Arrive at Monads by going from composition of pure functions to composition of effectful functions

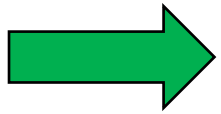
summary/overview of

Functional Programming Effects

by **Rob Norris**

```
f: A ⇒ B
g: B ⇒ C
f ; g: A ⇒ C
```

```
f ; g
```



```
f ⟹ g
```

```
f: A ⇒ F[B]
g: B ⇒ F[C]
f ⟹ g: A ⇒ F[C]
```

```
f ⟹ g ≡ λa.f(a) ≫= g
```

```
f ; g
```

```
; andThen
```

applies f before g

```
g ∘ f
```

```
∘ compose
```

applies g after f

```
⟹ Kleisli composition
```

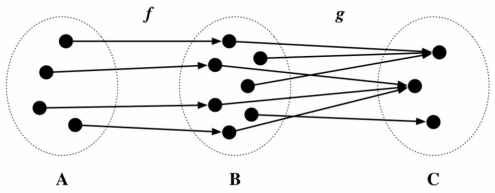
```
((A ⇒ F[B]) ⟹ (B ⇒ F[C])) ⇒ (A ⇒ F[C])
```

```
≫= flatMap
```

```
(F[A] ≫= (A ⇒ F[B])) ⇒ F[B]
```

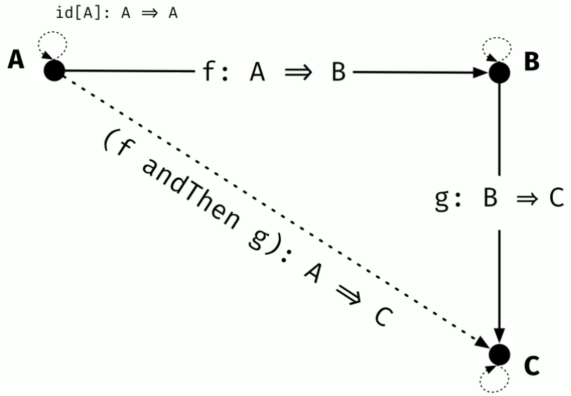
slides by  @philip_schwarz FP Illuminated http://fpilluminated.com/

Function Composition



Rob Norris @tpolecat

Function Composition



Category of Scala Types and Functions

- Our **objects** are types.
- Our **arrow** are pure functions.
- Our **associative composition** op is `andThen`.
- Our **identity arrows** at each object are `id[A]`.

Function Composition

```
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)
```

[In functional programming] all we have are values and pure functions. We have given up a lot of expressiveness to do this.

So what about …

- Partiality?
- Exceptions?
- Nondeterminism?
- Dependency injection?
- Logging?
- Mutable state?
- Imperative programming generally?

- Functions have to have an answer, but in Java sometimes you **return null.** How do we deal with that?
- We are in the world of expressions, and expressions don't throw **exceptions.**
- Functions have to have exactly **one answer**
- The power of FP comes from, it gives us the ability to reason locally about stuff, and if we have this sort of big **global scope** that is introducing stuff that any part of our program might depend on then that hinders our ability to do that.
- **Logging** is a side effect, right? The whole point of logging is to see when things are happening, and in FP we are dealing with expressions, we don't care when things happen: it doesn't matter. So what happens to logging [in FP]?
- **Mutable state**, obviously we don't have var any more.
- And **imperative programming** in general: where does it go?

It seems like a lot to give up, right? And a big barrier to learning FP is understanding what to do when you need one of these things and what FP principles do you apply to solve these problems that you run into all the time when you are writing programs. This is where **effects** come into play

Six Effects

Effect is a very vague term and that is ok because we are trying to talk about something that is outside the language. **Effect** and **side effect** are not the same thing. **Effects** are good. **Side effects** are **bugs**.

Their lexical similarity is really unfortunate because it leads to a lot of people conflating these ideas when they read about them and people using one instead of the other so it leds to a lot of confusion. So when you see **effect**, think a little bit about what is going on because it is a continual point of confusion.

So what I want to do is talk about **six of the effects**, they are kind of **the first ones you learn when you start doing FP**. They are kind of the first things in your toolbox.

There are many more and many ways to classify them but we are going to start small. We are going to talk about these **effects**… what they mean and … so hopefully by the end we'll have a pretty precise definition of what these things have in common.

Let's talk about Option

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

// Functons that may not yield an answer
val f: A ⇒ Option[B]
val g: B ⇒ Option[C]

// We can't compose them :-(
f andThen g // type error
```

Partiality

Partiality: we can define functions that might not return an answer.
If we could combine f and g the effects would have to be combined. If either of them failed, we couldn't possibly get a C out.
So if we had a way of smashing them together by doing function composition that would be pretty powerful but we can't.

Let's talk about List

```scala
// Abbreviated Definition
sealed trait List[+A]
case object Nil extends List[Nothing]
case class ::[A](head: A, tail: List[S]) extends List[A]

// Intuition: Functions that may yield many answers
val f: A ⇒ List[B]
val g: B ⇒ List[C]

// We can't compose them :-(
f andThen g // type error
```

Nondeterminism

In FP we take an interesting interpretation of List, we sometimes think about it as a kind of **nondeterminism**. We can define functions that might return any number of answers. And if we were to compose these two together end to end we would want to get every possible answer that we could have got from these two functions. But again: **we can't do that.**

Let's talk about Writer

Logging

```scala
// Abbreviated Definition
case class Writer[W, A](w: W, a: A)

// Intuition: Functions that annotate the values they compute.
val f: A ⇒ Writer[Info, B] // equivalent to A ⇒ (Info, B)
val g: B ⇒ Writer[Info, C]
```

```scala
// Example
type Log = List[String]
def toDouble(n: Int): Writer[Log, Double] =
  Writer(List(s"Converted $n to Double!"), n.toDouble)

toDouble(10) // Writer(List(Converted 10 to Double!),10.0)
```

This is another effect, that's a pair of some value W and an answer A. And the intuition here is functions that can annotate the values that they compute…[with some info] and the info might be a **log message**.
If we were able to compose these things together, what we would get are computations that can talk about what thery are doing and then if we had a way to smash those infos together we could make a big computation, run it, and get an answer and some kind of extra collected bits of information like a **log** for instance.

Six Effects

Rob Norris @tpolecat

So what about …

- Partiality?
- Exceptions?
- Nondeterminism?
- Dependency injection?
- Logging?
- Mutable state?
- Imperative programming generally?

Functional Programming with Effects
https://www.youtube.com/watch?v=po3wmq4S15A

You can have functions that might fail but also give you a reason why they failed.
This kind of gives us **exceptions** back.
We can't compose f and g. If you were able to compose them together, if you finally got a C out, you would know that both of the computations worked. If one of them failed you woud get a Left and you would get the first Left that was encountered because the computation would have to stop there because it wouldn't have a vaue to pass on. But again **that composition operator isn't defined**. But can you see the power that you would get by being able to do that, by chaning these things together?

Let's talk about Either

Exceptions

```scala
// Abbreviated Definition
sealed trait Either[+A, +B]
case class Left [+A, +B](a: A) extends Either[A, B]
case class Right[+A, +B](b: A) extends Either[A, B]

// Intuition: Functions that may fail with a reason.
val f: A ⇒ Either[String, B]
val g: B ⇒ Either[String, C]

// We can't compose them :-(
f andThen g // type error
```

We can compute values that have a **dependency**.
I can construct this computation p with a path and then I can run it with different hosts and I'll get a different answer back.
It's just sort of a currying thing at this point, a partial application thing, but if we were to compose these things we can have multiple computations that were dependent on that same configuration. We could compose them together and get a new computation and pass the configuration in and get our complete answer back. **We can't do that because we haven't yet defined function composition for that type**.

Let's talk about Reader

Dependency Injection

```scala
// Abbreviated Definition
case class Reader[A, B](run: A ⇒ B)

// Intuition: Functions with dependencies.
val f: A ⇒ Reader[Config, B] // equivalent to A ⇒ (Config ⇒ B)
val g: B ⇒ Reader[Config, C]
```

```scala
// Example
type Host = String
def path(s: String): Reader[Host, String] =
  Reader { host ⇒ s"http://$host/$s" }

val p = path("foo/bar")
p.run("google.com")     // http://google.com/foo/bar
p.run("tpolecat.org")   // http://tpolecat.org/foo/bar
```

A computation that takes some input state and computes a value and returns another state that might have been modified.
And **if we were able to compose these things together** then we would have our state threaded through our computation, which is nice, that kind of gives us **mutability** back, or a lot of the cases that mutabiity is used for.

Let's talk about State

Mutability

```scala
// Abbreviated Definition
case class State[S, A](run: S ⇒ (A, S))

// Intuition: Computations with a state transition.
val f: A ⇒ State[Info, B] // equivalent to A ⇒ (B ⇒ (B, Info))
val g: B ⇒ State[Info, C]
```

```scala
// Example
type Counter = Int
def greet(name: String): State[Counter, String] =
  State { count ⇒
    (s"Hello $name, you are person number $count", count + 1)
  }

val x = greet("Bob")
x.run(1)  // (Hello Bob, you are person number 1,2)
x.run(20) // (Hello Bob, you are person number 20,21)
```

Six Effects

Rob Norris @tpolecat

What do they have in common?

- All compute an "answer"but also encapsulate something extra about the computation.
- This is what we call an effect. But it's very vague. Can we be more precise about what they have in common?

But they don't compose!

That's our problem. So what can we do?
What would it take to make them compose?

All have shape F[A]

```
type F[A] = Option[A]
type F[A] = Either[E, A] // for any type E
type F[A] = List[A]
type F[A] = Reader[E, A] // for any type E
type F[A] = Writer[W, A] // for any type W
type F[A] = State[S, A]  // for any type S
```

An effect is whatever distinguishes `F[A]` from `A`.

Because effectful value takes too long to say, we sometimes call them programs

All have shape F[A]

The Effect

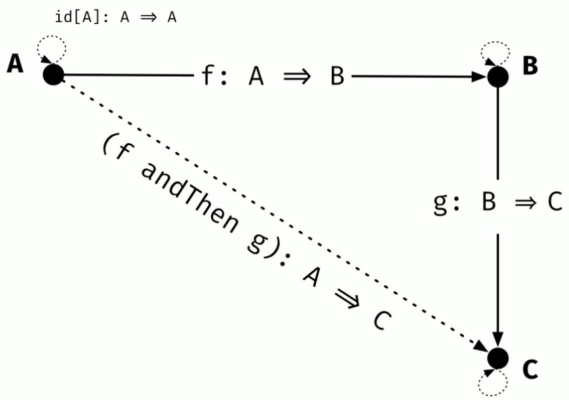
F[A]

"This is a program in F that computes a value of type A."

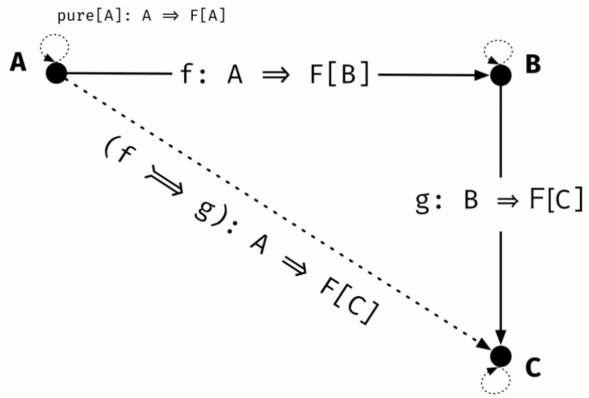
Here is our function diagram for pure function composition. And if we sort of replace things with **effectful** functions, they look like this, so we have something like **andThen**, looks something like a **fish**, and every type has an id, we are calling it **pure**. **<u>If we were able to define this and make it compose then we would get that power that we were talking about</u>.** So how do we write this in Scala?

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**.

What would it take?



What would it take?



The Operations

```
// 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]

}
```

The Operations

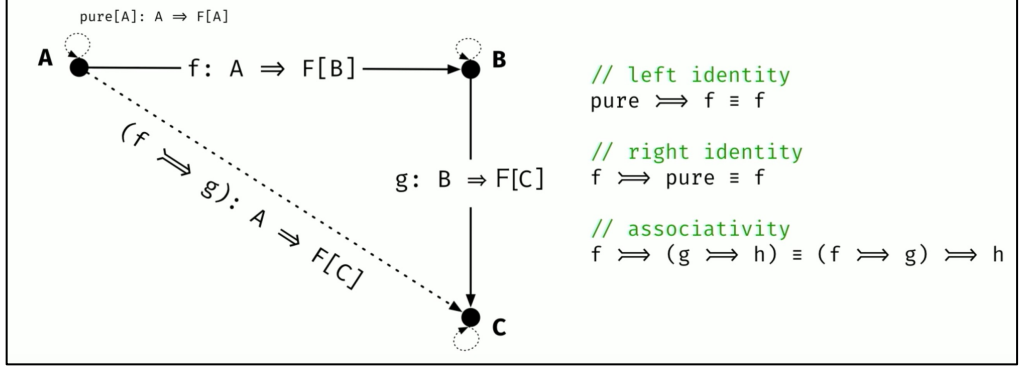
```
// Now we can define ⟹ as an infix operator using a syntax class
implicit class FishyFunctionOps[F[_], A, B](f: A ⇒ F[B]) {
  def ⟹[C](g: B ⇒ F[C])(implicit ev: Fishy[F]): A ⇒ F[C] =
    a ⇒ ev.flatMap(f(a))(g)
}
```

So we implemented the **fish operator** and we can do this kind of **composition**, but what we have forgotten about are all the rules for the category.

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

So I am going to tell you what we have been doing: this is called the **Kleisli category**.

The Rules




```
// left identity
pure ⟹ f ≡ f

// right identity
f ⟹ pure ≡ f

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

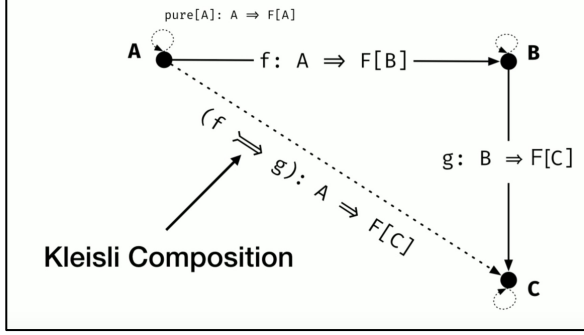
The Rules

```
// 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))
```

Kleisli Category for F



Rob Norris @tpolecat

We also derived all the **laws**, 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 **Monad**. You can satisfy this type and break the **laws**. So when we define instances we have to verify that they satisfy the **laws**, and **Cats** and **Scalaz** both provide some machinery to make this very easy for you to do. So if you define instances, you have to check them.

And this **Fishy** typeclass that ve 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.

So we have **pure** and **flatMap** that are abstract but we can define some familiar operations in terms of them, e.g. **map** and **tuple**.

Fishy

```
// 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

```
// 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

```
// 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]

  def map[A, B](fa: F[A])(f: A ⇒ B): F[B] =
    flatMap(fa)(a ⇒ pure(f(a)))

  def tuple[A, B](fa: F[A], fb: F[B]): F[(A, B)] =
    flatMap(fa)(a ⇒ map(fb)(b ⇒ (a, b)))

}
```

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

Monad

```
// Monad syntax
implicit class MonadOps[F[_], A](fa: F[A])(implicit ev: Monad[F]) {

  // Delegate to `ev`
  def flatMap[B](f: A ⇒ F[B]): F[B]    = ev.flatMap(fa)(f)
  def map[B](f: A ⇒ B): F[B] = ev.map(fa)(f)
  def tuple[B](fb: F[B]): F[(A, B)]    = ev.tuple(fa, fb)

}
```

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
      }
  }
```

Partiality

Let's talk about Either Again

```scala
// Abbreviated Definition
sealed trait Either[+A, +B]
case class Left [+A, +B](a: A) extends Either[A, B]
case class Right[+A, +B](b: A) extends Either[A, B]

// Monad instance
implicit def eitherMonad[L]: Monad[Either[L, ?]] =
  new Monad[Either[L, ?]] {
    def pure[A](a: A) = Right(a)
    def flatMap[A, B](fa: Either[L, A])(f: A ⇒ Either[L, B]) =
      fa match {
        case Left(l)  ⇒ Left(l)
        case Right(a) ⇒ f(a)
      }
  }
```

Exceptions

Let's talk about List Again

```scala
// Abbreviated Definition
sealed trait List[+A]
case object Nil extends List[Nothing]
case class ::[A](head: A, tail: List[S]) extends List[A]

// Monad instance
implicit val ListMonad: Monad[List] =
  new Monad[List] {
    def pure[A](a: A) = a :: Nil
    def flatMap[A, B](fa: List[A])(f: A ⇒ List[B]): List[B] =
      fa.foldRight(List.empty[B])((a, bs) ⇒ f(a) ::: bs)
  }
```

Nondeterminism

Let's talk about Reader Again

```scala
// Abbreviated Definition
case class Reader[A, B](run: A ⇒ B)

// Monad instance
implicit def readerMonad[E]: Monad[Reader[E, ?]] =
  new Monad[Reader[E, ?]] {
    def pure[A](a: A) = Reader(e ⇒ a)
    def flatMap[A, B](fa: Reader[E, A])(f: A ⇒ Reader[E, B]) =
      Reader { e ⇒
        val a = fa.run(e)
        f(a).run(e)
      }
  }
```

Dependency Injection

Rob Norris @tpolecat

Our effects, again.

Let's talk about Writer Again

Let's not, because I have to introduce **Monoids** to do that. The gist of it is you can get **logging** back

Logging

Let's talk about Either Again

```scala
def validate(tag: String, value: String) =
  if (value.nonEmpty) Right(value) else Left(s"$tag is empty")

def validateName(first: String, last: String) =
  for {
    first ← validate("First name", first)
    last  ← validate("Last name", last)
  } yield s"$first $last"
```

```
scala> validateName("Bob", "Dole")
res24: Either[String,String] = Right(Bob Dole)

scala> validateName("Bob", "")
res25: Either[String,String] = Left(Last name is empty)
```

Let's talk about Reader Again

```scala
type Host = String
def path(s: String): Reader[Host, String] =
  Reader(host ⇒ s"http://$host/$s")

val hostLen: Reader[Host, Int] =
  Reader(host ⇒ host.length)

val prog = for {
  a ← path("foo/bar")
  b ← hostLen
} yield s"Path is $a and len is $b."
```

```
scala> prog.run("google.com")
res70: String = Path is http://google.com/foo/bar and len is 10.
```

Let's talk about State Again

```scala
// Abbreviated Definition
case class State[S, A](run: S ⇒ (A, S))

// Monad instance
implicit def monadState[S]: Monad[State[S, ?]] =
  new Monad[State[S, ?]] {
    def pure[A](a: A) = State(s ⇒ (a, s))
    def flatMap[A, B](fa: State[S, A])(f: A ⇒ State[S, B]) =
      State { s ⇒
        val (a, s') = fa.run(s)
        f(a).run(s')
      }
  }
```

Mutability

Let's talk about State Again

```scala
type Seed = Int
val rnd: State[Seed, Int] =
  State { s ⇒
    val next = ((s.toLong * 16807) % Int.MaxValue).toInt
    (next, next)
  }

val d6 = rnd.map(_ % 6)

// 2d6+2
val damage =
  for {
    a ← d6
    b ← d6
  } yield a + b + 2
```