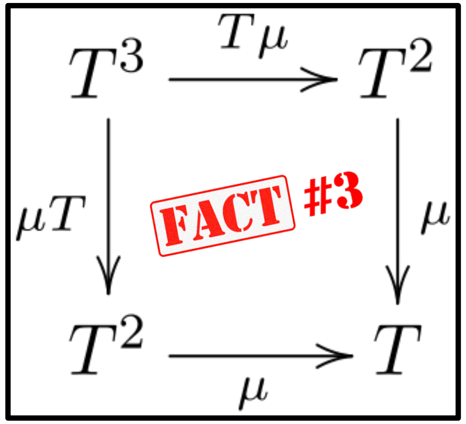
MONAD FACT #4

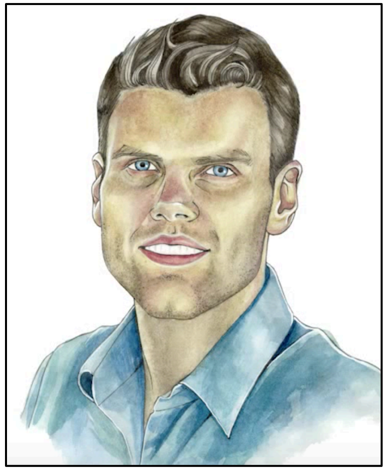
a **monad** is an implementation of one of the **minimal sets** of **monadic combinators**, satisfying the laws of **associativity** and **identity**

see how **compositional responsibilities** are distributed in each **combinator set**



slides by  @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz



Runar Bjarnason

@runarorama

We've seen three minimal sets of primitive `Monad` combinators, and instances of `Monad` will have to provide implementations of one of these sets:

- `unit` and `flatMap`
- `unit` and `compose`
- `unit`, `map`, and `join`

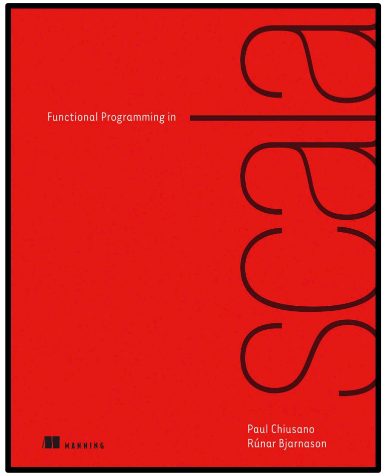
And we know that there are two monad laws to be satisfied, associativity and identity, that can be formulated in various ways. **So we can state plainly what a monad is**:

> A monad is an implementation of one of the minimal sets of monadic combinators, satisfying the laws of associativity and identity.

That's a perfectly respectable, precise, and terse definition. And **if we're being precise, this is the only correct definition**. A monad is precisely defined by its operations and laws; no more, no less.

Paul Chiusano

@pchiusano



Functional Programming in Scala

by **Paul Chiusano** and **Runar Bjarnason**

@philip_schwarz

One of the minimal sets of primitive Monad combinators seen on the previous slide consists of a unit function and a compose function.

The compose function in question is Kleisli composition.

If you need an introduction to Kleisli composition then see **MONAD FACT #2**.

If you need an introduction to the unit function then see **MONAD FACT #1**.

Another set of combinators includes the join function. In Scala this function is known as flatten.

Let's take the simplest monad, i.e. the identity monad, which does nothing, and let's define it in terms of Kleisli composition and unit.

The Id monad wraps a value of some type A

```
case class Id[A](value: A)
```

Id also acts as the unit function. i.e. to lift the value 3 into the Id monad we use Id(3).

Now we have to come up with a body for the Kleisli composition function (shown below as the infix fish operator >=>):

```
implicit class IdFunctionOps[A,B](f: A => Id[B]) {
  def >=>[C](g: B => Id[C]): A => Id[C] = ???
}
```

The body must be a function of type A => Id[C]

```
a => ???
```

The only way we can get an Id[C] is by calling g, which takes a B as a parameter. But all we have to work with are the a parameter, which is of type A, and function f. But that is fine because if we call f with a we get an Id[B] and if we then ask the latter for the B value that it wraps, we have the B that we need to invoke g.

```
a => g(f(a).value)
```

So here is how we define Kleisli composition

```
implicit class IdFunctionOps[A,B](f: A => Id[B]) {
  def >=>[C](g: B => Id[C]): A => Id[C] =
    a => g(f(a).value)
}
```

And here is a simple test for the function

```
val double: Int => Id[Int] = n => Id(n * 2)
val square: Int => Id[Int] = n => Id(n * n)
assert( (double >=> square)(3) == Id(36))
```

So yes, we have defined the **identity monad** in terms of **Kleisli composition** and **unit**.

```scala
case class Id[A](value: A)

object Id {

  implicit class IdFunctionOps[A,B](f: A => Id[B]) {
      def >=>[C](g: B => Id[C]): A => Id[C] =
        a => g(f(a).value)
    }
}
```

But we want to be able to use the **monad** in a **for comprehension**, so we now have to define a **flatMap** function and a **map** function. The **flatMap** function can be defined in terms of **Kleisli composition**:

```scala
case class Id[A](value: A) {

  def flatMap[B](f: A => Id[B]): Id[B] =
    (((_:Unit) => this) >=> f)(())

}
```

and **map** can then be defined in terms of **flatMap**:

```scala
case class Id[A](value: A) {

  def flatMap[B](f: A => Id[B]): Id[B] =
    (((_:Unit) => this) >=> f)(())

  def map[B](f: A => B): Id[B] =
    this flatMap { a => Id(f(a)) }
}
```

Here is a test for our **map** function

```scala
val increment: Int => Int = n => n + 1
assert( (Id(3) map increment) == Id(4) )
```

And here is a test for both our **map** and **flatMap** functions

```scala
val result =
  for {
    six       <- double(3)
    thirtySix <- square(six)
  } yield six + thirtySix
assert(result == Id(42))
```

We can also define **join** (aka **flatten**) in terms of the **flatMap** function

```scala
def join[A](mma: Id[Id[A]]): Id[A] =
  mma flatMap identity
```

Here is a simple test for **join**

```scala
assert( join(Id(Id(3))) == Id(3) )
```

Here is the whole code for the **identity monad**

```scala
case class Id[A](value: A) {

  def flatMap[B](f: A => Id[B]): Id[B] =
    (((_:Unit) => this) >=> f)(())

  def map[B](f: A => B): Id[B] =
    this flatMap { a => Id(f(a)) }
}

object Id {

  implicit class IdFunctionOps[A,B](f: A => Id[B]) {
    def >=>[C](g: B => Id[C]): A => Id[C] =
      a => g(f(a).value)
  }

  def join[A](mma: Id[Id[A]]): Id[A] =
    mma flatMap identity
}
```

```scala
val double: Int => Id[Int] = n => Id(n * 2)
val square: Int => Id[Int] = n => Id(n * n)

assert( (double >=> square)(3) == Id(36))

val increment: Int => Int = n => n + 1

assert( (Id(3) map increment) == Id(4) )

assert( join(Id(Id(3))) == Id(3) )

val result =
  for {
    six       <- double(3)
    thirtySix <- square(six)
  } yield six + thirtySix

assert(result == Id(42))
```

In this slide deck we are going to compare the **identity monad** with the **Option monad** and the **List monad**.

How do the functions of the above **identity monad**, which is defined in terms of **Kleisli composition**, relate to the equivalent **Option monad** functions?

See the next slide for the differences.

@philip_schwarz

First of all we see that apart from the obvious swapping of **Id** for **Option** in their signatures, the **flatMap** and **join** functions of the two **monads** are identical.

The only difference between the **map** functions of the two **monads** are the **unit** functions that they use: one uses **Id** and the other uses **Some**.

So **the only real difference between the two monads is the logic in the fish operator**. That makes sense, since the **monads** are defined in terms of **unit** and **Kleisli composition**, and since **unit** is a very simple function.

```scala
case class Id[A](value: A) {

  def flatMap[B](f: A => Id[B]): Id[B] =
    (((_:Unit) => this) >=> f)(())

  def map[B](f: A => B): Id[B] =
    this flatMap { a => Id(f(a)) }
}

object Id {

  implicit class IdFunctionOps[A,B](f: A => Id[B]) {
    def >=>[C](g: B => Id[C]): A => Id[C] =
      a => ???
  }

  def join[A](mma: Id[Id[A]]): Id[A] =
    mma flatMap identity
}
```

```scala
sealed trait Option[+A] {

  def flatMap[B](f: A => Option[B]): Option[B] =
    (((_:Unit) => this) >=> f)(())

  def map[B](f: A => B): Option[B] =
    this flatMap { a => Some(f(a)) }
}
case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]

object Option {

  implicit class OptionFunctionOps[A, B](f: A => Option[B]) {
    def >=>[C](g: B => Option[C]): A => Option[C] =
      a => ???
  }

  def join[A](mma: Option[Option[A]]): Option[A] =
    mma flatMap identity
}
```

The same is true of the differences between the functions of the **Id monad** and those of the **List monad**: the differences are in the **fish operator**;

The apparent additional difference between the **map** functions is only due to the fact that we are using **Cons**(x,**Nil**) as a **unit** function rather **List**(x), i.e. some singleton list constructor that we could define.

```
case class Id[A](value: A) {

  def flatMap[B](f: A => Id[B]): Id[B] =
    (((_:Unit) => this) >=> f)(())

  def map[B](f: A => B): Id[B] =
    this flatMap { a => Id(f(a)) }

}

object Id {

  implicit class IdFunctionOps[A,B](f: A => Id[B]) {
    def >=>[C](g: B => Id[C]): A => Id[C] =
      a => ???
  }

  def join[A](mma: Id[Id[A]]): Id[A] =
    mma flatMap identity

}
```

```
sealed trait List[+A] {

  def flatMap[B](f: A => List[B]): List[B] =
    (((_:Unit) => this) >=> f)(())

  def map[B](f: A => B): List[B] =
    this flatMap { a => Cons(f(a),Nil) }

}
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List {

  implicit class ListFunctionOps[A,B](f: A => List[B]) {
    def >=>[C](g: B => List[C]): A => List[C] =
      a => ???
  }

def join[A](mma: List[List[A]]): List[A] =
  mma flatMap identity

}
```

Let's now turn to the function that differentiates the **monads**, i.e. **Kleisli composition** (the **fish operator**)

```
implicit class IdFunctionOps[A,B](f: A => Id[B]) {
  def >=>[C](g: B => Id[C]): A => Id[C] =
    a => ???
}
```

The **composite function** that it returns (the **composition** of **f** and **g**) has the following **responsibilities** (let's call them **compositional responsibilities**):

1) use **f** to compute a **first value** **wrapped** in a **functional effect**
2) dig underneath the **wrapper** to access the **first value**, discarding the **wrapper**
3) Use **g** to compute, using the **first value**, a **second value** also **wrapped** in a **functional effect**
4) return a **third value** **wrapped** in a **functional effect** that represents the **composition** (**combination**) of the **first two functional effects**

As a slight variation on that, we can replace '**wrapped** in' with 'in the **context** of'

1) use **f** to compute a **first value** in the **context** of a **functional effect**
2) dig inside the **context** to access the **first value**, discarding the **context**
3) Use **g** to compute, using the **first value**, a **second value** also in the **context** of a **functional effect**
4) return a **third value** in the **context** of a **functional effect** that represents the **composition** (**combination**) of the **first two functional effects**

Here are the **Kleisli composition** functions of the three **monads** (their **fish operators**).

Notice how different they are. The one in the **identity monad** seems to do almost nothing, the one in the **Option monad** seems to do a bit more work, and the one in the **List monad** does quite a bit more.

See the next slide for some test code for the **Option monad** and the **List monad**.

```scala
object Id {

  implicit class IdFunctionOps[A,B](f: A => Id[B]) {
    def >=>[C](g: B => Id[C]): A => Id[C] =
      a => g(f(a).value)
  }

}
```

```scala
object Option {

  implicit class OptionFunctionOps[A, B](f: A => Option[B]) {
    def >=>[C](g: B => Option[C]): A => Option[C] =
      a => f(a) match {
        case Some(b) => g(b)
        case None => None
      }
  }

}
```

```scala
sealed trait List[+A] {

  def foldRight[B](b: B, f: (A,B) => B): B =
    this match {
      case Nil => b
      case Cons(a, tail) =>
        f(a, tail.foldRight(zero, f))
    }

}

object List {

  implicit class ListFunctionOps[A,B](f: A => List[B]) {
    def >=>[C](g: B => List[C]): A => List[C] =
      a => f(a).foldRight(Nil,
                          (b:B, cs:List[C]) => concatenate(g(b), cs))
  }

  def concatenate[A](left: List[A], right: List[A]): List[A] =
    left match {
      case Nil => right
      case Cons(head, tail) => Cons(head, concatenate(tail, right))
    }
}
```

```scala
// Tests for Option monad

assert( join(Some(Some(3))) == Some(3) )

val increment: Int => Int = n => n + 1

assert( (Some(3) map increment) == Some(4) )

val double: Int => Option[Int] =
  n => if (n % 2 == 1) Some(n * 2) else None
val square: Int => Option[Int] =
  n => if (n < 100) Some(n * n) else None

assert( (double >=> square)(3) == Some(36))

val result =
  for {
    six       <- double(3)
    thirtySix <- square(six)
  } yield six + thirtySix

assert(result == Some(42))
```

```scala
// Tests for List monad

assert( join(Cons(
               Cons(1, Cons(2, Nil)),
               Cons(
                 Cons(3, Cons(4, Nil)),
                 Nil))
        ) == Cons(1, Cons(2, Cons(3, Cons(4, Nil))) ) )

val increment: Int => Int = n => n + 1

assert( (Cons(1, Cons(2, Cons(3, Cons(4, Nil))) ) map increment)
        == Cons(2, Cons(3, Cons(4, Cons(5, Nil))) ) )

val double: Int => List[Int] = n => Cons(n, Cons(n * 2, Nil))
val square: Int => List[Int] = n => Cons(n, Cons(n * n, Nil))

assert( (double >=> square)(3) == Cons(3,Cons(9,Cons(6,Cons(36, Nil)))))

val result =
  for {
    x <- double(3)
    y <- square(x)
  } yield Cons(x, Cons(y, Nil))

assert(result == Cons(
                   Cons(3,Cons(3,Nil)),
                   Cons(
                     Cons(3,Cons(9,Nil)),
                     Cons(
                       Cons(6,Cons(6,Nil)),
                       Cons(
                         Cons(6,Cons(36,Nil)),
                         Nil)))))
```

@philip_schwarz

Next, we are going to look at the **Kleisli composition** functions of `Id`, `Option` and `List` to see how each of them discharges its **compositional responsibilities**.

In the special case of the **Identity monad**, which does nothing, the **compositional responsibilities** are discharged in a degenerate and curious way:

```
implicit class IdFunctionOps[A,B](f: A => Id[B]) {
  def >=>[C](g: B => Id[C]): A => Id[C] =
    a => g(f(a).value)
}
```

1) use **f** to compute a **first value wrapped** in a **functional effect**

   just call **f**
   `f(a)`

2) dig underneath the **wrapper** to access the **first value**, discarding the **wrapper**

   digging under the wrapper simply amounts to asking the resulting `Id[B]` for the `B` that it is wrapping
   `f(a).value`

3) use **g** to compute, using the **first value**, a **second value** also **wrapped** in a **functional effect**

   just call **g** with the **first value**
   `g(f(a).value)`

4) return a **third value wrapped** in a **functional effect** that represents the **composition** (**combination**) of the **first two functional effects**

   because the effect of the **Id monad** is nonexistent, there simply is nothing to combine, so just return the **second value**
   `g(f(a).value)`

Next, let's look at how the **compositional responsibilities** are discharged in the **Option monad**:

```
implicit class OptionFunctionOps[A, B](f: A => Option[B]) {
  def >=>[C](g: B => Option[C]): A => Option[C] =
    a => f(a) match {
      case Some(b) => g(b)
      case None    => None
    }
}
```

1) use **f** to compute a **first value wrapped** in a **functional effect**

   just call **f**
   `f(a)`

2) dig underneath the **wrapper** to access the **first value**, discarding the **wrapper**

   digging under the **wrapper** and discarding it is done by pattern matching, destructuring `Option[B]` to get the **wrapped** B value
   `Some(b)`

3) use **g** to compute, using the **first value**, a **second value** also **wrapped** in a **functional effect**

   just call **g** with the **first value**
   `g(b)`

4) return a **third value wrapped** in a **functional effect** that represents the **composition** (**combination**) of the **first two functional effects**

   If the **1st effect** is that **a value is defined** then the **3rd value** is just the **2nd value** and **composition** of the **1st effect** with the **2nd effect** is just the **2nd effect**
   `case Some(b) => g(b)`
   If the **1st** effect is that **no value is defined** then there is no **3rd value** as the **composition** of the **1st** and **2nd effects** is just the **1st effect**
   `case None => None`

Let's now look at how the **compositional responsibilities** are discharged in the **List monad**:

```
implicit class ListFunctionOps[A,B](f: A => List[B]) {
  def >=>[C](g: B => List[C]): A => List[C] =
    a => f(a).foldRight(Nil, (b:B, cs:List[C]) => concatenate(g(b), cs))
}
```

```
def foldRight[B](b: B,
                 f:(A, B) => B): B =
  this match {
    case Nil => b
    case Cons(a, tail) =>
      f(a, tail.foldRight(b, f))
  }

def concatenate[A](left: List[A],
                   right: List[A]): List[A] =
  left match {
    case Nil =>
      right
    case Cons(head, tail) =>
      Cons(head, concatenate(tail, right))
  }
```

1) use **f** to compute a **first value wrapped** in a **functional effect**

just call **f** – the **first value** consists of the B items in the resulting `List[B]`

`f(a)`

2) dig underneath the **wrapper** to access the **first value**, discarding the **wrapper**

digging under the **wrapper** and discarding it is done by **foldRight**, which calls its callback function with each B item in the **first value**

`f(a).foldRight(Nil, (b:B, cs:List[C]) => concatenate(g(b), cs))`

3) use **g** to compute, using the **first value**, a **second value** also **wrapped** in a **functional effect**

callback function **g** is called with each B item in the **first value**, so the **second value** consists of all `List[C]` results returned by **g**

`(b:B, …) => …(g(b), …))`

4) return a **third value wrapped** in a **functional effect** that represents the **composition** (**combination**) of the **first two functional effects**

If the **1st effect** is that there are no B items then there are no **2nd** and **3rd values** and the **composition** of **1st** and **2nd effect** is also that there are no items

`f(a)` is `Nil` so `f(a).foldright(…)` is also `Nil`

otherwise the **1st effect** is the multiplicity of items in the **1st value**, the **2nd effect** is the multiplicity of items in the **2nd value**, the **3rd value** is the concatenation of all the `List[C]` results returned by **g**, and the **composition** of the **1st** and **2nd effects** is the multiplicity of items in the concatenation

`f(a).foldRight(Nil, (b:B, cs:List[C]) => concatenate(g(b), cs))`

@philip_schwarz

What we have done so far is take three **monads** and define them in terms of **Kleisli composition** and **unit**.

In the next three slides we are going to **refactor the three monads so that they are defined in terms of flatMap** and **unit** and **see how the compositional responsibilities get redistributed**.

COMPOSITIONAL RESPONSIBILITY

1. use **f** to compute a **first value wrapped** in a **functional effect**
2. dig underneath the **wrapper** to access the **first value**, discarding the **wrapper**
3. use **g** to compute, using the **first value**, a **second value** also **wrapped** in a **functional effect**
4. return a **third value wrapped** in a **functional effect** that represents the **composition (combination)** of the **first two functional effects**

LOCATION CHANGE

remains in **>=>**
moves from **>=>** to **flatMap**
moves from **>=>** to **flatMap**
moves from **>=>** to **flatMap**

Id monad defined in terms of Kleisli composition and unit

```scala
case class Id[A](value: A) {

  def flatMap[B](f: A => Id[B]): Id[B] =
    (((_:Unit) => this) >=> f)(())

  def map[B](f: A => B): Id[B] =
    this flatMap { a => Id(f(a)) }
}

object Id {

  implicit class IdFunctionOps[A,B](f: A => Id[B]) {
    def >=>[C](g: B => Id[C]): A => Id[C] =
      a => g(f(a).value)
  }

  def join[A](mma: Id[Id[A]]): Id[A] =
    mma flatMap identity
}
```

invoke 1st function
invoke 2nd function
dig underneath wrapper
compose effects
discard wrapper

Id monad defined in terms of flatMap and unit

```scala
case class Id[A](value: A) {

  def flatMap[B](f: A => Id[B]): Id[B] =
    f(value)

  def map[B](f: A => B): Id[B] =
    this flatMap { a => Id(f(a)) }
}

object Id {

  implicit class IdFunctionOps[A,B](f: A => Id[B]) {
    def >=>[C](g: B => Id[C]): A => Id[C] =
      a => f(a) flatMap g
  }

  def join[A](mma: Id[Id[A]]): Id[A] =
    mma flatMap identity
}
```

invoke 2nd function
dig underneath wrapper
compose effects
discard wrapper
invoke 1st function

Option monad defined in terms of **Kleisli composition** and **unit**

```scala
sealed trait Option[+A] {

  def flatMap[B](f: A => Option[B]): Option[B] =
    (((_:Unit) => this) >=> f)(())

  def map[B](f: A => B): Option[B] =
    this flatMap { a => Some(f(a)) }
}
case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]

object Option {

  implicit class OptionFunctionOps[A,B](f: A => Option[B]) {
    def >=>[C](g: B => Option[C]): A => Option[C] =
      a => f(a) match {
        case Some(b) => g(b)
        case None => None
      }
  }

  def join[A](mma: Option[Option[A]]): Option[A] =
    mma flatMap identity
}
```

invoke 1st function

invoke 2nd function

dig underneath wrapper

compose effects

discard wrapper

Option monad defined in terms of **flatMap** and **unit**

```scala
sealed trait Option[+A] {

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

  def map[B](f: A => B): Option[B] =
    this flatMap { a => Some(f(a)) }
}
case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]

object Option {

  implicit class OptionFunctionOps[A,B](f: A => Option[B]) {
    def >=>[C](g: B => Option[C]): A => Option[C] =
      a => f(a) flatMap g
  }

  def join[A](mma: Option[Option[A]]): Option[A] =
    mma flatMap identity
}
```

dig underneath wrapper

compose effects

invoke 2nd function

discard wrapper

invoke 1st function

List monad defined in terms of Kleisli composition and unit

```
sealed trait List[+A] {

  def flatMap[B](f: A => List[B]): List[B] =
    (((_:Unit) => this) >=> f)(())

  def map[B](f: A => B): List[B] =
    this flatMap { a => Cons(f(a),Nil) }

  def foldRight[B](b: B, f: (A,B) => B): B =
    this match {
      case Nil => b
      case Cons(a, tail) =>
        f(a, tail.foldRight(zero, f))
    }
}
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List {

  implicit class ListFunctionOps[A,B](f: A => List[B]) {
    def >=>[C](g: B => List[C]): A => List[C] =
      a => f(a).foldRight(Nil, (b:B, cs:List[C]) => concatenate(g(b), cs))
  }

  def join[A](mma: List[List[A]]): List[A] =
    mma flatMap identity

  def concatenate[A](left: List[A], right: List[A]): List[A] =
    left match {
      case Nil => right
      case Cons(head, tail) => Cons(head, concatenate(tail, right))
    }
}
```

invoke 2nd function

invoke 1st function | dig underneath wrapper | compose effects | discard wrapper

List monad defined in terms of flatMap and unit

```
sealed trait List[+A] {

  def flatMap[B](f: A => List[B]): List[B] =
    this foldRight(Nil, (a:A, bs:List[B]) => concatenate(f(a), bs))

  def map[B](f: A => B): List[B] =
    this flatMap { a => Cons(f(a),Nil) }

  def foldRight[B](b: B, f: (A,B) => B): B =
    this match {
      case Nil => b
      case Cons(a, tail) =>
        f(a, tail.foldRight(zero, f))
    }
}
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List {

  implicit class ListFunctionOps[A,B](f: A => List[B]) {
    def >=>[C](g: B => List[C]): A => List[C] =
      a => f(a) flatMap g
  }

  def join[A](mma: List[List[A]]): List[A] =
    mma flatMap identity

  def concatenate[A](left: List[A], right: List[A]): List[A] =
    left match {
      case Nil => right
      case Cons(head, tail) => Cons(head, concatenate(tail, right))
    }
}
```

dig underneath wrapper

invoke 2nd function

compose effects

discard wrapper

invoke 1st function

@philip_schwarz

And finally, we are going to **refactor the three monads so that they are defined in terms of map**, **join** and **unit** and again **see how the compositional responsibilities get redistributed.**

COMPOSITIONAL RESPONSIBILITY

1. use **f** to compute a **first value wrapped** in a **functional effect**
2. dig underneath the **wrapper** to access the **first value**, discarding the **wrapper**
3. use **g** to compute, using the **first value**, a **second value** also **wrapped** in a **functional effect**
4. return a **third value wrapped** in a **functional effect** that represents the **composition (combination)** of the **first two functional effects**

LOCATION CHANGE

remains in **>=>**

moves from **flatMap** to **map/join**

moves from **flatMap** to **map**

moves from **flatMap** to **join**

Id monad defined in terms of flatMap and unit

```scala
case class Id[A](value: A) {

  def flatMap[B](f: A => Id[B]): Id[B] =
      f(value)

  def map[B](f: A => B): Id[B] =
      this flatMap { a => Id(f(a)) }
}

object Id {

implicit class IdFunctionOps[A,B](f: A => Id[B]) {
  def >=>[C](g: B => Id[C]): A => Id[C] =
    a => f(a) flatMap g
}

  def join[A](mma: Id[Id[A]]): Id[A] =
    mma flatMap identity
}
```

invoke 2nd function

dig underneath wrapper

compose effects

discard wrapper

invoke 1st function

Id monad defined in terms of map, join and unit

```scala
case class Id[A](value: A) {

  def flatMap[B](f: A => Id[B]): Id[B] =
    join(this map f)

  def map[B](f: A => B): Id[B] =
    Id(f(value))
}

object Id {

implicit class IdFunctionOps[A,B](f: A => Id[B]) {
  def >=>[C](g: B => Id[C]): A => Id[C] =
    a => f(a) flatMap g
}

  def join[A](mma: Id[Id[A]]): Id[A] =
    mma.value
}
```

dig underneath wrapper

invoke 2nd function

invoke 1st function

discard wrapper

compose effects

Option monad defined in terms of **flatMap** and **unit**

```
sealed trait Option[+A]  {

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

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

}

case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]

object Option {

  def join[A](mma: Option[Option[A]]): Option[A] =
    mma flatMap identity

  implicit class IdFunctionOps[A,B](f: A => Option[B]) {
    def >=>[C](g: B => Option[C]): A => Option[C] =
      a => f(a) flatMap g
  }

}
```

invoke 2nd function

dig underneath wrapper

compose effects

discard wrapper

invoke 1st function

Option monad defined in terms of **map**, **join** and **unit**

```
sealed trait Option[+A]  {

  def flatMap[B](f: A => Option[B]): Option[B] =
    join(this map f)

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

case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]

object Option {

  def join[A](mma: Option[Option[A]]): Option[A] =
    mma match {
      case Some(ma) => ma
      case None => None
    }

  implicit class IdFunctionOps[A,B](f: A => Option[B]) {
    def >=>[C](g: B => Option[C]): A => Option[C] =
      a => f(a) flatMap g
  }

}
```

dig underneath wrapper

invoke 2nd function

discard wrapper

compose effects

invoke 1st function

List monad defined in terms of **flatMap** and **unit**

```scala
sealed trait List[+A] {

  def flatMap[B](f: A => List[B]): List[B] =
    this foldRight(Nil, (a:A, bs:List[B]) => concatenate(f(a), bs))

  def map[B](f: A => B): List[B] =
    this flatMap { a => Cons(f(a),Nil) }

  def foldRight[B](zero: B, f: (A,B) => B): B =
    this match {
      case Nil => zero
      case Cons(a, tail) =>
        f(a, tail.foldRight(zero, f))
    }

}
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List {

  implicit class ListFunctionOps[A,B](f: A => List[B]) {
    def >=>[C](g: B => List[C]): A => List[C] =
      a => f(a) flatMap g
  }

  def join[A](mma: List[List[A]]): List[A] =
    mma flatMap identity

  def concatenate[A](left: List[A], right: List[A]): List[A] =
    left match {
      case Nil => right
      case Cons(head, tail) => Cons(head, concatenate(tail, right))
    }
}
```

invoke 2nd function

dig underneath wrapper

compose effects

discard wrapper

invoke 1st function

List monad defined in terms of **map**, **join** and **unit**

```scala
sealed trait List[+A] {

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

  def map[B](f: A => B): List[B] =
    this.foldRight(Nil, (a:A,bs:List[B]) => Cons(f(a),bs))

  def foldRight[B](zero: B, f: (A,B) => B): B =
    this match {
      case Nil => zero
      case Cons(a, tail) =>
        f(a, tail.foldRight(zero, f))
    }

}
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List {

  implicit class ListFunctionOps[A,B](f: A => List[B]) {
    def >=>[C](g: B => List[C]): A => List[C] =
      a => f(a) flatMap g
  }

  def join[A](mma: List[List[A]]): List[A] =
    mma.foldRight(Nil, (as: List[A], bs:List[A]) => concatenate(as,bs) )

  def concatenate[A](left: List[A], right: List[A]): List[A] =
    left match {
      case Nil => right
      case Cons(head, tail) => Cons(head, concatenate(tail, right))
    }
}
```

dig underneath wrapper

invoke 2nd function

invoke 1st function

discard wrapper

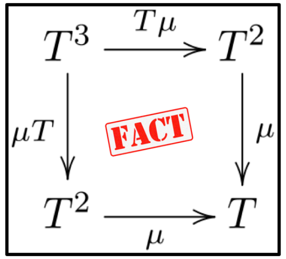
compose effects

See the following slide deck for the list of all available decks in the **MONAD FACT** series

The MONAD FACT Slide Deck Series

a very simple rationale for the series

plus a list of currently available slide decks



slides by  @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

slideshare https://www.slideshare.net/pjschwarz