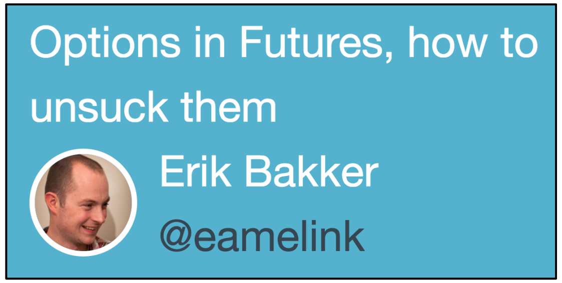
Monad Transformers

inspired by, and based on, **Erik Bakker**'s talk

Options in Futures, how to unsuck them



Scala Days

Part 1

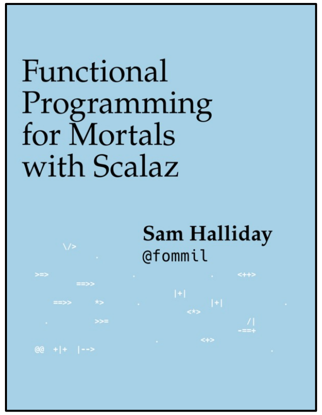
slides by  @philip_schwarz

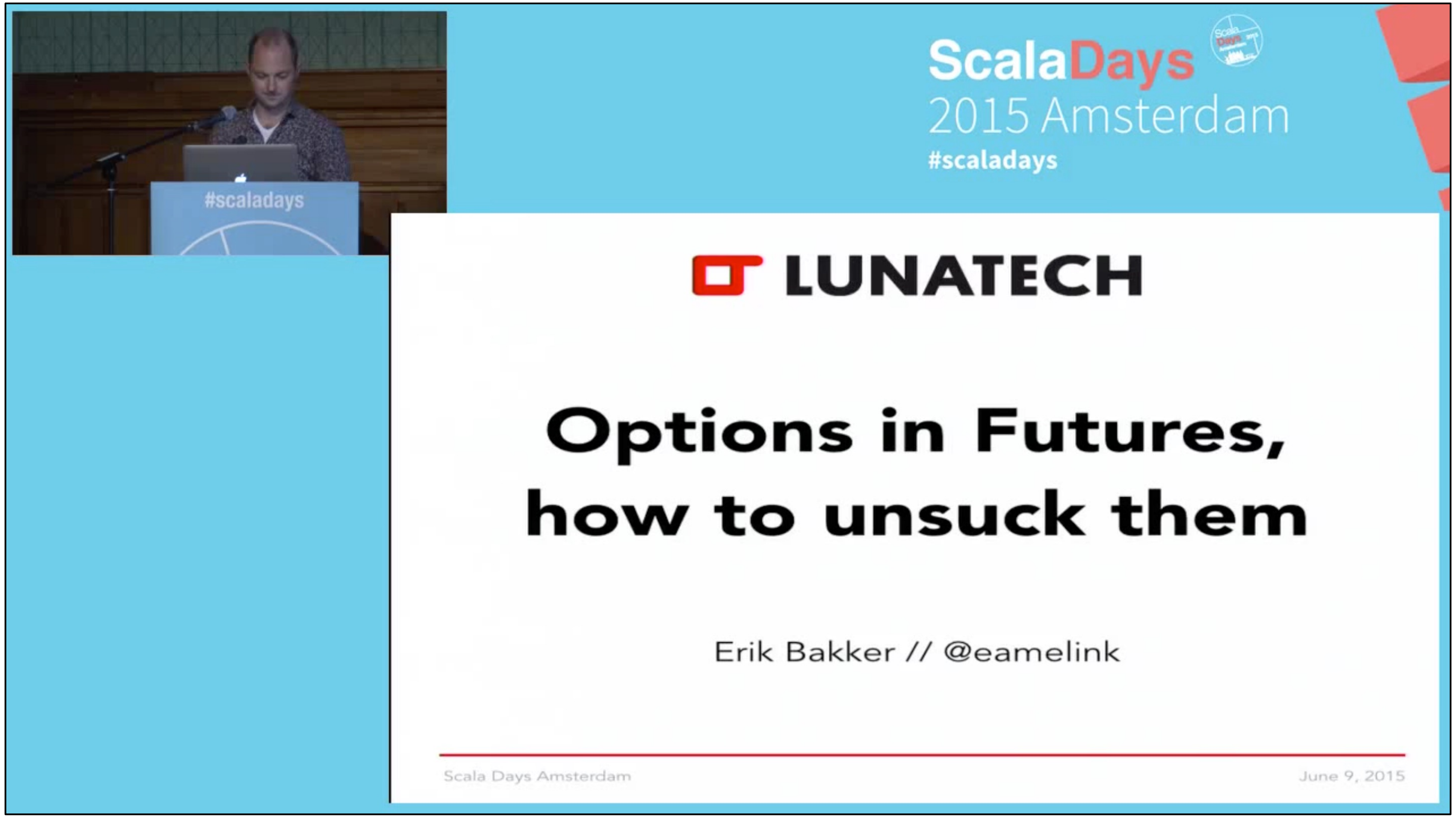
This slide deck is inspired, and based on, a great talk by **Erik Bakker**:

YouTube **Options in Futures, how to unsuck them** @eamelink

In his book, **Functional Programming for Mortals with Scalaz, Sam Halliday** has a 'Thanks' section in which he says: "Some material was particularly helpful for my own understanding of the concepts that are in this book". That section thanks **Erik Bakker** for 'Options in Futures, how to unsuck them'

Sam Halliday @fommil





Options in Futures, how to unsuck them

Erik Bakker

@eamelink

author of

We have a **problem**: we want to add two numbers **but both of the numbers are optional**.

Erik Bakker
@eamelink

```
def getX: Option[Int] = Some(3)
def getY: Option[Int] = Some(5)
```

So how can we do that?

We can use flatMap and then map:

```
val z: Option[Int] =
  getX flatMap { x =>
    getY map { y =>
      x + y
    }
  }

assert( z  == Some(8) )
```

There is a slightly nicer way which involves a for comprehension.

```
val z: Option[Int] =
  for {
    x <- getX
    y <- getY
  } yield x + y

assert( z  == Some(8) )
```

This is exactly the same as the previous one.

The latter one is just syntactic sugar for the map and flatMap but it is slightly nicer because **you don't get the nesting** and it is **visually clearer** what you are doing.

So this is just a **mechanical transformation** that is just written in the Scala language specification: if you have this for comprehension then this is the way it desugars, it is very early on in the compile phase, and it is really just a mechanical transformation.

And what is interesting here is that this doesn't just work for Options, it works for anything that has the necessary methods that the for comprehension desugars to, so in the cases that I am using that means the objects need to have map and flatMap methods.

So, we are familiar with some objects that have map and flatMap methods: Option, Future, List, etc, so we can use all these in for comprehensions.

So for example take this one: it's the same problem except this time the numbers are not in an Option but they are in a Future

```
def getX: Future[Int] = Future(5)
def getY: Future[Int] = Future(3)
```

We can still do it with map and flatMap:

We can use the same for comprehension, except the result is a Future of an Int this time

```
val z: Future[Int] =
  getX flatMap { x =>
    getY map { y =>
      x + y
    }
  }

Await.ready(z,Duration.Inf)
assert( z.toString == "Future(Success(8))")
```

```
val z: Future[Int] =
  for {
    x <- getX
    y <- getY
  } yield x + y
```

And here is an example for `List`

```
def getX: List[Int] = List(1,2)
def getY: List[Int] = List(3,4)
```

```
val z: List[Int] =
  getX flatMap { x =>
    getY map { y =>
      x + y
    }
  }
```

```
val z: List[Int] =
  for {
    x <- getX
    y <- getY
  } yield x + y
```

```
assert( z == List(4,5,5,6) )
```

```scala
def getX: Future[Int] = Future(3)
def getY: Future[Int] = Future(5)
```

```scala
val z: Future[Int] =
  getX flatMap { x =>
    getY map { y =>
      x + y
    }
  }
```

```scala
val z: Future[Int] =
  for {
    x <- getX
    y <- getY
  } yield x + y
```

```scala
Await.ready(z,Duration.Inf)
assert( z.toString == "Future(Success(8))” )
```

```scala
def getX: Option[Int] = Some(3)
def getY: Option[Int] = Some(5)
```

```scala
val z: Option[Int] =
  getX flatMap { x =>
    getY map { y =>
      x + y
    }
  }
```

```scala
val z: Option[Int] =
  for {
    x <- getX
    y <- getY
  } yield x + y
```

```scala
assert( z  == Some(8) )
```

```scala
def getX: List[Int] = List(1,2)
def getY: List[Int] = List(3,4)
```

```scala
val z: List[Int] =
  getX flatMap { x =>
    getY map { y =>
      x + y
    }
  }
```

```scala
val z: List[Int] =
  for {
    x <- getX
    y <- getY
  } yield x + y
```

```scala
assert( z == List(4,5,5,6) )
```

@philip_schwarz

As **Erik** said, we can use ’**the same**‘ **for comprehension** for **Option**[Int], **Future**[Int], **List**[Int], etc.

Similarly for the nested **flatMap** and **map**.

But what do we mean by ‘**the same**’?

We mean that **copies** of ’**the same**‘ **for comprehension**, or **copies** of ’**the same**‘ nested **flatMap**/**map**, can be used for **Option**[Int], **Future**[Int], **List**[Int]. This is because it is only the type of **z**, **getX** and **getY**, that needs to change.


```scala
def getX: Future[Int] = Future(3)
def getY: Future[Int] = Future(5)
```

```scala
val z: Future[Int] =
  getX flatMap { x =>
    getY map { y =>
      x + y
    }
  }
```

```scala
val z: Future[Int] =
  for {
    x <- getX
    y <- getY
  } yield x + y
```

```scala
Await.ready(z,Duration.Inf)
assert( z.toString == "Future(Success(8))” )
```

```scala
def getX: Option[Int] = Some(3)
def getY: Option[Int] = Some(5)
```

```scala
val z: Option[Int] =
  getX flatMap { x =>
    getY map { y =>
      x + y
    }
  }
```

```scala
val z: Option[Int] =
  for {
    x <- getX
    y <- getY
  } yield x + y
```

```scala
assert( z  == Some(8) )
```

```scala
def getX: List[Int] = List(1,2)
def getY: List[Int] = List(3,4)
```

```scala
val z: List[Int] =
  getX flatMap { x =>
    getY map { y =>
      x + y
    }
  }
```

```scala
val z: List[Int] =
  for {
    x <- getX
    y <- getY
  } yield x + y
```

```scala
assert( z == List(4,5,5,6) )
```

What does it take to allow the **very same code**, rather than **copies of the same code**, to be used for **Option**[**Int**], **Future**[**Int**], **List**[**Int**], etc?

Is it possible to write a **single method**, **sum** say, that takes a pair of **Option**[**Int**] or a pair of **Future**[**Int**] or a pair of **List**[**Int**], etc, and uses the nested **flatMap**/**map**, or the **for comprehension**, to add two integers and return an **Option**[**Int**] or **Future**[**Int**] or **List**[**Int**], etc?

i.e. is it possible to get the following two methods to work?

```
def sum[M[_]](mx:M[Int], my:M[Int])(implicit m: Monad[M]): M[Int] =
  m.flatMap(mx) { x =>
    m.map(my) { y =>
      x + y
    }
  }
```

```
def sum[M[_]](mx:M[Int],my:M[Int])(implicit m: Monad[M]): M[Int] =
  for {
    x <- mx
    y <- my
  } yield x + y
```

If you are interested in this question, and you are quite familiar with **Monads**, then see the following short slide deck, otherwise you can safely move on.

https://www.slideshare.net/pjschwarz/ abstracting-over-the-monad-yielded-by-a-for-comprehension-and-its-generators

slideshare @philip_schwarz

Erik Bakker
@eamelink

So far so easy: **what's the problem**?

The problem that you run into a lot these days, because there are so many asynchronous libraries that return futures, is that **you get nested things**, **nested containers, nested contexts**, for example, a `Future` with an `Option` inside, and if you try to work with these you might have noticed, this kind of **sucks**.

```scala
def getX: Future[Option[Int]] = Future(Some(5))
def getY: Future[Option[Int]] = Future(Some(3))
```

So **what we are going to see in this talk is a way to unsuck working with these things**.

If we try to use a **for comprehension** like we did for the previous example, then this doesn't work because if you write it like this then in the **for comprehension**, left of the arrows, the x and y are `Option` of `Int` and they are not `Int`s, so **in the yield they are still `Option` of `Int`, and we cannot just add them**.

```scala
val z: Future[Option[Int]] = for {
  x <- getX
  y <- getY
} yield x + y
```

Type mismatch, expected: String, actual: Option[Int]

Of course there is no real issue, we can solve this, we can make this program where we just want to add these two integers, **we just use some more maps and flatMaps**, first to **map** the futures and then once we have got stuff out of the futures we **map** and **flatMap** some more to **map** the options.

```scala
val z: Future[Option[Int]] =
  getX flatMap { xOpt =>
    getY map { yOpt =>
      xOpt flatMap { x =>
        yOpt map { y =>
          x + y
        }
      }
    }
  }
```

But **this gets messy** - it is this messy if you have **two levels deep** and **it gets much messier** if you have more things coming out of a `Future` or `Option`.

You can improve slightly on this in an easy way by doing pattern matching immediately, so you can write it like this and avoid mapping on the Option because we immediately pattern match on the None and the Some of the Option

```
val z: Future[Option[Int]] =
  getX flatMap { xOpt =>
    getY map { yOpt =>
      xOpt flatMap { x =>
        yOpt map { y =>
          x + y
        }
      }
    }
  }
```

```
val z: Future[Option[Int]] =
  getX flatMap { xOpt =>
    xOpt match {
      case None => Future.successful(None)
      case Some(x) => getY map { yOpt =>
        yOpt match {
          case None => None
          case Some(y) => Some(x + y)
        }
      }
    }
  }
```

Another way of improving slightly on the above

@philip_schwarz

```
val z: Future[Option[Int]] =
  getX flatMap { xOpt =>
    getY map { yOpt =>
      (xOpt,yOpt) match {
        case (Some(x),Some(y)) => Some(x + y)
        case _ => None
      }
    }
  }
```

So, what is the main issue that we have? the main issue that we have is that we are trying to use **map** and **flatMap** on a thing but **map** and **flatMap** do not work on the most inner value, so the integer in the structure, it works only one level deep, so if we use **map** and **flatMap** on **Future**, then **what we work with is the Option**, while **what we actually want to work on is the integer**, so **that is basically what we are going to solve**.

And the solution is not very hard.

We'll just define **a new wrapper**, let's call it **FutureOption**, that contains one of these values, that contains a **Future** of **Option**.

```
case class FutureOption[A](inner: Future[Option[A]])
```

And now we are going to implement **map** and **flatMap** on this thing in such a way that it works on the innermost value.

Then you get to the point: what is a **map** function? How should it look?

Well, for me that is just monkey see monkey do: we **take a look at some other map methods**, on **List** for example, on **Option** and **Future**, and **you can see that they all have the same structure**.

```
// List[A]
def map[B](f: A => B): List[B]

// Option[A]
def map[B](f: A => B): Option[B]

// Future[A]
def map[B](f: A => B): Future[B]
```

except in **Scala** the last one also takes an execution context, but we'll just ignore that for now, actually, for the entire talk.

```
map[B](f:A => B)(implicit executor:ExecutionContext):Future[B]
```

But this is how **map** looks on most of the other stuff in **Scala** so let's just mimic that.

We are going to implement on our `FutureOption` a method **map** like this:

```scala
def map[B](f: A => B): FutureOption[B]
```

That's **not terribly hard**

```scala
// List[A]
def map[B](f: A => B): List[B]

// Option[A]
def map[B](f: A => B): Option[B]

// Future[A]
def map[B](f: A => B): Future[B]
```

```scala
case class FutureOption[A](inner: Future[Option[A]]){

  def map[B](f: A => B): FutureOption[B] =
    FutureOption { inner map { _ map { f } } }

}
```

We are done. One down, one to go: **flatMap**.

How does **flatMap** look like on these existing classes from the standard library?

```scala
// List[A]
def flatMap[B](f: A => List[B]): List[B]

// Option[A]
def flatMap[B](f: A => Option[B]): Option[B]

// Future[A]
def flatMap[B](f: A => Future[B]): Future[B]
```

Very similar, except the function is not A to B, but it's A to a B **inside the** container, **inside the** context, for `List`, `Option`, `Future`, **very similar**, and looking at that we can define **the function we need to implement**:

```scala
def flatMap[B](f:A => FutureOption[B]):FutureOption[B]
```

Implementing **flatMap** is **slightly harder**, it's not hard, it's an **interesting puzzle**, so I encourage you to try, but as you can see, the solution is **not very hard**

```scala
case class FutureOption[A](inner: Future[Option[A]]){

  def map[B](f: A => B): FutureOption[B] =
    FutureOption { inner map { _ map { f } } }

  def flatMap[B](f: A => FutureOption[B]): FutureOption[B] =
    FutureOption {
      inner flatMap {
        case Some(a) => f(a).inner
        case None => Future.successful(None)
      }
    }
}
```

That completes our FutureOption class. It now has a **map** and a **flatMap** function, and **they both work on the inner value**, they don't work on the Option inside the Future, they work on the value of type A that's at the centre of this structure.

And **given that we now have a thing that has a map and a flatMap, we can use this in for comprehensions**, because **for comprehensions work on anything with map and flatMap**, there is no trait that you need to implement, there is nothing, **as long as you have map and flatMap, it will just work**.

So back to our original problem.

getX and **getY** still return a Future of an Option of an Int

```
def getX: Future[Option[Int]] = Future(Some(5))
def getY: Future[Option[Int]] = Future(Some(3))
```

and we want to add the two integers that are at the centre of these structures, so now we just wrap our Future of Option of Int in our new FutureOption class, use that in a **for comprehension**, and now in the **for comprehension**, x and y are integers and we can just add them.

```
val z: FutureOption[Int] =
  for {
    x <- FutureOption(getX)
    y <- FutureOption(getY)
  } yield x + y
```

What happened when we used straight Future[Option[Int]]

```
val z: Future[Option[Int]] = for {
  x <- getX
  y <- getY
} yield x + y
```

Type mismatch, expected: String, actual: Option[Int]

Of course, the result that we get there is also a FutureOption of Int which is probably not the structure that you want to continue using in the remainder of your program, this is just a class that we made up, so we have to get the inner value out again, which is easy, we just take the **inner** field

```
val z1: FutureOption[Int] =
  for {
    x <- FutureOption(getX)
    y <- FutureOption(getY)
  } yield x + y

val z = z1.inner
```

What happens now that we use FutureOption

```
val z: FutureOption[Int] = for {
  x <- FutureOption(getX)
  y <- FutureOption(getY)
} yield x + y
```

Pattern: y: Int

or get to the inner value using pattern matching

```
val FutureOption(z): FutureOption[Int] =
  for {
    x <- FutureOption(getX)
    y <- FutureOption(getY)
  } yield x + y
```

Let's see again the whole code for the **FutureOption** example

```
case class FutureOption[A](inner: Future[Option[A]]){

  def map[B](f: A => B): FutureOption[B] =
    FutureOption { inner map { _ map { f } } }

  def flatMap[B](f:A => FutureOption[B]): FutureOption[B] =
    FutureOption {
      inner flatMap {
        case Some(a) => f(a).inner
        case None => Future.successful(None)
      }
    }
}
```

```
def getX: Future[Option[Int]] = Future(Some(5))
def getY: Future[Option[Int]] = Future(Some(3))
```

```
val FutureOption(z): FutureOption[Int] =
  for {
    x <- FutureOption(getX)
    y <- FutureOption(getY)
  } yield x + y
```

```
val result = Await.result(z,Duration.Inf)
assert( result == Some(8) )
```

And here is another example: **ListOption**

```
case class ListOption[A](inner: List[Option[A]]){

  def map[B](f: A => B): ListOption[B] =
    ListOption { inner map { _ map { f } } }

  def flatMap[B](f:A => ListOption[B]): ListOption[B] =
    ListOption {
      inner flatMap {
        case Some(a) => f(a).inner
        case None => List(None)
      }
    }
}
```

```
def getX: List[Option[Int]] = List(Some(5), Some(6))
def getY: List[Option[Int]] = List(Some(3), Some(4))
```

```
val ListOption(z): ListOption[Int] =
  for {
    x <- ListOption(getX)
    y <- ListOption(getY)
  } yield x + y
```

```
assert( z == List(Some(8),Some(9),Some(9),Some(10)) )
```

So, basically this is almost everything there is to it: we have an **interesting structure** and we just wrap it in something that knows how to get the **innermost value**, we define **map** and **flatMap** for that, and then we can use it in **for comprehensions**.

Except that **the thing we have now is very specific**, **it only works on this structure: a `Future` with an `Option` inside**, but that is not the only structure that we are working with, we have values that come in all different kinds of shapes, so **we need to see if we can generalize this a bit**.

So in part 2, **we are going to try to generalize this very simple class**, that you could have written, **into something that is more widely applicable**.

Part 2
Generalizing FutureOption

```
case class FutureOption[A](inner: Future[Option[A]]){

  def map[B](f: A => B): FutureOption[B] =
    FutureOption { inner map { _ map { f } } }

  def flatMap[B](f: A => FutureOption[B]): FutureOption[B] =
    FutureOption {
      inner flatMap {
        case Some(a) => f(a).inner
        case None => Future.successful(None)
      }
    }
}
```

So **take another good look at `FutureOption`**. What you see here is that **from the `Future`, the inner, we only use three things**. We use **map**, we use **flatMap** and **we create a new one**, we just create a new `Future` with some value inside.

So that's interesting to notice.

We only do three things with the outer container:

- **map**
- **flatMap**
- **create** a new one

These are the operations we have on **monads**!

So that is something we could abstract over.

So let's say, instead of making this thing for **Future**, let's make an interface for this.

Yes, let's just make a trait that has a type parameter, and the type is a **Future**, that has a **map** and a **flatMap** method, and **give it a suitable name**, people have done that, and **the suitable name for this is Monad**.

So let's define a **Monad** trait that looks like this

```
trait Monad[M[_]] {
  def map[A, B](ma: M[A])(f: A => B): M[B]
  def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]
  def create[A](a:A): M[A]
}
```

It has a **map** and a **flatMap** that look very similar to the ones we have defined before. The only difference is then we defined **map** and **flatMap** on an object and here the object is external, so the first parameter to **map** and **flatMap** is the thing that you want to **map** and **flatMap**.

But using this trait we can **generalize** our **FutureOption** class and make it an **AnyMonadOption** class that is parameterised not just by the **inner** value type but also by the type of **Monad**, the **outer** of the stack, so we had a **Future Option** something, I call **Future** the **outer** and **Option** the **inner** thing.

```
case class AnyMonadOption[M[_], A](inner: M[Option[A]])(implicit m: Monad[M]) {

  def map[B](f: A => B): AnyMonadOption[M, B] =
    AnyMonadOption {
      m.map(inner)(_ map { f } )
    }

  def flatMap[B](f: A => AnyMonadOption[M, B]): AnyMonadOption[M, B] =
    AnyMonadOption {
      m.flatMap(inner){
        case Some(a) => f(a).inner
        case None => m.create(None)
      }
    }
}
```

So we have paremeterised over the **outer** one, which is **M**, and then we say this thing takes a value, some **M** with inside it an **Option** of **A**. We need a **Monad** instance for this thing, otherwise we don't know how we would **map** and **flatMap** the **M**. Now that we have the type class for that we can do that, and now we can redefine **map** and **flatMap** to not call **map** and **flatMap** on the object itself, but on the implementation of the **Monad** trait for this thing.

So what would we need to reuse this for **Future**s, **Option**s? We have to implement this **Monad** trait for **Future**s. Well, you can imagine that it is not to hard, to implement **map**, **flatMap** and **create** for **Future**s, because it already has **map** and **flatMap** methods.

So that's easy.

let's have a go at using our `AnyMonadOption` with `Future`

```scala
def getX: Future[Option[Int]] = Future(Some(5))(global)
def getY: Future[Option[Int]] = Future(Some(3))(global)

implicit val futureMonad: Monad[Future] = new Monad[Future] {
  def map[A, B](ma: Future[A])(f: A => B): Future[B] = ma map f
  def flatMap[A, B](ma: Future[A])(f: A => Future[B]): Future[B] = ma flatMap f
  def create[A](a: A): Future[A] = Future(a)
}

val z: AnyMonadOption[Future,Int] = for {
  x <- AnyMonadOption(getX)(futureMonad)
  y <- AnyMonadOption(getY)(futureMonad)
} yield x + y

val result: Option[Int] = Await.result(z.inner,Duration.Inf)
assert( result == Some(8) )
```

and now with `List`

@philip_schwarz

```scala
def getX: List[Option[Int]] = List(Some(5),Some(6))
def getY: List[Option[Int]] = List(Some(3),Some(4))

implicit val listMonad: Monad[List] = new Monad[List] {
  def map[A, B](ma: List[A])(f: A => B): List[B] = ma map f
  def flatMap[A, B](ma: List[A])(f: A => List[B]): List[B] = ma flatMap f
  def create[A](a: A): List[A] = List(a)
}

val z: AnyMonadOption[List,Int] = for {
  x <- AnyMonadOption(getX)(listMonad)
  y <- AnyMonadOption(getY)(listMonad)
} yield x + y

assert( z.inner == List(Some(8),Some(9),Some(9),Some(10)))
```

Just as a recap, let's compare the **initial approach**, in which we have to write a new class for each **outer** type that we want to wrap an **Option** with, i.e. **Future**, **List**, etc

```
case class FutureOption[A](inner: Future[Option[A]]){

  def map[B](f: A => B): FutureOption[B] =
    FutureOption { inner map { _ map { f } } }

  def flatMap[B](f:A => FutureOption[B]): FutureOption[B] =
    FutureOption {
      inner flatMap {
        case Some(a) => f(a).inner
        case None => Future.successful(None)
      }
    }
}
```

```
case class ListOption[A](inner: List[Option[A]]){

  def map[B](f: A => B): ListOption[B] =
    ListOption { inner map { _ map { f } } }

  def flatMap[B](f:A => ListOption[B]): ListOption[B] =
    ListOption {
      inner flatMap {
        case Some(a) => f(a).inner
        case None => List(None)
      }
    }
}
```

And the improved approach, in which instead of writing a new class, for each **outer** type **Future**, **List**, etc, we instantiate **AnyMonadOption** for the **outer** type (and supply an implicit monad for the **outer** type).

```
case class AnyMonadOption[M[_], A](inner: M[Option[A]])(implicit m: Monad[M]) {

  def map[B](f: A => B): AnyMonadOption[M, B] =
    AnyMonadOption {
      m.map(inner)(_ map { f } )
    }

  def flatMap[B](f: A => AnyMonadOption[M, B]): AnyMonadOption[M, B] =
    AnyMonadOption {
      m.flatMap(inner){
        case Some(a) => f(a).inner
        case None => m.create(None)
      }
    }
}
```

```
trait Monad[M[_]] {
  def map[A, B](ma: M[A])(f: A => B): M[B]
  def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]
  def create[A](a:A): M[A]
}
```

```
implicit val futureMonad: Monad[Future] = new Monad[Future] {
  def map[A, B](ma: Future[A])(f: A => B): Future[B] = ma map f
  def flatMap[A, B](ma: Future[A])(f: A => Future[B]): Future[B] = ma flatMap f
  def create[A](a: A): Future[A] = Future(a)
}
```

```
implicit val listMonad: Monad[List] = new Monad[List] {
  def map[A, B](ma: List[A])(f: A => B): List[B] = ma map f
  def flatMap[A, B](ma: List[A])(f: A => List[B]): List[B] = ma flatMap f
  def create[A](a: A): List[A] = List(a)
}
```

So what we have got now is some sort of structure that **takes a Monad and it also is a Monad itself**. Why is it a Monad? Because **it has map and flatMap methods and it has a constructo**r, so you can create new ones if you put a value in, and **people also have given this a name**, they say **this is a MonadTransformer because it takes a Monad and it transforms it into a Monad that behaves slightly differently**.

AnyMonadOption[M[_], A]
is a
Monad Transformer

```scala
case class AnyMonadOption[M[_], A](inner: M[Option[A]])(implicit m: Monad[M]) {

  def map[B](f: A => B): AnyMonadOption[M, B] =
    AnyMonadOption {
      m.map(inner)(_ map { f } )
    }

  def flatMap[B](f: A => AnyMonadOption[M, B]): AnyMonadOption[M, B] =
    AnyMonadOption {
      m.flatMap(inner){
        case Some(a) => f(a).inner
        case None => m.create(None)
      }
    }
}
```

A natural question would be, hey, **if we have generalised over the outer container, the Future, can we also generalize over the inner container, the Option**?

Can we basically make some class and whatever stack of **monads** you put in, it will end up a single **monad** and **it is going to be perfect**?

Let's have a go at at generalising AnyMonadOption over the inner container.

So we are taking AnyMonadOption

```scala
case class AnyMonadOption[M[_], A](inner: M[Option[A]])(implicit m: Monad[M]) {

  def map[B](f: A => B): AnyMonadOption[M, B] = …

  def flatMap[B](f: A => AnyMonadOption[M, B]): AnyMonadOption[M, B] = …
}
```

And turning it into AnyMonadMonad

```scala
case class AnyMonadMonad[M[_], N[_], A](inner: M[N[A]])(implicit m: Monad[M], n: Monad[M]) {

  def map[B](f: A => B): AnyMonadMonad[M, N, B] = ???

  def flatMap[B](f: A => AnyMonadMonad[M, N, B]): AnyMonadMonad[M, N, B] = ???

}
```

and we now want to have a go at implementing **map** and **flatMap**

@philip_schwarz

Implementing **map** is easy.

Here is how we can modify the **map** implementation of **AnyMonadOption** to obtain a **map** implementation for **AnyMonadMonad**.

```
def map[B](f: A => B): AnyMonadOption[M, B] =
    AnyMonadOption {
      m.map(inner)(_ map { f } )
    }
```

```
def map[B](f: A => B): AnyMonadMonad[M, N, B] =
    AnyMonadMonad {
      m.map(inner)(na => n.map(na){ f } )
    }
```

Let’s try it out.

```
def getX: Future[Option[Int]] = Future(Some(5))(global)

implicit val futureMonad: Monad[Future] = new Monad[Future] {
  def map[A, B](ma: Future[A])(f: A => B): Future[B] = ma map f
  def flatMap[A, B](ma: Future[A])(f: A => Future[B]): Future[B] = ma flatMap f
  def create[A](a: A): Future[A] = Future(a)
}

 implicit val optionMonad: Monad[Option] = new Monad[Option] {
  def map[A, B](ma: Option[A])(f: A => B): Option[B] = ma map f
  def flatMap[A, B](ma: Option[A])(f: A => Option[B]): Option[B] = ma flatMap f
  def create[A](a: A): Option[A] = Option(a)
}

val z: AnyMonadMonad[Future,Option,Int] = for {
  x <- AnyMonadMonad(getX)(futureMonad,optionMonad)
} yield x + 3

val result: Option[Int] = Await.result(z.inner,Duration.Inf)
assert( result == Some(8) )
```

Now let's try to implement flatMap.

Let's have a go at modifying the flatMap implementation of AnyMonadOption to obtain a flatMap implementation for AnyMonadMonad.

```
def flatMap[B](f: A => AnyMonadOption[M, B]): AnyMonadOption[M, B] =
  AnyMonadOption {
    m.flatMap(inner){
      case Some(a) => f(a).inner
      case None => m.create(None)
    }
  }
```

```
def flatMap[B](f: A => AnyMonadMonad[M, N, B]): AnyMonadMonad[M, N, B] =
  AnyMonadMonad {
    m.flatMap(inner){ na =>
      n.flatMap(na){ a => f(a).inner }
    }
  }
```

It doesn't work!

```
[error]  found    : M[N[B]]
[error]  required: N[?]
[error]           n.flatMap(na){ a => f(a).inner }
[error]                                    ^
```

Let's just add types in a couple of places to aid comprehension

```
def flatMap[B](f: A => AnyMonadMonad[M, N, B]): AnyMonadMonad[M, N, B] =
  AnyMonadMonad {
    m.flatMap(inner:M[N[A]]){ na:N[A] =>
      n.flatMap(na){ a:A => val mnb: M[N[B]] = f(a).inner; mnb }
    }
  }
```

Expression of type M[N[B]] doesn't conform to expected type N[B_]

The problem is that f(a) yields an AnyMonadMonad[M, N, B] and so f(a).inner is an M[N[B]], whereas the n Monad's flatMap is supposed to yield an N[B]:

```
def flatMap[A, B](na: N[A])(f: A => N[B]): N[B]
```

But how can flatMap possibly turn M[N[B]] into N[B], without knowing anything about M and N other than that they are Monads? **It can't**.

Note that it is possible for any monad to turn N[N[B]] into N[B], because every monad can define a function that does just that, i.e. join (aka flatten).

```
trait Monad[M[_]] {
  def map[A, B](ma: M[A])(f: A => B): M[B]
  def flatMap[A, B](ma: M[A])(f: A => M[B]): M[B]
  def create[A](a:A): M[A]
  def join[A](mma:M[M[A]]): M[A] = flatMap(mma)(ma => ma)
}
```

But flattening N[N[B]] to N[B] is not the problem at hand. The problem is turning M[N[B]] into N[B], which AnyMonadMonad cannot do.

We had a go at at generalising **AnyMonadOption** over the **inner** container.

We tried taking **AnyMonadOption** and turning it into **AnyMonadMonad**

```
case class AnyMonadMonad[M[_], N[_], A](inner: M[N[A]])(implicit m: Monad[M], n: Monad[M]) {

  def map[B](f: A => B): AnyMonadMonad[M, N, B] = ???

  def flatMap[B](f: A => AnyMonadMonad[M, N, B]): AnyMonadMonad[M, N, B] = ???

}
```

But we did not succeed: we were able to implement **map**, but not **flatMap**

@philip_schwarz

So back to this question:

A natural question would be, hey, **if we have generalised over the outer container, the Future, can we also generalize over the inner container, the Option**?

Can we basically make some class and whatever stack of **monads** you put in, it will end up a single **monad** and **it is going to be perfect**?

Here is **Erik**'s answer:

That is **not possible apparently**.

Maybe you have heard people say, or have read the phrase, **monads are not composable**, and this is basically what they mean: **you can't make a single recipe that takes two monads and transforms them into a new monad, you have to specialize it for one of the two monads**.

So we have made a specific recipe that works with any **monad** with an **Option** inside. We can make that, but **we cannot make a transformer for 'any' monad with 'any' other monad inside**. **That's not possible.**

Monads are **not composable**.
We **cannot make** a single recipe that takes two **monads** and **transforms** them into a new **monad**.
We **cannot make** a generic **transformer** for 'any' **monad** with 'any' nested **monad**.

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

Functors compose. i.e. a generic **Functor** can be written that is the **composition** of any two other **Functors**.

The same is true for **Applicatives**: they also **compose**.

But it turns out that **Monads** <u>**do not**</u> **compose**.

See the next two slides for how **FPiS** puts it, and how it says that **monad transformers** are a way of addressing this problem.

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

Functor
 	
map
	`trait Functor[F[_]] {`
	
	`def map[A, B](m: F[A], f: A ⇒ B): F[B]`

Applicative
 	
unit
map2
	`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))`

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`

EXERCISE 12.11

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

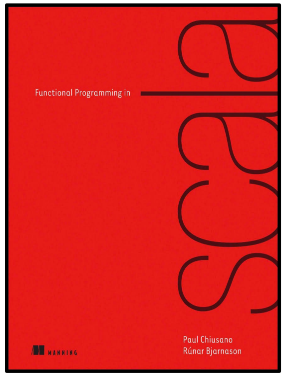
```
def compose[G[_]](G: Monad[G]): Monad[({type f[x] = F[G[x]]})#f]
```

Answer to Exercise 12.11

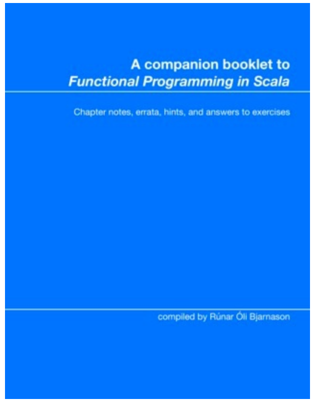
You want to try writing **flatMap** in terms of **Monad**[F] and **Monad**[G].

```
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 <u>need to be able to "**swap**" the F and G types</u>. In other words, you'd need a **distributive law**. <u>Such an operation is not part of the **Monad** interface</u>.



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



A companion booklet to
Functional Programming in Scala
(by Runar Bjarnason)
@runarorama

Earlier, when we tried to implement **flatMap** for **AnyMonadMonad**, we couldn't because we weren't able to **swap** M with N in M[N[B]] to allow **n.flatMap** to return an N[_]

```
def flatMap[B](f: A => AnyMonadMonad[M, N, B]): AnyMonadMonad[M, N, B] =
  AnyMonadMonad {
    m.flatMap(inner:M[N[A]]){ na:N[A] =>
      n.flatMap(na){ a:A => val mnb: M[N[B]] = f(a).inner; mnb }
    }
  }
```

Expression of type M[N[B]] doesn't conform to expected type N[B_]

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.

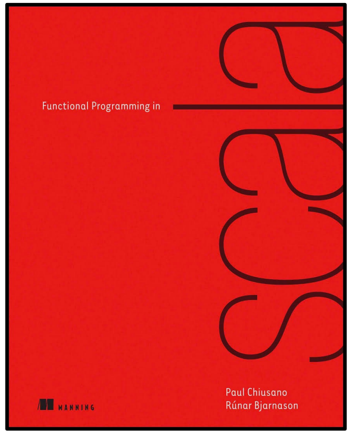
The issue of composing monads is often addressed with a custom-written version of each monad that's specifically constructed for composition. This kind of thing is called a **monad transformer**. For example, the `OptionT` monad transformer composes `Option` with any other **monad**:

```scala
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**.



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

If you want to know more about how **Functors** and **Applicatives** **compose** but **Monads** do not then see the following

slideshare @philip_schwarz

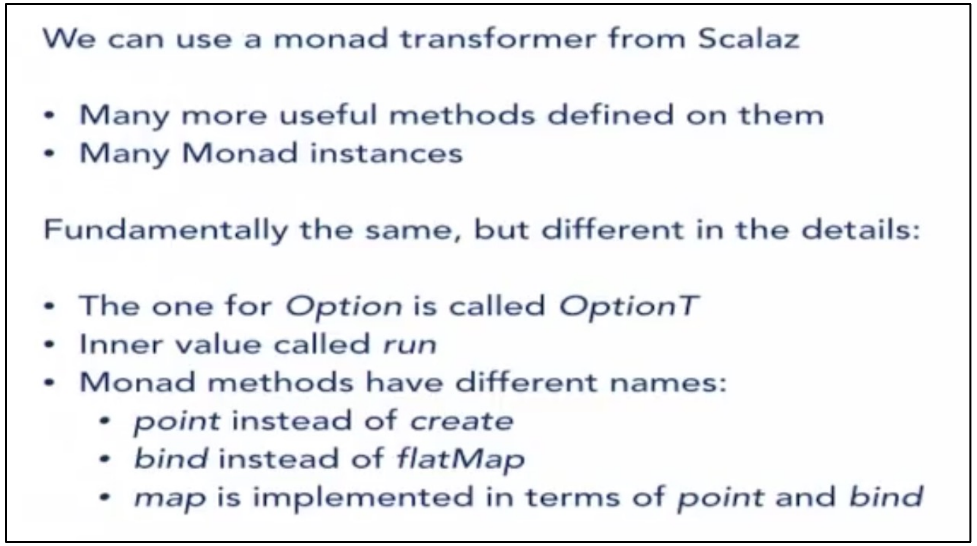
https://www.slideshare.net/pjschwarz/ https://www.slideshare.net/pjschwarz/monads-do-not-compose

We've made a monad transformer, we've defined a monad trait, it is all very easy, easily fits on a single slide.

So hopefully you feel comfortably now that monad transformers are not a very hard concept.

But **you don't necessarily have to define them yourself in your code**, of course. **We could, for example use the ones defined in the scalaz library**. They have many more methods defined on them beside **map** and **flatMap** and they also provide many instances for **monads**, so they have the instance of the **monad** trait for **List**, for **Option**, for **Future**, etc, which is very useful.

But they are fundamentally the same stuff as we just built, different in the details.



We can use a monad transformer from Scalaz

- Many more useful methods defined on them
- Many Monad instances

Fundamentally the same, but different in the details:

- The one for *Option* is called *OptionT*
- Inner value called *run*
- Monad methods have different names:
  - *point* instead of *create*
  - *bind* instead of *flatMap*
  - *map* is implemented in terms of *point* and *bind*

to be continued in part 2