwarz

In the next two slides we look at what **FPiS** says about **Monad composition**, and we'll see that it discusses turning `F[G[F[G[A]]]]` into `F[F[G[G[A]]]]`.

EXERCISE 12.11

Try to write `compose` on **Monad**. It's **not possible**, but it is instructive to attempt it and understand why this is the case.

```
def compose[G[_]](G: Monad[G]): Monad[({type f[x] = F[G[x]]})#f]
```



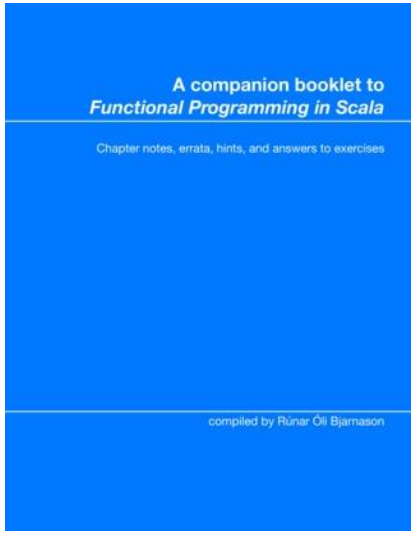
Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

Answer to Exercise 12.11

You want to try writing **flatMap** in terms of **Monad**[F] and **Monad**[G].

```
def flatMap[A,B](mna: F[G[A]])(f: A => F[G[B]]): F[G[B]] =
  self.flatMap(na => G.flatMap(na)(a => ???))
```

Here all you have is f, which returns an F[G[B]]. For it to have the appropriate type to return from the argument to G.**flatMap**, you'd need to be able to "**swap**" the F and G types. In other words, you'd need a **distributive law**. Such an operation is not part of the **Monad** interface.



(by Runar Bjarnason)
@runarorama

Many Concepts Go Well Together

Before we delve into **monads**, let us consider those cases where there are no problems. If we know that both f and g are **functors**, we can make a new **functor** out of their **composition**.

...

Traversables provide a similar interface to **functors** but work with functions of the form a -> f b, where f is an **applicative functor**.

...

Given these similarities with **Functor**, we can reuse the idea of mapping twice to obtain an instance for the **composition** of two, **traversable functors**

...

The **applicative functor** is another structure that works well under **composition**.

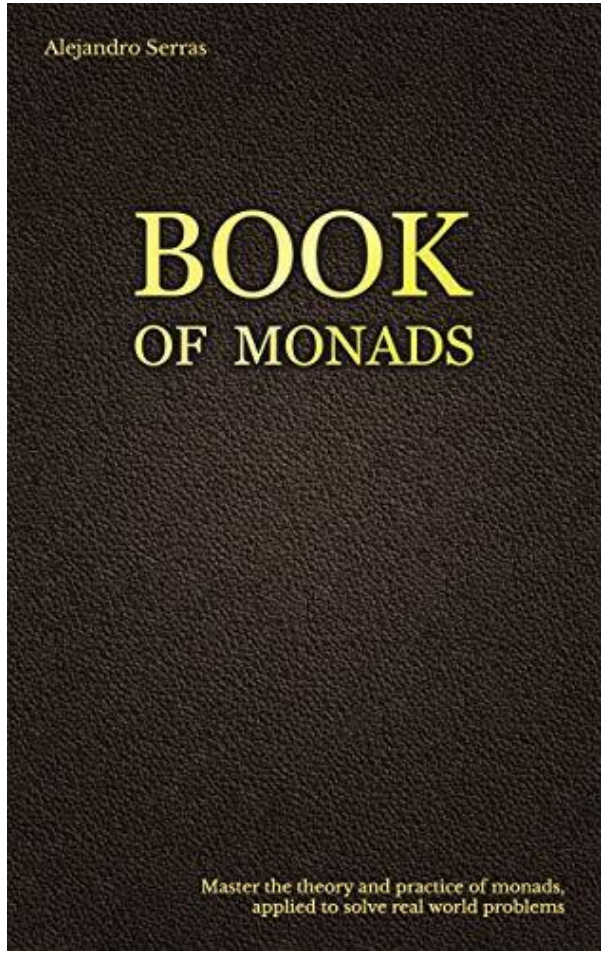
...

...

It seems like the promise of **composition** is achieved: start with a set of primitive **functors** — which might also be **traversables**, **applicatives**, or **alternatives** — and compose them as desired. The resulting **combination** is guaranteed to support at least the same operations as its constituents. If only that were true of **monads**.

But Monads Do Not

As you might have already guessed, **it is not possible to take two monads f and g and compose them into a new monad f :.: g in a generic way**. By generic way, we mean **a single recipe that works for every pair of monads**. Of course, there are some pairs of **monads** that can be combined easily, like two Readers, and others that need a bit more work, like lists and optionals, as shown at the beginning of the chapter. But, stressing the point once again, **there is no uniform way to combine any two of them**.



The Book of Monads: Master the theory and practice of monads, applied to solve real world problems

Alejandro Serrano Mena

@trupill

In order to understand why, we are going to consider in greater detail the idea of monads as boxes

```
class Monad m where
  return :: a -> m a
  join :: m (m a) -> m
```

We have just seen how to combine the **fmap** operations of two functors and the **pure** operations — **return** for **monads** — of two applicative functors. The latter method, **return**, poses no problem for composition: just take the definition of **pure** we described for the composition of two applicative functors. Therefore, we must conclude that **join** **is the one to blame**. The **join** operation for the composition of two **monads** has the following type:

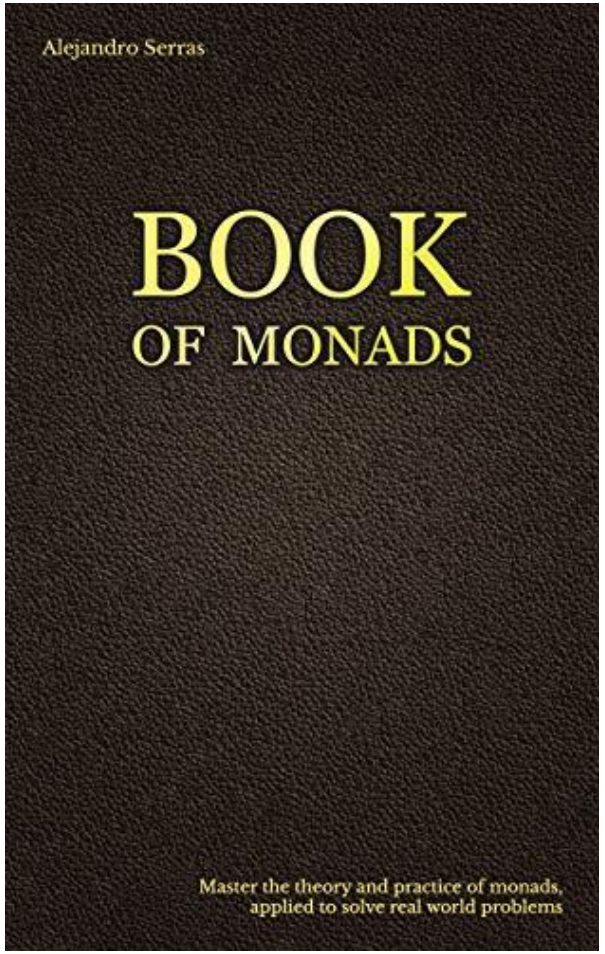
```
join :: (f :.: g) (( f :.: g) a) -> (f :.: g) a
```

If we leave out for a moment the newtype, this type amounts to:

```
join :: f (g (f (g a))) -> f (g a)
```

map	fmap
flatMap	bind
flatten/join	join
unit	pure/return

In a **monad,** we only have methods that add layers of monads — **return** and **fmap** — and a method that flattens two consecutive layers of the same **monad**. Alas, f (g (f (g a))) has interleaved layers, so there is no way to use join to reduce them to f (g a). As you can see, the reason why we cannot combine two **monads** is not very deep. It is just a simple matter of types that do not match. But the consequences are profound, since it tells us that **monads** cannot be freely composed.



The Book of Monads: Master the theory and practice of monads, applied to solve real world problems

Alejandro Serrano Mena

@trupill

Distributive Laws for Monads

One way to work around this problem is to provide a function that swaps the middle layers:

swap :: g (f a) -> f (g a)

This way, we can first turn f (g (f (g a))) into f (f (g (g a))) by running swap under the first layer. Then we can **join** the two outer layers, obtaining f (g (g a)) as a result. Finally, we **join** the two inner layers by applying the **fmap** operation under the **functor** f.

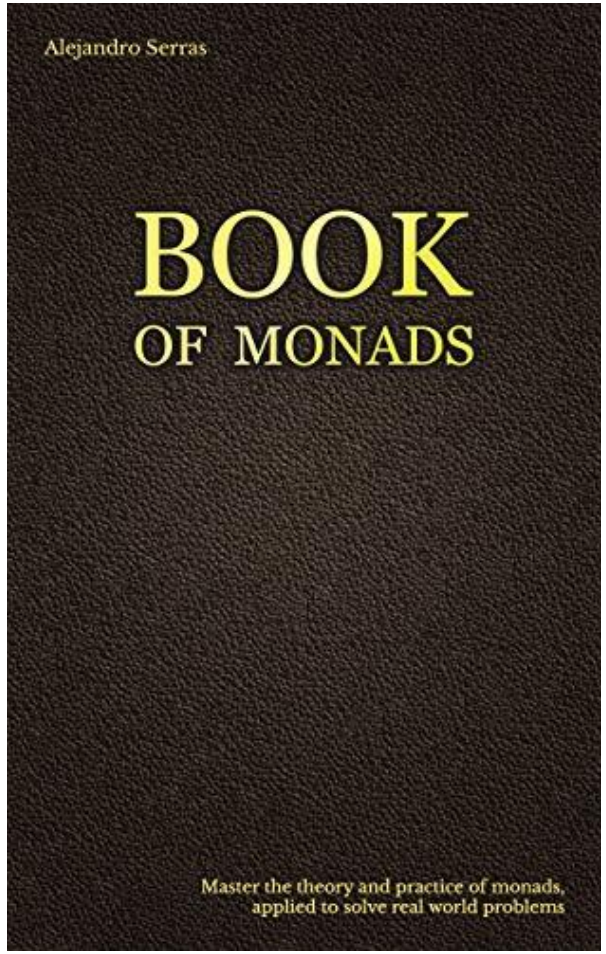
…

When such a function, **swap**, exists for two monads f and g, we say that there is a **distributive law** for those monads. In other words, if f and g have a **distributive relationship**, then they can be **combined** into a new **monad**.

…

Some **monads** even admit a **distributive law** with any other **monad**. The simplest example is given by **Maybe**.

…



The Book of Monads: Master the theory and practice of monads, applied to solve real world problems

Alejandro Serrano Mena

@trupill

The list monad

The discussion above contains a **small lie**. We definitely need a **swap** function if we want to combine two **monads**, but this is not enough. The reason is that a well-typed implementation may lead to a combined **monad** that violates one of the **monad laws**. The list **monad** is a well-known example of this.

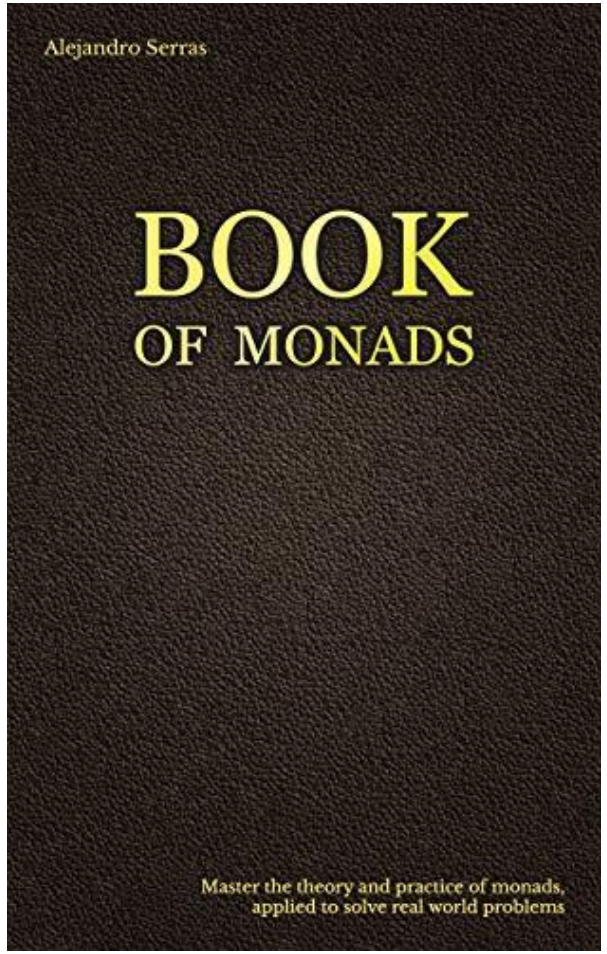
…

For the specific situation of the list **monad**, the cases for which its combination with another **monad** m lead to a new **monad** have been described. In particular, for the **swap** procedure to be correct, m needs to be a **commutative monad**, that is, the following blocks must be equal. The difference lies in the distinct order in which xs and ys are bound:

```
do x <- xs                    do y <- ys
   y <- ys          ≡             x <- xs
return (x, y)                 return (x, y)
```

The **Maybe monad** is **commutative**, since the order of failure does not matter for a final absent value, and in the case in which both elements are Just values, the result is the same. On the other hand, the list **monad** is not **commutative**: both blocks will ultimately result in the same elements, but the order in which they are produced will be different.

…



The Book of Monads: Master the theory and practice of monads, applied to solve real world problems

Alejandro Serrano Mena

@trupill

12.7.6 Monad composition

Let's now return to the issue of **composing monads**. As we saw earlier in this chapter, `Applicative` instances always compose, but `Monad` instances do not. If you tried before to implement general **monad composition**, then you would have found that in order to implement `join` for nested monads `F` and `G`, you'd have to write something of a type like `F[G[F[G[A]]]] => F[G[A]]`. And that can't be written generally.

But if `G` also happens to have a `Traverse` instance, we can `sequence` to turn `G[F[_]]` into `F[G[_]]`, leading to `F[F[G[G[A]]]]`. Then we can **join** the adjacent `G` layers using their respective `Monad` instances.

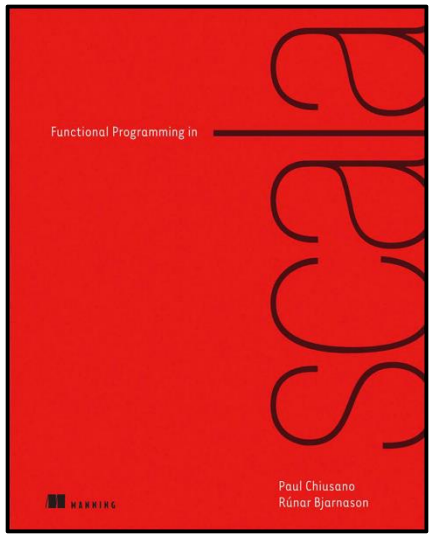
EXERCISE 12.20

Hard: implement the **composition** of two **monads** where one of them is **traversable**.

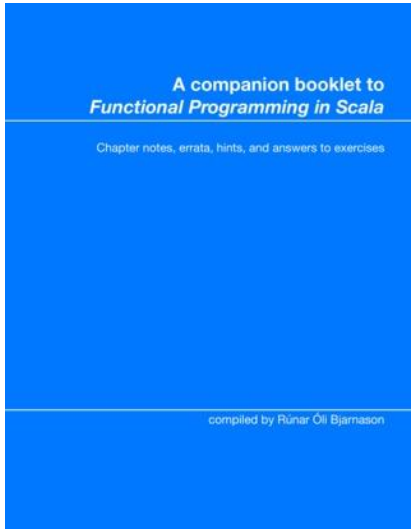
```
def composeM[F[_],G[_]](F: Monad[F], G: Monad [G], T: Traverse[G]):
  Monad[({type f[x] = F[G[x]]})#f]
```

Answer to Exercise 12.20

```
def composeM[G[_],H[_]](implicit G: Monad[G], H: Monad[H], T: Traverse[H]):
  Monad[({type f[x] = G[H[x]]})#f] = new Monad[({type f[x] = G[H[x]]})#f] {
    def unit[A](a: => A): G[H[A]] = G.unit(H.unit(a))
    override def flatMap[A,B](mna: G[H[A]])(f: A => G[H[B]]): G[H[B]] =
      G.flatMap(mna)(na => G.map(T.traverse(na)(f))(H.join))
}
```



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama



(by Runar Bjarnason)
@runarorama

As you can see on the previous slide, composeM uses Traverse.traverse

There is a lot to say about the Traverse trait that is out of scope for this slide deck.

The following excerpt from **FPiS** is sufficient for our current purposes.

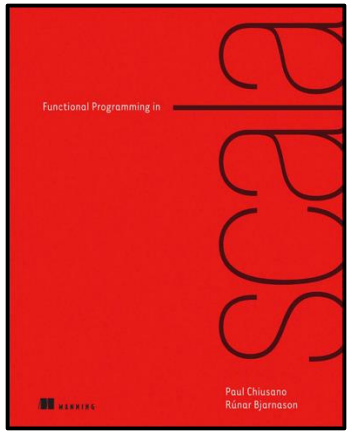
```
trait Traverse[F[_]] {

  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]
    sequence(map(fa)(f))

  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

[6] The name `Traversable` is already taken by an unrelated trait in the Scala standard library.

The interesting operation here is **sequence**. Look at its signature closely. It takes `F[G[A]]` and swaps the order of `F` and `G`, so long as `G` is an **applicative functor**. Now, this is a rather abstract, algebraic notion. We'll get to what it all means in a minute, but first, let's look at a few instances of **Traverse**.



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

If you are interested in Traverse, then see the following for an in depth explanation.

@philip_schwarz

slideshare @philip_schwarz

https://www.slideshare.net/pjschwarz/sequence-and-traverse-part-1
https://www.slideshare.net/pjschwarz/sequence-and-traverse-part-2
https://www.slideshare.net/pjschwarz/sequence-and-traverse-part-3

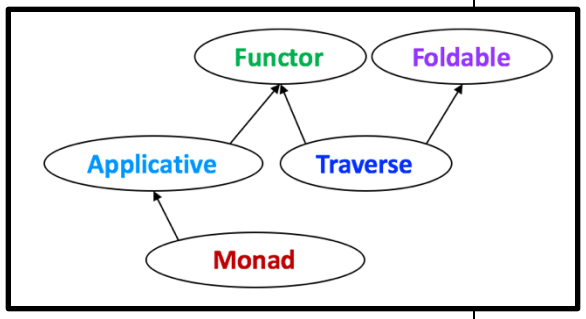
The next slide is just a teaser to whet your appetite.

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

trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]

  override def map[A,B](m: F[A])(f: A => B): F[B] =
    flatMap(m)(a => unit(f(a)))

  override def map2[A,B,C](ma:F[A], mb:F[B])(f:(A, B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))
}
```




```scala
trait Foldable[F[_]] {
  import Monoid._

  def foldRight[A, B](as: F[A])(z: B)(f: (A, B) => B): B =
    foldMap(as)(f.curried)(endoMonoid[B])(z)

  def foldLeft[A, B](as: F[A])(z: B)(f: (B, A) => B): B =
    foldMap(as)(a => (b: B) => f(b, a))(dual(endoMonoid[B]))(z)

  def foldMap[A,B](as:F[A])(f:A=>B)(implicit mb: Monoid[B]):B =
    foldRight(as)(mb.zero)((a, b) => mb.op(f(a), b))

  def concatenate[A](as: F[A])(implicit m: Monoid[A]): A =
    foldLeft(as)(m.zero)(m.op)
}
```

```scala
trait Traverse[F[_]] extends Functor[F] with Foldable[F] { self =>

  def traverse[M[_]:Applicative,A,B](fa:F[A])(f:A=>M[B]):M[F[B]]

  def sequence[M[_] : Applicative, A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)

  type Id[A] = A
  val idMonad = new Monad[Id] {
    def unit[A](a: => A) = a
    override def flatMap[A, B](a: A)(f: A => B): B = f(a)
  }

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

  import Applicative._

  override def foldMap[A,B](as: F[A])(f: A => B)
    (implicit mb: Monoid[B]): B =
    traverse[({type f[x] = Const[B,x]})#f,A,Nothing](
      as)(f)(monoidApplicative(mb))
 …
}
```

```scala
trait Monoid[A] {
  def op(x: A, y: A): A
  def zero: A
}
```

```scala
type Const[M, B] = M
implicit def monoidApplicative[M](M: Monoid[M]) =
  new Applicative[({ type f[x] = Const[M, x] })#f] {
    def unit[A](a: => A): M = M.zero
    def map2[A,B,C](m1: M, m2: M)(f: (A,B) => C): M = M.op(m1,m2)
  }
```

```scala
trait Applicative[F[_]] extends Functor[F] {

  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]

  def unit[A](a: => A): F[A]

  def map[B](fa: F[A])(f: A => B): F[B] =
    map2(fa, unit(()))((a, _) => f(a))

  def sequence[A](fas: List[F[A]]): F[List[A]] =
    traverse(fas)(fa => fa)

  def traverse[A,B](as: List[A])(f: A => F[B]): F[List[B]]
    as.foldRight(unit(List[B]()))((a, fbs) => map2(f(a), fbs)(_ :: _))
  …
}
```

In the next two slides we look at an example of composing a **Monad** with a **traversable Monad**.

@philip_schwarz

Example of composing a Monad with a traversable Monad

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

```scala
trait Applicative[F[_]] extends Functor[F] {

  def unit[A](a: => A): F[A]
  def map2[A,B,C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]

  def map[A,B](fa: F[A])(f: A => B): F[B] =
    map2(fa, unit(()))((a, _) => f(a))
}
```

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

  def traverse[M[_]:Applicative,A,B](fa: F[A])(f: A => M[B]): M[F[B]]

  def sequence[M[_]:Applicative,A](fma: F[M[A]]): M[F[A]] =
    traverse(fma)(ma => ma)
}
```

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

  def join[A](mma: F[F[A]]): F[A] = flatMap(mma)(ma => ma)

  def flatMap[A, B](ma: F[A])(f: A => F[B]): F[B]

  override def map[A, B](m: F[A])(f: A => B): F[B] =
    flatMap(m)(a => unit(f(a)))

  override def map2[A, B, C](ma: F[A], mb: F[B])(f: (A, B) => C): F[C] =
    flatMap(ma)(a => map(mb)(b => f(a, b)))

  def composeM[G[_]](G: Monad[G], T: Traverse[G]): Monad[λ[α => F[G[α]]]] = {
    val self = this
    new Monad[λ[α => F[G[α]]]] {
      def unit[A](a: => A): F[G[A]] = self.unit(G.unit(a))
      def flatMap[A, B](fga: F[G[A]])(f: A => F[G[B]]): F[G[B]] = {
        self.flatMap(fga){ ga => self.map(T.traverse(ga)(f)(self))(G.join) }
    }
  }
}
```

Example of composing a Monad with a traversable Monad

```scala
val optionApplicative = new Applicative[Option] {

  def unit[A](a: => A): Option[A] = Some(a)

  def map2[A,B,C](fa:Option[A],fb:Option[B])(f:(A,B)=>C):Option[C] =
    (fa, fb) match {
      case (Some(a), Some(b)) => Some(f(a,b))
      case _ => None
    }
}
```

```scala
val optionMonad = new Monad[Option] {

  def unit[A](a: => A): Option[A] = Some(a)

  def flatMap[A,B](ma: Option[A])(f: A => Option[B]): Option[B] =
    ma match {
      case Some(a) => f(a)
      case None => None
    }
}
```

```scala
val traversableListMonad = new Monad[List] with Traverse[List] {

  def unit[A](a: => A): List[A] = List(a)

  def flatMap[A,B](ma: List[A])(f: A => List[B]): List[B] = {
    ma.foldRight(List.empty[B])((a,bs) => f(a) ::: bs)
  }

  override def traverse[M[_],A,B](as: List[A])
    (f: A => M[B])(implicit M: Applicative[M]): M[List[B]] =
    as.foldRight(M.unit(List[B]()))((a, fbs) => M.map2(f(a), fbs)(_ :: _))
}
```

```scala
val parseInt: String => Option[Int] =
  s => Try{ s.toInt }.toOption

val charInts: String => Option[List[Char]] =
  s => parseInt(s).map(_.toString.toList)

assert( charInts("12") == Option( List('1', '2')) )
assert( charInts("1x") == None )
```

```scala
assert( traversableListMonad.traverse(List("12","23"))(parseInt)(optionApplicative) == Some(List(12, 23)))
assert( traversableListMonad.traverse(List("12","2x"))(parseInt)(optionApplicative) == None)

val listOptionMonad = optionMonad.composeM(traversableListMonad,traversableListMonad)

assert( listOptionMonad.flatMap( Option( List( "12", "34" ) ) )(charInts) == Option( List( '1', '2', '3', '4' ) ) )
assert( listOptionMonad.flatMap( Option( List( "12", "3x" ) ) )(charInts) == None )
```

There is **no generic composition strategy** that works for every **monad**

The issue of composing **monads** is often addressed with **monad transformers**

Expressivity and power sometimes come at the price of compositionality and modularity.

The issue of composing monads is often addressed with a custom-written version of each monad that's specifically constructed for composition. This kind of thing is called a **monad transformer**. For example, the `OptionT` monad transformer composes `Option` with any other **monad**:

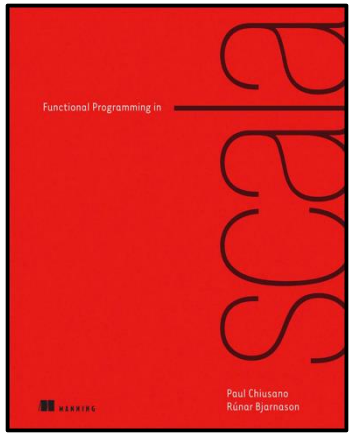
```
case class OptionT[M[_],A](value: M[Option[A]])(implicit M: Monad[M]) {

  def flatMap[B](f: A => OptionT[M, B]): OptionT[M, B] =
    OptionT(value flatMap {
      case None => M.unit(None)
      case Some(a) => f(a).value
    })

}
```

The `flatMap` definition here maps over both `M` and `Option`, and flattens structures like `M[Option[M[Option[A]]]]` to just `M[Option[A]]`. But this particular implementation is specific to `Option`. And the general strategy of taking advantage of `Traverse` works only with **traversable functors**. To compose with `State` (which can't be **traversed**), for example, a specialized `StateT` **monad transformer** has to be written. There's no generic composition strategy that works for every **monad**.

See the chapter notes for more information about monad transformers.



Functional Programming in Scala
(by Paul Chiusano and Runar Bjarnason)
@pchiusano @runarorama

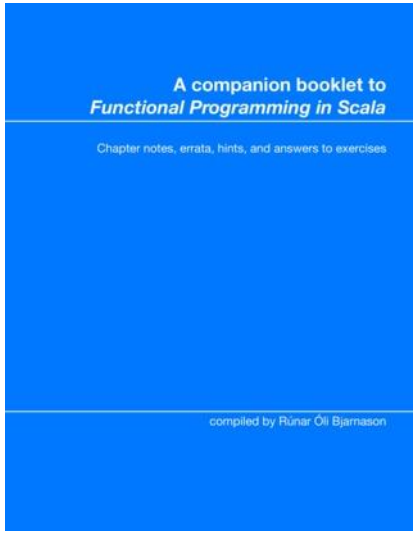
Monad transformers

A **monad transformer** is a data type that **composes** a particular **monad** with any other **monad**, giving us a **composite monad** that shares the behavior of both.

There is no general way of composing **monads**. Therefore we have to have a specific transformer for each **monad**.

For example, **OptionT** is a **monad transformer** that adds the behavior of **Option** to any other **monad**. The type **OptionT**[**M**, **A**] behaves like the composite **monad** **M**[**Option**[_]]. Its **flatMap** method binds over both the **M** and the **Option** inside, saving us from having to do the gymanstics of binding over both.

Scalaz provides many more **monad transformers**, including **StateT**, **WriterT**, **EitherT**, and **ReaderT** (also known as **Kleisli**).



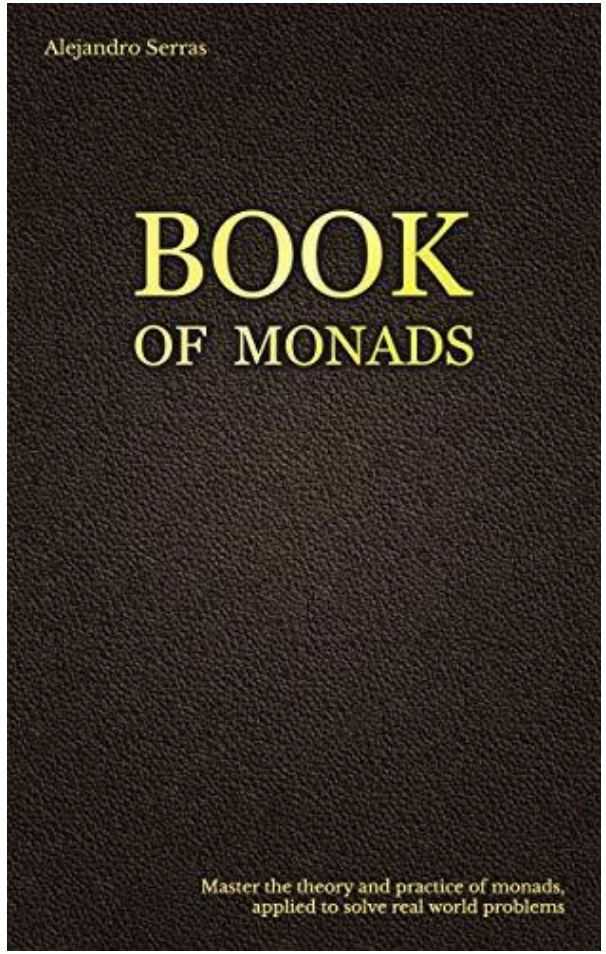
(by Runar Bjarnason)
@runarorama

A Solution: Monad Transformers

It would be impolite to thoroughly describe a problem — the composition of **monads** — and then not describe at least one of the solutions. That is the goal of this chapter, to describe how **monad transformers** allow us to combine the operations of several **monads** into one, single **monad**. It is not a complete solution, however, since we need to change the building blocks: instead of composing several different **monads** into a new **monad**, we actually enhance one **monad** with an extra set of operations via a **transformer**.

Alas, there is one significant downside to the naïve, **transformers** approach: we cannot abstract over **monads** that provide the same functionality but are not identical. This hampers the maintainability of our code, as any change in our **monadic** needs would lead to huge rewrites. The classic solution is to introduce type classes for different sets of operations. Consider that solution carefully, as it forms the basis of one of the styles for developing your own **monads**.

One word of warning before proceeding: **monad** transformers are ***a solution*** to the **monad composition** problem. But they are not ***the solution***. Another approach, **effects**, is gaining traction in the functional programming community. The right way to design an effects system is an idea that is still in flux, however, as witnessed by its many, different implementations.



The Book of Monads: Master the theory and practice of monads, applied to solve real world problems

Alejandro Serrano Mena

@trupill

If you are interested in knowing more about **Monad Transformers** then see the following

slideshare @philip_schwarz

https://www.slideshare.net/pjschwarz/ https://www.slideshare.net/pjschwarz/monad-transformers-part-1