Scala

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

Haskell

Let's start by reminding ourselves of a few aspects of Monads and Kleisli composition.

Philip Schwarz
@philip_schwarz

Defining a Monad in terms of Kleisli composition and Kleisli identity function

Kleisli composition + unit

```
trait Monad[F[_]] {
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C]
  def unit[A](a: => A): F[A]
}
```

Kleisli composition + return

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

Defining Kleisli composition in terms of flatMap (bind)

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

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

Defining a Monad in terms of flatmap (bind) and unit (return)

flatMap + unit

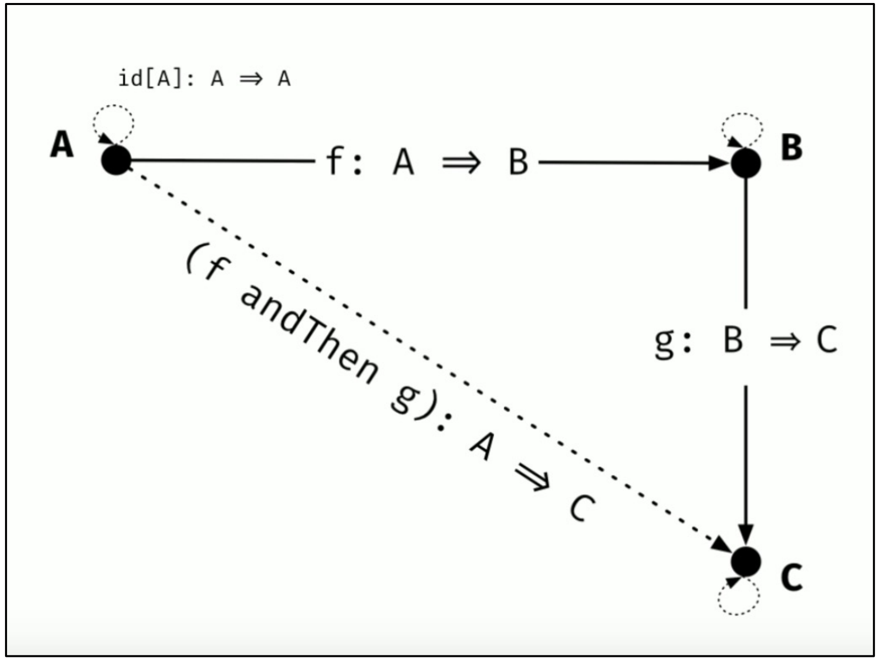
```
trait Monad[F[_]] {
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
  def unit[A](a: => A): F[A]

  // can then implement Kleisli composition using flatMap
  def compose[A,B,C](f: A => F[B], g: B => F[C]): A => F[C] =
    a => flatMap(f(a))(g)
}
```

bind + return (Kleisli composition can then be implemented with bind)

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

  -- can then implement Kleisli composition using bind
  (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
  (>=>) = \a -> (f a) >>= g
```

Function Composition

```scala
def andThen[A, B, C](f: A ⇒ B, g: B ⇒ C): A ⇒ C =
  a ⇒ g(f(a))
```



When we see an operation like this [function composition: **andThen**]…it is interesting to look for **algebraic properties** that operators like this have, and we might ask, for instance, is this an associative operation? So let's find out.

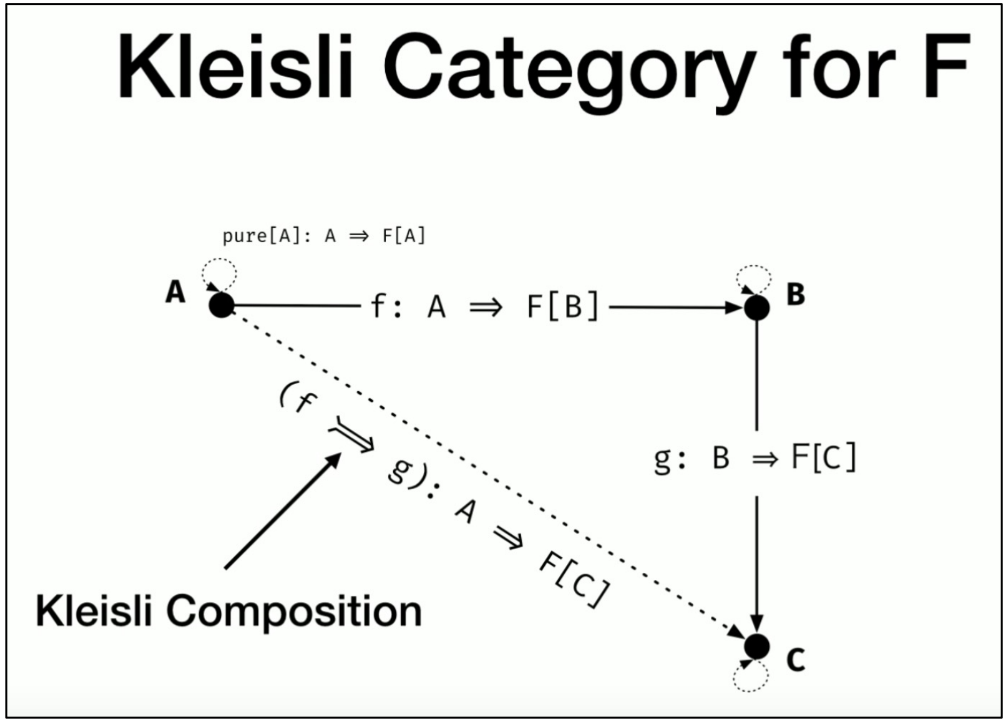
Rob Norris
@tpolecat

Rules (laws) for function composition

```scala
def andThen[A, B, C](f: A ⇒ B, g: B ⇒ C): A ⇒ C =
  a ⇒ g(f(a))

def id[A]: A ⇒ A =
  a ⇒ a

// right identity
f andThen id = f

// left identity
id andThen f = f

// associativity
(f andThen g) andThen h = f andThen (g andThen h)
```

So we have mappings between types, **we have an associative operator with an identity, at each type, and we proved it is true by the definition of function composition**, and because there is really only one way to define function composition, **this actually follows naturally from the type of function composition**, which I think is really interesting.

YouTube **scale.bythebay.io: Rob Norris, Functional Programming with Effects**



Rules (laws) for Kleisli composition

```
// left identity
pure ⋙ f ≡ f

// right identity
f ⋙ pure ≡ f

// associativity
f ⋙ (g ⋙ h) ≡ (f ⋙ g) ⋙ h
```

Rob Norris
@tpolecat

So what we want to do is figure out what this means in terms of flatMap.

Monad Rules (laws)

```
// left identity
pure(a).flatMap(f)           ≡ f(a)

// right identity
m.flatMap(pure)              ≡ m

// associativity
m.flatMap(g).flatMap(h)      ≡ m.flatMap(b ⇒ g(b).flatMap(h))
```

So we have **two operations**, **pure** and **flatMap**, **and laws telling us how they relate to each other**, and **this isn't something arbitrary**, that's what I am trying to get across. **These are things that come naturally from the category laws, just by analogy with pure function composition**.

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]
}

// Monad laws
pure(a).flatMap(f)          ≡ f(a)
m.flatMap(pure)             ≡ m
m.flatMap(g).flatMap(h)     ≡ m.flatMap(b ⇒ g(b).flatMap(h))
```

Everything you can say about monads is on this slide, but notice that **unlike the rules for function composition**, **which we proved were true and are necessarily true from the types, this is not the case for a monad, you can satisfy this [Monad] type and break the laws**, **so when we define instances we have to verify that they meet the laws**, and Cats and Scalaz both provide some machinery to make this very easy for you to do, **so if you define [monad] instances you have to check them [the laws]**.

Someone should do a conference talk on that, because it is really important and I have never seen a talk about it.

Function Composition

```scala
def andThen[A, B, C](f: A ⇒ B, g: B ⇒ C): A ⇒ C =
  a ⇒ g(f(a))

def id[A]: A ⇒ A =
  a ⇒ a

// right identity
f andThen id = f

// left identity
id andThen f = f

// associativity
(f andThen g) andThen h = f andThen (g andThen h)
```

Rob Norris

@tpolecat

Let's talk about Option Again

```scala
// Abbreviated Definition
sealed trait Option[+A]
case object None extends Option[Nothing]
case class  Some[+A](a: A) extends Option[A]

// Monad instance
implicit val OptionMonad: Monad[Option] =
  new Monad[Option] {
    def pure[A](a: A) = Some(a)
    def flatMap[A, B](fa: Option[A])(f: A ⇒ Option[B]) =
      fa match {
        case Some(a) ⇒ f(a)
        case None    ⇒ None
      }
  }
```

Let's talk about **Option** again. This is a monad instance for **Option**. I went ahead and wrote out how **flatMap** works here.

Notice, this [flatMap] method could return None all the time and it would type check, but it would break the right indentity law.

So this is why we check our laws when we implement typeclasses, it is very very important to do so.

Scala is not quite expressive enough to prove that stuff in the types, you have to do this with a second pass.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Monads

Principles of Reactive Programming

Martin Odersky

To qualify as a monad, a type has to satisfy three laws that connect **flatmap** and **unit**.

What is a Monad?

A monad `M` is a parametric type M[T] with two operations, `flatMap` and `unit`, that have to satisfy some laws.

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

In the literature, `flatMap` is more commonly called `bind`.

Monad Laws

To qualify as a monad, a type has to satisfy three laws:

Associativity:

```
m flatMap f flatMap g  ==  m flatMap (x => f(x) flatMap g)
```

Left unit

```
unit(x) flatMap f  ==  f(x)
```

Right unit

```
m flatMap unit  ==  m
```

Examples of Monads

- List is a monad with `unit(x) = List(x)`
- Set is monad with `unit(x) = Set(x)`
- Option is a monad with `unit(x) = Some(x)`
- Generator is a monad with `unit(x) = single(x)`

`flatMap` is an operation on each of these types, whereas `unit` in Scala is different for each monad.

```
// left identity
pure(a).flatMap(f)          ≡ f(a)

// right identity
m.flatMap(pure)             ≡ m

// associativity
m.flatMap(g).flatMap(h)     ≡ m.flatMap(b ⇒ g(b).flatMap(h))
```

Rob Norris
@tpolecat

Verifying that the Option Monad satisfies the Monad Laws

Checking Monad Laws

Let's check the monad laws for Option.

Here's flatMap for Option:

```
abstract class Option[+T] {

  def flatMap[U](f: T => Option[U]): Option[U] = this match {
    case Some(x) => f(x)
    case None => None
  }
}
```

Checking the Left Unit Law

Need to show: `Some(x) flatMap f == f(x)`

```
      Some(x) flatMap f

==    Some(x) match {
        case Some(x) => f(x)
        case None => None
      }

==    f(x)
```

Checking the Right Unit Law

Need to show: `opt flatMap Some == opt`

```
      opt flatMap Some

==    opt match {
        case Some(x) => Some(x)
        case None => None
      }

==    opt
```

Checking the Associative Law

Need to show: `opt flatMap f flatMap g == opt flatMap (x => f(x) flatMap g)`

```
      opt flatMap f flatMap g

==    opt match { case Some(x) => f(x) case None => None }
        match { case Some(y) => g(y) case None => None }

==    opt match {
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None =>
          None match { case Some(y) => g(y) case None => None }
      }
```

@odersky

Checking the Associative Law (2)

```
==    opt match {
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None
      }

==    opt match {
        case Some(x) => f(x) flatMap g
        case None => None
      }

==    opt flatMap (x => f(x) flatMap g)
```

Try

Another type: Try

In the later parts of this course we will need a type named Try.

Try resembles Option, but instead of Some/None there is a Success case with a value and a Failure case that contains an exception:

```
abstract class Try[+T]
case class Success[T](x: T)       extends Try[T]
case class Failure(ex: Exception) extends Try[Nothing]
```

Try is used to pass results of computations that can fail with an exception between threads and computers.

Creating a Try

You can wrap up an arbitrary computation in a Try.

```
Try(expr)     // gives Success(someValue) or Failure(someException)
```

Here's an implementation of Try:

```
object Try {
  def apply[T](expr: => T): Try[T] =
    try Success(expr)
    catch {
      case NonFatal(ex) => Failure(ex)
    }
```

NonFatal is a fairly technical thing, essentially, an exception is **fatal** if it does not make sense to export this beyond a single thread, there are a couple of exceptions that are, but the vast majority of them, both runtime exceptions and normal exceptions are **NonFatal**

It looks like **Try** might be a **Monad** with **unit = Try**

Composing Try

Just like with Option, Try-valued computations can be composed in for expresssions.

```
for {
  x <- computeX
  y <- computeY
} yield f(x, y)
```

If computeX and computeY succeed with results Success(x) and Success(y), this will return Success(f(x, y)).

If either computation fails with an exception ex, this will return Failure(ex).

@odersky

Definition of flatMap and map on Try

```
abstract class Try[T] {
  def flatMap[U](f: T => Try[U]): Try[U] = this match {
    case Success(x) => try f(x) catch { case NonFatal(ex) => Failure(ex) }
    case fail: Failure => fail
  }

  def map[U](f: T => U): Try[U] = this match {
    case Success(x) => Try(f(x))
    case fail: Failure => fail
  }}
```

So, for a Try value t,

```
t map f  ==  t flatMap (x => Try(f(x)))
         ==  t flatMap (f andThen Try)
```

Is Try a Monad?

Exercise

It looks like `Try` might be a monad, with `unit = Try`.

Is it?

```
O    Yes
O    No, the associative law fails
O    No, the left unit law fails
O    No, the right unit law fails
O    No, two or more monad laws fail.
```

@odersky

Is Try a Monad?

In fact it turns out that **the left unit law fails**.

Try in a sense **trades one monad law for another law** which in this context is more useful.

I call that other law the **bullet-proof principle**.

Monad Laws

To qualify as a monad, a type has to satisfy three laws:

Associativity:

```
m flatMap f flatMap g  ==  m flatMap (x => f(x) flatMap g)
```

Left unit

```
unit(x) flatMap f  ==  f(x)
```

Right unit

```
m flatMap unit  ==  m
```

Solution

It turns out the left unit law fails.

```
Try(expr) flatMap f  !=  f(expr)
```

Indeed the left-hand side will never raise a non-fatal exception whereas the right-hand side will raise any exception thrown by `expr` or `f`.

Hence, `Try` trades one monad law for another law which is more useful in this context:

> *An expression composed from 'Try', 'map', 'flatMap' will never throw a non-fatal exception.*

Call this the "bullet-proof" principle.

Exercise

It looks like Try might be a monad, with unit = Try.

Is it?

```
O    Yes
O    No, the associative law fails
O    No, the left unit law fails
O    No, the right unit law fails
O    No, two or more monad laws fail.
```