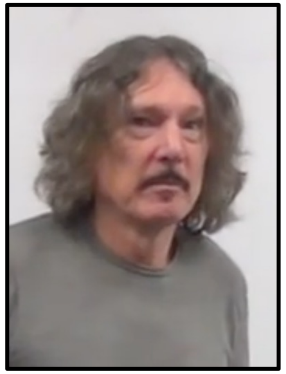
Kleisli composition, flatMap, join, map, unit

a study/memory aid

to help learn/recall their implementation/interrelation

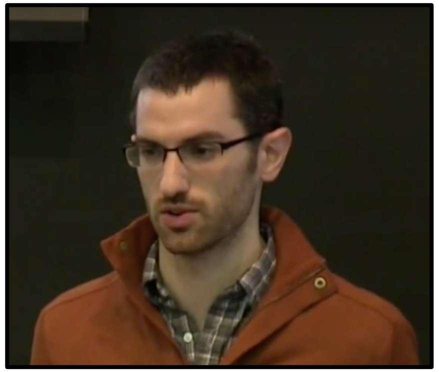
inspired by, and based on, the work of

Rob Norris
@tpolecat

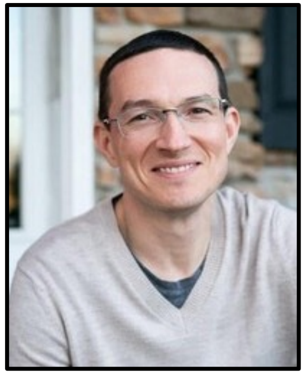


Bartosz Milewski
@BartoszMilewski

Runar Bjarnason
@runarorama



Paul Chiusano
@pchiusano



Michael Pilquist
@mpilquist

slides by

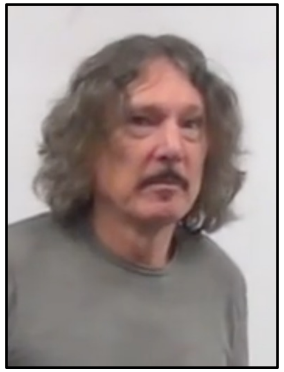
@philip_schwarz FP Illuminated http://fpilluminated.com/

VERSION 2 – UPDATED FOR SCALA 3

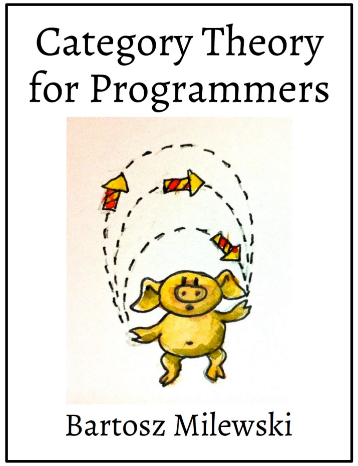
This slide deck is meant both for (1) those who are familiar with the monadic functions that are **Kleisli composition**, **unit**, **map**, **join** and **flatMap**, and want to reinforce their knowledge (2) and as a memory aid, for those who sometimes need a reminder of how these functions are implemented and how they interrelate.

I learned about this subject mainly from **Functional Programming in Scala**, from **Bartosz Milewski**'s YouTube videos (and his book), and from **Rob Norris**'s YouTube video, **Functional Programming with Effects**.

Runar Bjarnason
@runarorama



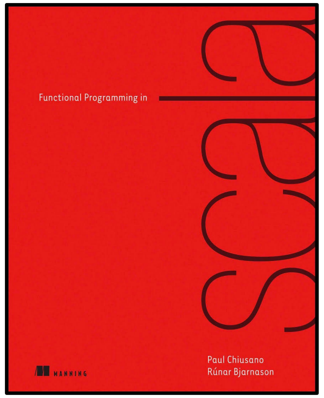
Bartosz Milewski
@BartoszMilewski



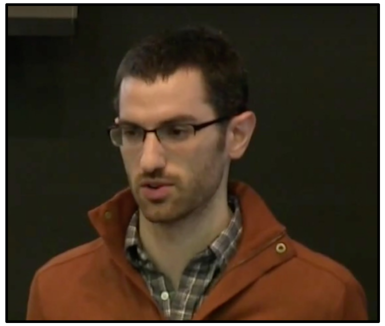
Rob Norris
@tpolecat

@philip_schwarz

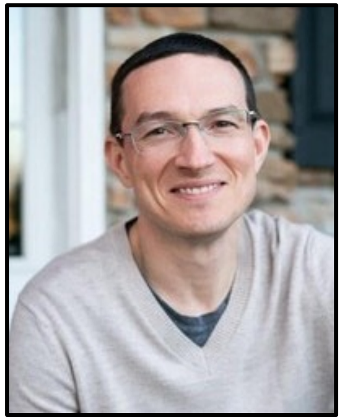
See the final slide of this deck for some of the inspiration/ideas I got from **Rob Norris**'s video.



Functional Programming In Scala



Paul Chiusano
@pchiusano



Michael Pilquist
@mpilquist

If you need an intro to, or refresher on, the monadic functions that are **Kleisli composition**, **unit**, **map**, **join** and **flatMap**, then see the following

slideshare

https://www.slideshare.net/pjschwarz/rob-norrisfunctionalprogrammingwitheffects

https://www.slideshare.net/pjschwarz/kleisli-monad-as-functor-with-pair-of-natural-transformations

https://www.slideshare.net/pjschwarz/fish-operator-anatomy

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

https://www.slideshare.net/pjschwarz/compositionality-and-category-theory-a-montage-of-slidestranscript-for-sections-of-rnar-bjarnasons-keynote-composing-programs

A simple example of **hand-coding Kleisli composition** (i.e. >=>, the fish operator) for Option and List

Option

```scala
extension [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
case class Insurance(name:String)
case class Car(insurance: Option[Insurance])
case class Person(car: Option[Car])

val car: Person => Option[Car] =
  person => person.car

val insurance: Car => Option[Insurance] =
  car => car.insurance
```

```scala
val carInsurance: Person => Option[Insurance] =
  car >=> insurance
```

```scala
val nonDriver= Person(car=None)
val uninsured = Person(Some(Car(insurance=None)))
val insured = Person(Some(Car(Some(Insurance("Acme")))))

assert(carInsurance(nonDriver).isEmpty)
assert(carInsurance(uninsured).isEmpty)
assert(carInsurance(insured).contains(Insurance("Acme")))
```

List

```scala
extension [A,B](f: A => List[B])
  def >=>[C](g: B => List[C]): A => List[C] =
    a => f(a).foldRight(List.empty[C]):
      (b, cs) => g(b) ++ cs
```

```scala
val toChars: String => List[Char] = _.toList
val toAscii: Char => List[Char] = _.toInt.toString.toList

assert( toChars("AB") == List('A','B'))
assert( toAscii('A') == List('6','5') )
```

```scala
val toCharsAscii: String => List[Char] =
  toChars >=> toAscii
```

```scala
assert(toCharsAscii("AB") == List('6','5','6','6'))
```

We have implemented >=> by hand. Twice. Once for **Option** and once for **List**.

Now let's make >=> generic and implement it in terms of the **built-in** **flatMap** function of the **Option** and **List** **Monads**.

Before

>=> is **hand-coded** and specialised for **Option** and **List**

```scala
extension [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

extension [A,B](f: A => List[B])
  def >=>[C](g: B => List[C]): A => List[C] =
    a => f(a).foldRight(List.empty[C]):
      (b, cs) => g(b) ++ cs
```

After

>=> is **generic** and defined in terms of **built-in flatMap**

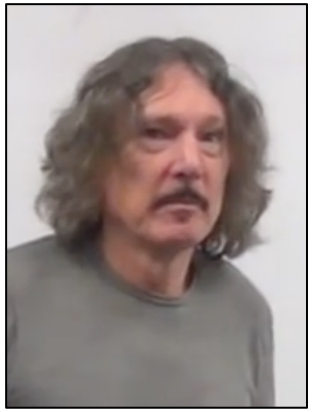
```scala
extension [A,B,F[_]: Monad](f: A => F[B])
  def >=>[C](g: B => F[C]): A => F[C] =
    a => f(a).flatMap(g)
```

```scala
trait Monad[F[_]]:
  def unit[A](a: => A): F[A]
  extension [A](fa: F[A])
    def flatMap[B](f: A => F[B]): F[B]

given Monad[Option] with
  def unit[A](a: => A): Option[A] = Some(a)
  extension [A](fa: Option[A])
    def flatMap[B](f: A => Option[B]): Option[B] = fa.flatMap(f)

given Monad[List] with
  def unit[A](a: => A): List[A] = List(a)
  extension [A](fa: List[A])
    def flatMap[B](f: A => List[B]): List[B] = fa.flatMap(f)
```

I first saw the definition of >=> in a YouTube video by **Bartosz Milewski**.



Bartosz Milewski's definition of the **fish operator** (**Kleisli composition**) in his lecture on **monads**.

```
f >=> g = λ a → let mb = f a
                in mb >>= g
```

See the next slide for more

@BartoszMilewski   Category Theory 10.1: Monads

Kleisli Composition (fish operator)	>=>	compose
Bind	>>=	flatMap
lifts **a** to **m a** (lifts A to F[A])	return	unit/pure

```haskell
class Monad m where
  (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
  return :: a -> m a
```

```haskell
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Kleisli Composition (fish operator)	>=>	compose
Bind	>>=	flatMap
lifts **a** to **m a** (lifts A to F[A])	return	unit/pure

```
            f             g
(>=>) :: (a → mb) → (b → mc) → (a → mc)

f >=> g = λa → let mb = f a
                in mb >>= g

(>>=) :: m a → (a → mb) → m b

ma >>= f = join (fmap f ma)
                      (a→mb)   ma
                         m(mb)

join :: m(ma) → ma
```

```
class Monad m where
  (>>=) :: ma → (a → mb) → mb
  return :: a → ma

class Functor m ⇒ Monad m where
  join :: m(ma) → ma
  return :: a → ma
```

>=> Kleisli Composition (aka the fish operator)
>>= Bind

```haskell
f >=> g = λa -> let mb = f a in mb >>= g
        = λa -> (f a) >>= g
```

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
class Functor m => Monad m where
  join :: m(m a) -> ma
  return :: a -> m a
```

Now let’s hand-code ourselves the **flatMap** functions of the **Option** and **List Monads**.

```scala
trait Monad[F[_]]:
  def unit[A](a: => A): F[A]
  extension [A](fa: F[A])
    def flatMap[B](f: A => F[B]): F[B]
```

```scala
extension [A,B,F[_]: Monad](f: A => F[B])
  def >=>[C](g: B => F[C]): A => F[C] =
    a => f(a).flatMap(g)
```

Before

>=> is generic and defined in terms of built-in flatMap

```scala
given Monad[Option] with
  def unit[A](a: => A): Option[A] = Some(a)
  extension [A](fa: Option[A])
    def flatMap[B](f: A => Option[B]): Option[B] = fa.flatMap(f)
```

```scala
given Monad[List] with
  def unit[A](a: => A): List[A] = List(a)
  extension [A](fa: List[A])
    def flatMap[B](f: A => List[B]): List[B] = fa.flatMap(f)
```

After

>=> is generic and defined in terms of hand-coded flatMap

```scala
given Monad[Option] with
  def unit[A](a: => A): Option[A] = Some(a)
  extension [A](fa: Option[A])
    def flatMap[B](f: A => Option[B]): Option[B] =
      fa match
        case Some(a) => f(a)
        case None => None
```

```scala
given Monad[List] with
  def unit[A](a: => A): List[A] = List(a)
  extension [A](fa: List[A])
    def flatMap[B](f: A => List[B]): List[B] =
      fa.foldRight(List.empty[B]):
        (a,bs) => f(a) ++ bs
```

Earlier we implemented a generic >=> in terms of the built-in flatMap function of the Option and List Monads.

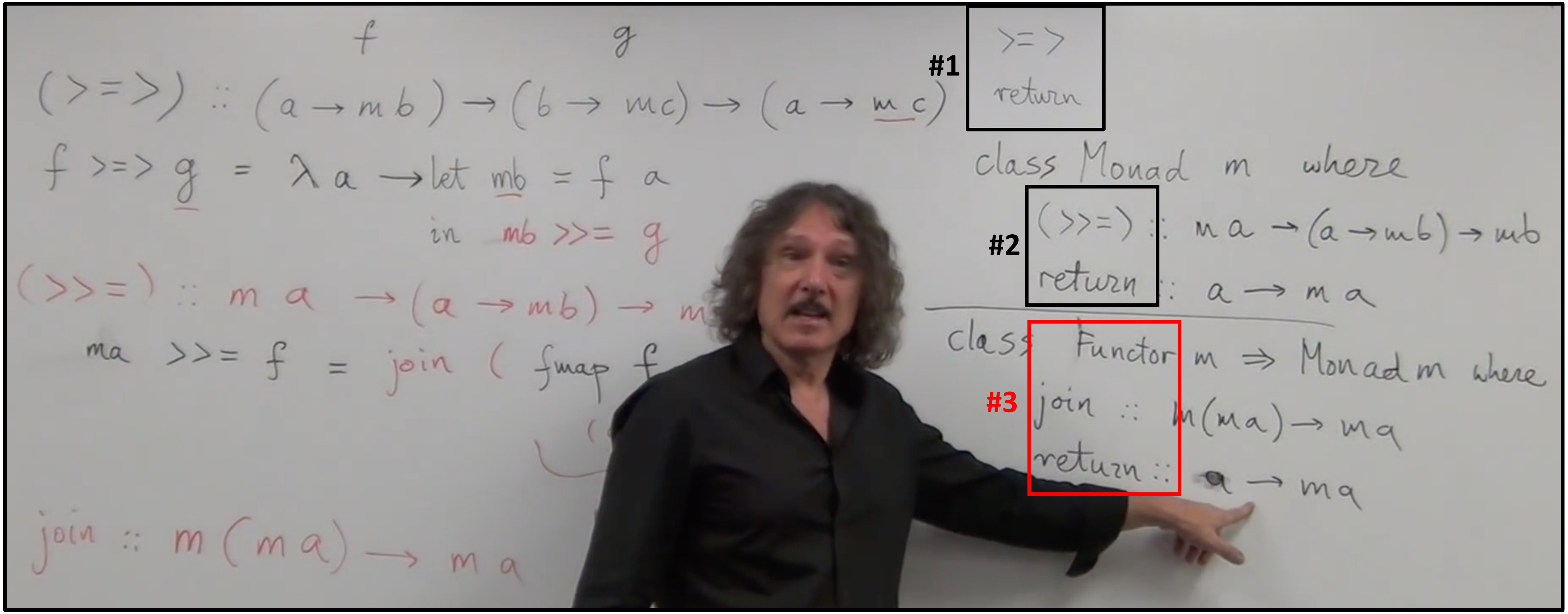
Let's do that again but this time implementing >=> in terms of the built-in map and join functions of the Option and List Monads.

@philip_schwarz

The next slide is just a refresher on the fact that it is possible to define a Monad in terms of unit, map and join, instead of in terms of unit and flatMap, or in terms of unit and the fish operator. If it is the first time that you go through this slide deck, you may want to skip the slide.

in Scala, join is called flatten

Bartosz Milewski introduces **a third definition of Monad** in terms of join and return, based on Functor



So this (**join** and **return**) is **an alternative definition of a monad**. But in this case I have to specifically say that m is a **Functor**, which is actually a nice thing, that I have to explicitly specify it.

…

But remember, in this case (**join** and **return**) you really have to assume that it is a **functor**. In this way, **join** is the most basic thing. Using just **join** and **return** is really more atomic than using either **bind** or the **Kleisli arrow**, because they additionally **susbsume functoriality**, whereas here, functoriality is separate, separately it is a functor and separately we define **join**, and separately we define **return**.

…

So this definition (**join** and **return**) or the definition with the **Kleisli arrow, they are not used in Haskell**, although they could have been. But **Haskell** people decided to use this (**>>=** and **return**) as their basic definition and then for every monad they separately define **join** and the **Kleisli arrow**. So if you have a monad you can use **join** and the **Kleisli arrow** because they are defined in the library for you. So it's always enough to define just **bind**, and then **fish** and **join** will be automatically defined for you, you don't have to do it.

Before

>=> is generic and defined in terms of built-in flatMap

```scala
trait Monad[F[_]]:
  def unit[A](a: => A): F[A]
  extension [A](fa: F[A])
    def flatMap[B](f: A => F[B]): F[B]

extension [A,B,F[_]: Monad](f: A => F[B])
  def >=>[C](g: B => F[C]): A => F[C] =
    a => f(a).flatMap(g)
```

```scala
given Monad[Option] with
  def unit[A](a: => A): Option[A] = Some(a)
  extension [A](fa: Option[A])
    def flatMap[B](f: A => Option[B]): Option[B] = fa.flatMap(f)
```

```scala
given Monad[List] with
  def unit[A](a: => A): List[A] = List(a)
  extension [A](fa: List[A])
    def flatMap[B](f: A => List[B]): List[B] = fa.flatMap(f)
```

After

>=> is generic and defined in terms of built-in map and join

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

trait Monad[F[_]] extends Functor[F]:
  def unit[A](a: => A): F[A]
  extension [A](ffa: F[F[A]])
    def join: F[A]

extension [A,B,F[_]: Monad](f: A => F[B])
  def >=>[C](g: B => F[C]): A => F[C] =
    a => f(a).map(g).join
```

```scala
given Monad[Option] with
  def unit[A](a: => A): Option[A] = Some(a)
  extension [A](fa: Option[A])
    def map[B](f: A => B): Option[B] = fa.map(f)
  extension [A](ffa: Option[Option[A]])
    def join: Option[A] = ffa.flatten
```

```scala
given Monad[List] with
  def unit[A](a: => A): List[A] = List(a)
  extension [A](fa: List[A])
    def map[B](f: A => B): List[B] = fa.map(f)
  extension [A](ffa: List[List[A]])
    def join: List[A] = ffa.flatten
```

Now let's hand-code ourselves the **map** and **join** functions of the **Option** and **List** **Monads**.

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

```scala
trait Monad[F[_]] extends Functor[F]:
  def unit[A](a: => A): F[A]
  extension [A](ffa: F[F[A]])
    def join: F[A]
```

```scala
extension [A,B,F[_]: Monad](f: A => F[B])
  def >=>[C](g: B => F[C]): A => F[C] =
    a => f(a).map(g).join
```

Before

>=> is generic and defined in terms of built-in map and join

```scala
given Monad[Option] with
  def unit[A](a: => A): Option[A] = Some(a)
  extension [A](fa: Option[A])
    def map[B](f: A => B): Option[B] = fa.map(f)
  extension [A](ffa: Option[Option[A]])
    def join: Option[A] = ffa.flatten
```

```scala
given Monad[List] with
  def unit[A](a: => A): List[A] = List(a)
  extension [A](fa: List[A])
    def map[B](f: A => B): List[B] = fa.map(f)
  extension [A](ffa: List[List[A]])
    def join: List[A] = ffa.flatten
```

After

>=> is generic and defined in terms of hand-coded map and join

```scala
given Monad[Option] with
  def unit[A](a: => A): Option[A] = Some(a)
  extension [A](fa: Option[A])
    def map[B](f: A => B): Option[B] = fa match
      case Some(a) => Some(f(a))
      case None => None
  extension [A](ffa: Option[Option[A]])
    def join: Option[A] = ffa match
      case Some(a) => a
      case None => None
```

```scala
given Monad[List] with
  def unit[A](a: => A): List[A] = List(a)
  extension [A](fa: List[A])
    def map[B](f: A => B): List[B] =
      fa.foldRight(List.empty[B])((a,bs) => f(a)::bs)
  extension [A](ffa: List[List[A]])
    def join: List[A] =
      ffa.foldRight(List.empty[A])((a,as) => a ++ as)
```

I would like to thank **Rob Norris** for his great talk (see next slide), from which I learned a lot about **Kleisli composition** and in which I first saw the use of a **syntax class** to add the **fish operator** to a type class.

```scala
// A typeclass that describes type constructors that allow composition with >=>
trait Fishy[F[_]] {

  // Our identity, A => F[A] for any type A
  def pure[A](a: A): F[A]

  // Composition - the "fish" operator
  def >=>[A, B, C](f: A => F[B], g: B => F[C]): A => F[C] =
    a => f(a).flatMap(g) // hey that looks like flatMap!

}
```

the **fish operator** (**Kleisli composition**), can be implemented using **flatMap**.

```scala
// A typeclass that describes type constructors that allow composition with >=>
trait Fishy[F[_]] {

  // Our identity, A => F[A] for any type A
  def pure[A](a: A): F[A]

  // The operation we need if we want to define >=>
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]

}
```

We can implement **compose**, the **fish operator** using **flatMap**, so the **fish operator** is something we can derive later really, the operation we need is **flatMap**.

Rob Norris @tpolecat

YouTube

scale.bythebay.io
Rob Norris
Functional Programming with Effects

```scala
// Now we can define >=> as an infix operator using a syntax class
implicit class FishyFunctionOps[F[_], A, B](f: A => F[B]) {
  def >=>[C](g: B => F[C])(implicit ev: Fishy[F]): A => F[C] =
    a => ev.flatMap(f(a))(g)
}


// Let's define an instance for Option
implicit val FishyOption: Fishy[Option] =
  new Fishy[Option] {
    def pure[A](a: A) = Some(a)
    def flatMap[A, B](fa: Option[A])(f: A => Option[B]) = fa.flatMap(f)
  }
```

We can define a **syntax class** that adds these methods so that anything that is an F[A], if there is a **Fishy** instance, gets these operations by syntax.

And this **Fishy** typeclass that we have derived from nothing, using math, is **Monad**. So this scary thing, it just comes naturally and I haven't seen people talk about getting to it from this direction. And so I hope that was helpful.

Fishy

```scala
// Fishy typeclass
trait Fishy[F[_]] {
  def pure[A](a: A): F[A]
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```

Monad

```scala
// Monad typeclass
trait Monad[F[_]] {
  def pure[A](a: A): F[A]
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}
```