Functional Programming with Effects

Rob Norris • SBTB 2017

Hello

- I'm Rob, I do functional programming.
- I'm **@tpolecat** pretty much everywhere, I'm easy to find.
- I work on a bunch of open-source FP libraries for Scala.
- I write software for the **Gemini Observatory**.

Rant

Motivation

- Five years ago everyone was talking about monads.
- Three years ago everyone was talking about free monads.
- Last year everyone was talking about fixpoints and recursion schemes.

Motivation

- But what about the people who just got here?
- Monads have not gotten any easier.
- Organizations are trying to use this stuff and are struggling.

Motivation

Adelbert Chang
Computer Programmer

Rob Norris
(File Photo)

Goals for Today

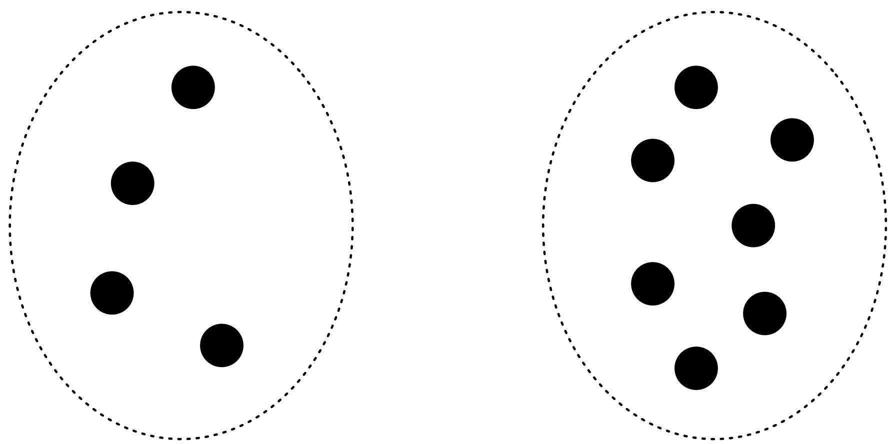
- Do not panic.
- Understand what FP is, and why it's useful.
- Gain some insight into the way functional programmers think about things.
- Understand why monads are useful, where they come from, and where they fit in the big scheme of things.
- Be inspired to be curious.

Functional Programming

Functional Programming

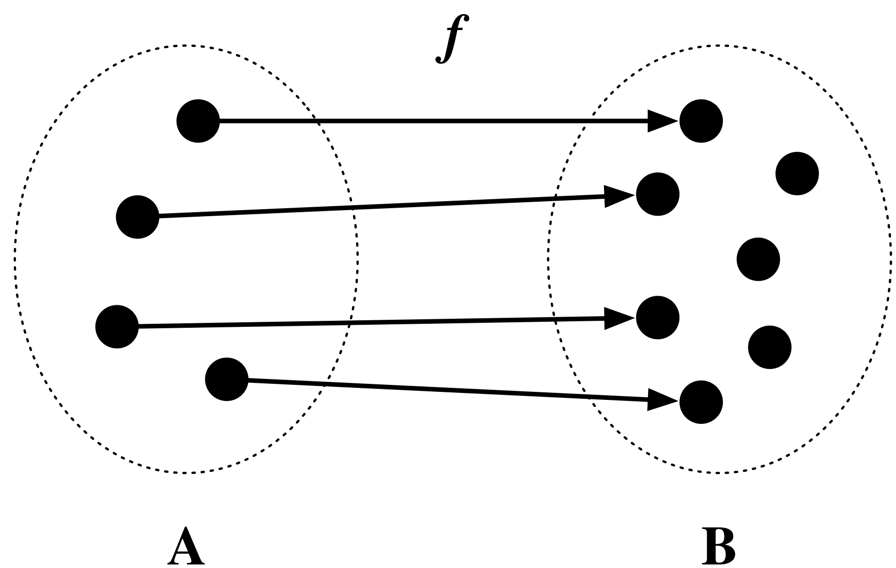
Functional Programming

Functional Programming



A B

Functional Programming



Pure Functions

- Such functions are said to be **pure**.
- Output is determined entirely by the input.
- Consequence of programming with pure functions:
    - Evaluating an expression always results in the same answer.
    - We can always **inline** a function, or **factor** one out.
    - We can always substitute a variable for the expression it's bound to, or introduce a new variable to factor out common sub-expressions. This property of expressions is called r**eferential transparency.**

Referential Transparency

Are these programs the same?

```
// program 1
val a = <expr>
(a, a)
```

```
// program 2
(<expr>, <expr>)
```

Referential Transparency

Are these programs the same?

```scala
// program 1
val a = 42
(a, a)
```

```scala
// program 2
(42, 42)
```

Referential Transparency

Are these programs the same?

```
// program 1
val a = iter.next() // an iterator
(a, a)
```

```
// program 2
(iter.next(), iter.next())
```

Referential Transparency

Are these programs the same?

```
// program 1
val a = println("hi")
(a, a)
```

```
// program 2
(println("hi"), println("hi"))
```

Referential Transparency

Are these programs the same?

```scala
// program 1
val a = Array(1, 2, 3)
(a, a)
```

```scala
// program 2
(Array(1, 2, 3), Array(1, 2, 3))
```

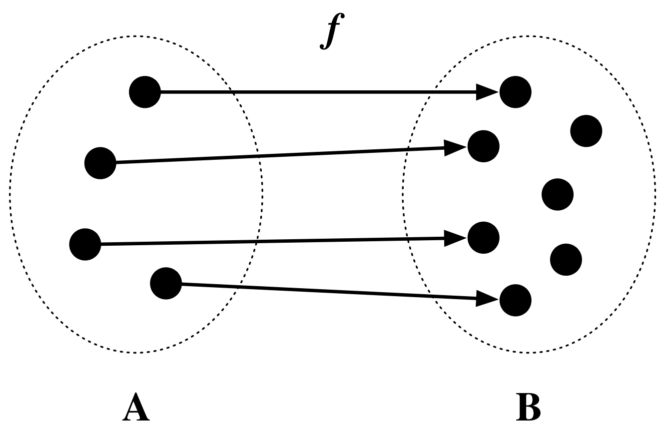
Referential Transparency

- Every expression is is either **referentially transparent**, or …
- … it's a **side-effect**. It's one or the other.
- This is a **syntactic property of programs**.
- We can use substitutions to perform **equational reasoning**.

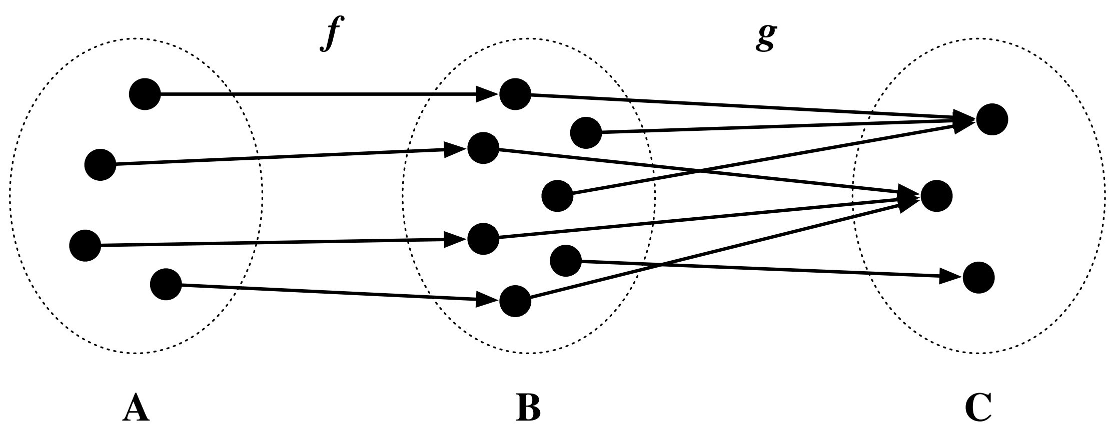
World of Expressions

- Functional programs are **expressions.**
- Running a functional program means we're **evaluating** an expression.
- We build bigger programs out of smaller ones by **composing** them.

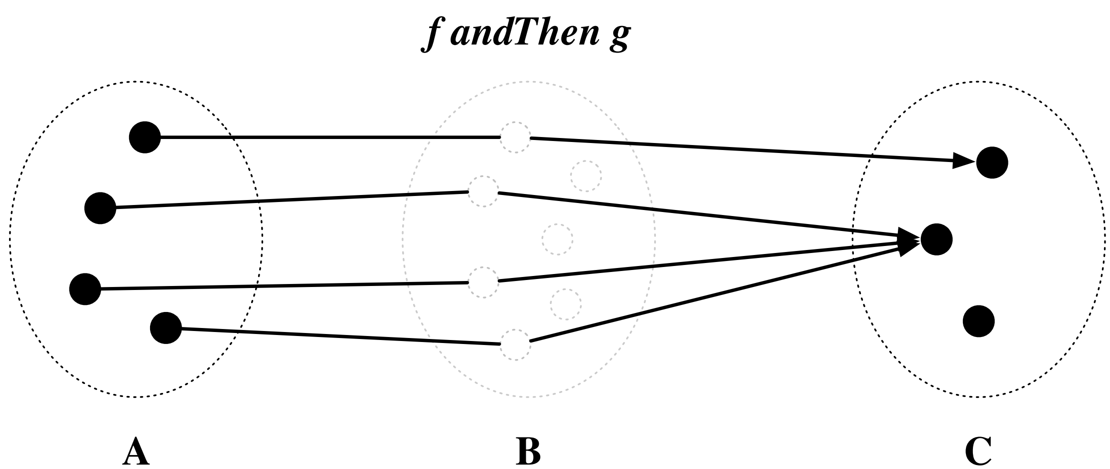
Function Composition



Function Composition



Function Composition



Function Composition

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

Function Composition

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

// right association
f andThen (g andThen h)
```

Function Composition

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

// right association
a ⇒ (g andThen h)(f(a))
```

Function Composition

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

// right association
a ⇒ (b ⇒ h(g(b)))(f(a))
```

Function Composition

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

// right association
a ⇒ h(g(f(a)))
```

Function Composition

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

// right association
a ⇒ h(g(f(a)))

// left association
(f andThen g) andThen h
```

Function Composition

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

// right association
a ⇒ h(g(f(a)))

// left association
a ⇒ h((f andThen g)(a))
```

Function Composition

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

// right association
a ⇒ h(g(f(a)))

// left association
a ⇒ h((b ⇒ g(f(b)))(a))
```

Function Composition

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

// right association
a ⇒ h(g(f(a)))

// left association
a ⇒ h(g(f(a)))
```

Function Composition

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

// a new legal substitution
(f andThen g) andThen h = f andThen (g andThen h)
```

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
```

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
```

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
a ⇒ id(f(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
a ⇒ (b ⇒ b)(f(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
a ⇒ f(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
```

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
```

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
```

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
a ⇒ f(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
a ⇒ f((b ⇒ b)(a))
```

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
a ⇒ f(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
f
```

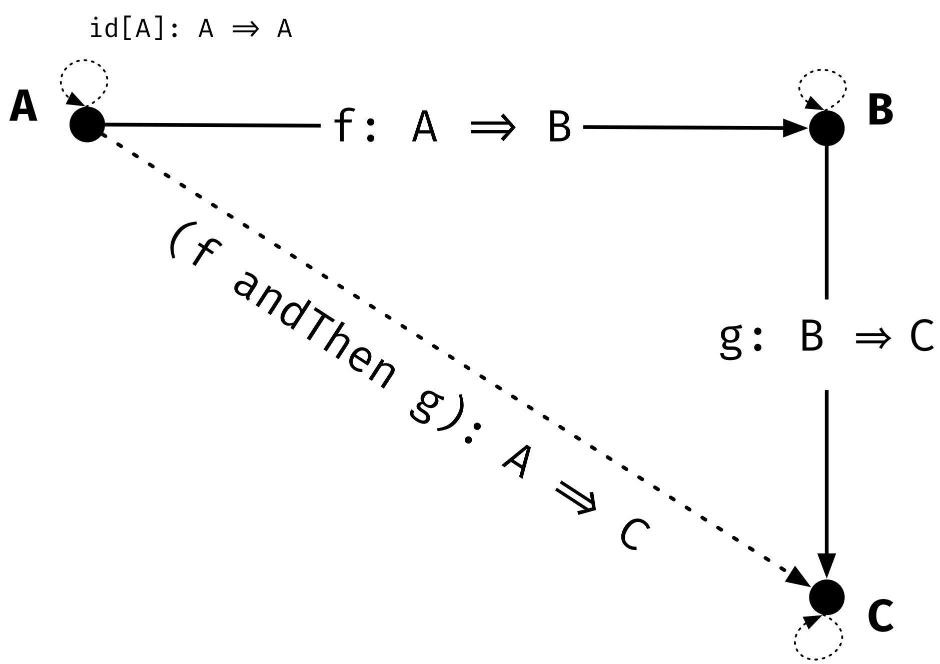
Function Composition

```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
```

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)
```

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]`.

So what about …

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

Six Effects

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
```

Let's talk about Either

```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
```

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
```

Jamie Allen
@jamie_allen

@andygscott Were you showing people the Reader monad before you left?

3:31 PM - Nov 15, 2017

1

Andy Scott ☜ @andygscott 1h
Replying to @jamie_allen
Yep

Jamie Allen
@jamie_allen

Okay, who do I need to deprogram from that malarkey? :p

3:33 PM - Nov 15, 2017

1

Let's talk about Reader

```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]
```

Let's talk about Reader

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

// 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
```

Let's talk about Reader

```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]

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

Let's talk about Writer

```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]
```

Let's talk about Writer

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

// 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)
```

Let's talk about Writer

```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]

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

Let's talk about State

```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]
```

Let's talk about State

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

// 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)
```

Let's talk about State

```
// 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]

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

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?

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` *.*

All have shape F[A]

The Effect

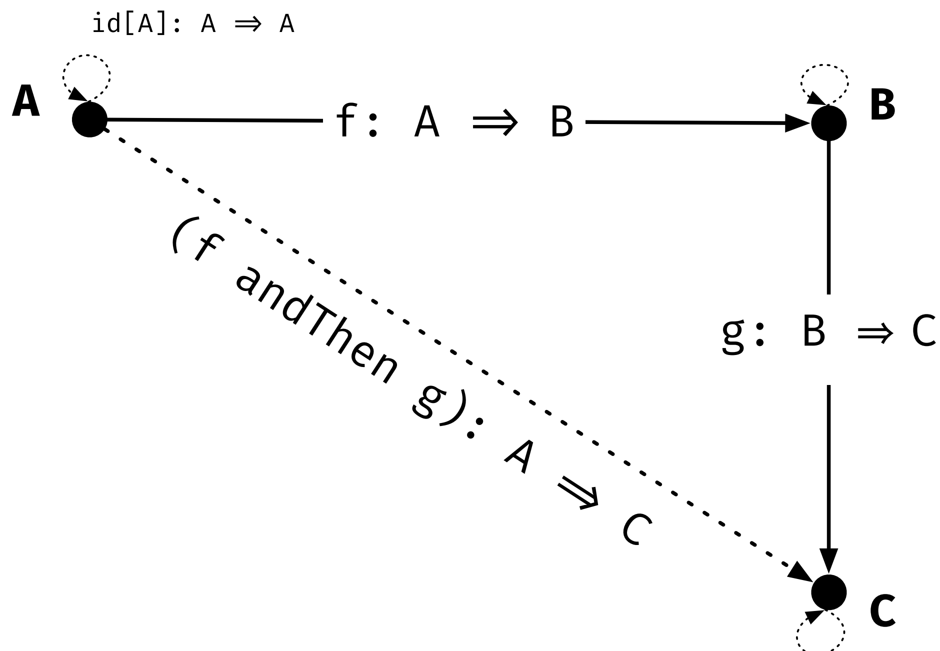
F[A]

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

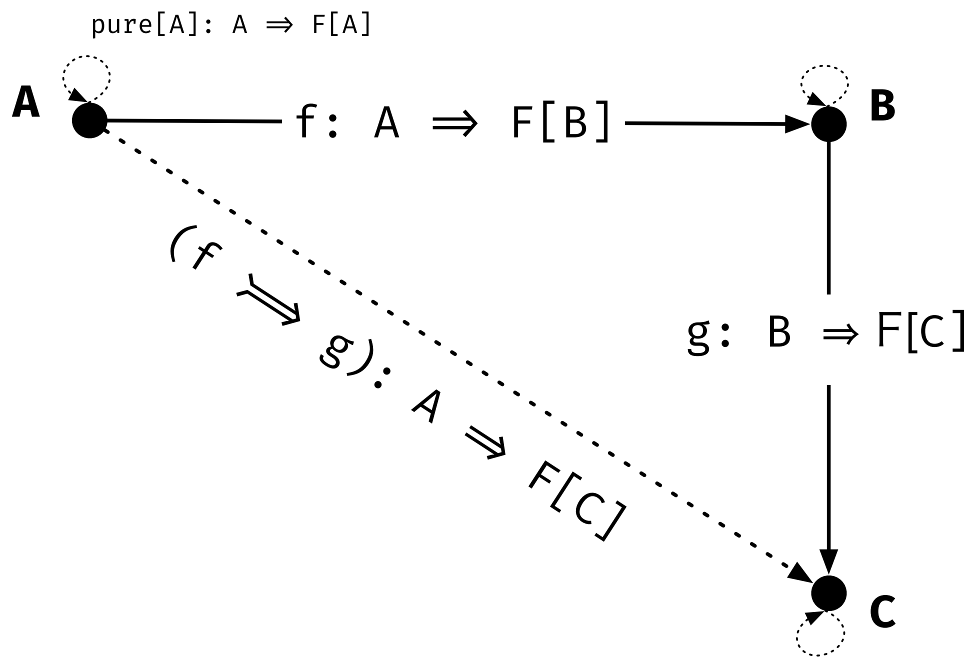
But they don't compose!

```
scala> val char10: String ⇒ Option[Char] =
     |   s ⇒ s.lift(10)
char10: String ⇒ Option[Char] = $$Lambda$5661/390122011@37974d1f

scala> val letter: Char ⇒ Option[Int] =
     |   c ⇒ if (c.isLetter) Some(c.toInt) else None
letter: Char ⇒ Option[Int] = $$Lambda$5662/973361211@77d94464

scala> char10 andThen letter
<console>:16: error: type mismatch;
 found   : Char ⇒ Option[Int]
 required: Option[Char] ⇒ ?
       char10 andThen letter
                      ^
```

What would it take?



What would it take?



The Operations

```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]

}
```

The Operations

```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) // we have an F[B] and a B => F[C] and we're stuck

}
```

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]

  // 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 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)
}
```

The Operations

```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)
  }
```

The Operations

```
scala> val char10: String ⇒ Option[Char] =
     |   s ⇒ s.lift(10)
char10: String ⇒ Option[Char] = $$Lambda$5661/390122011@37974d1f

scala> val letter: Char ⇒ Option[Int] =
     |   c ⇒ if (c.isLetter) Some(c.toInt) else None
letter: Char ⇒ Option[Int] = $$Lambda$5662/973361211@77d94464

scala> char10 andThen letter
<console>:16: error: type mismatch;
 found   : Char ⇒ Option[Int]
 required: Option[Char] ⇒ ?
       char10 andThen letter
                      ^
```

The Operations

```
scala> val char10: String ⇒ Option[Char] =
     |   s ⇒ s.lift(10)
char10: String ⇒ Option[Char] = $$Lambda$5661/390122011@37974d1f

scala> val letter: Char ⇒ Option[Int] =
     |   c ⇒ if (c.isLetter) Some(c.toInt) else None
letter: Char ⇒ Option[Int] = $$Lambda$5662/973361211@77d94464

scala> char10 ⟾ letter
res5: String ⇒ Option[Int] = FishyFunctionOps$$Lambda$5664/537915194@1c9443ec
```

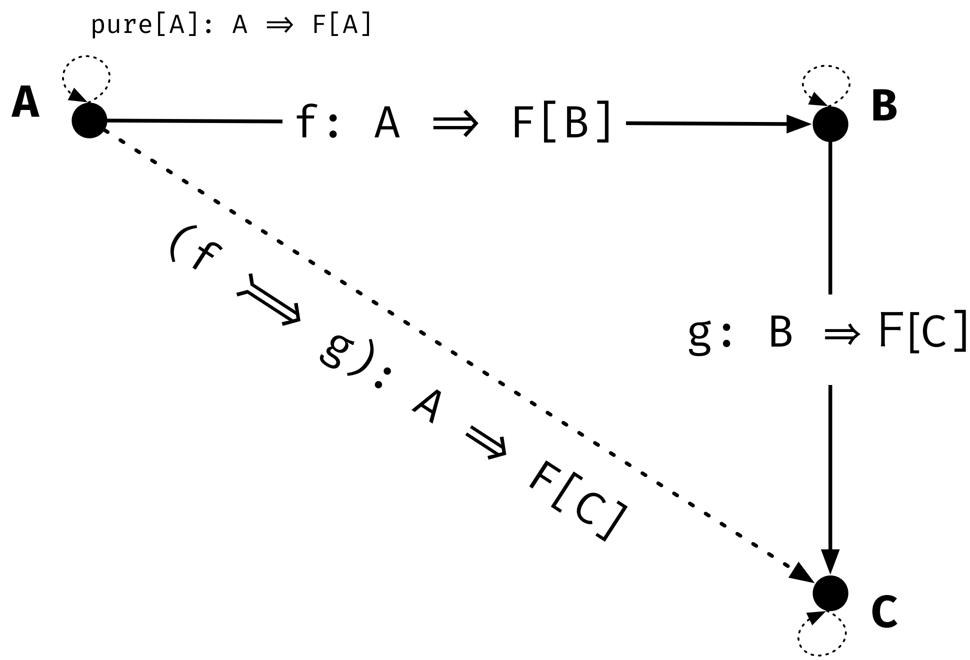
The Operations

```
scala> char10 >=> letter
res5: String => Option[Int] = FishyFunctionOps$$Lambda$5664/537915194@1c9443ec
```

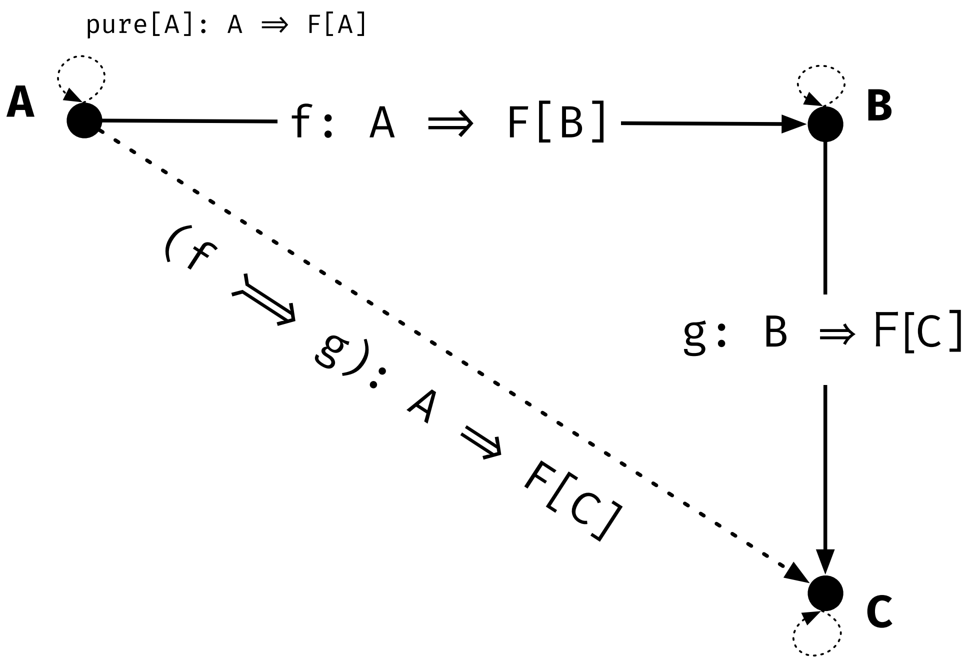
The Operations

```
scala> char10 ⟾ letter
res5: String ⇒ Option[Int] = FishyFunctionOps$$Lambda$5664/537915194@1c9443ec

scala> res5("foo")
res6: Option[Int] = None

scala> res5("foobarbazqux")
res7: Option[Int] = Some(117)

scala> res5("foobarbazq9x")
res8: Option[Int] = None
```

The Rules



The Rules




```
// left identity
pure >=> f ≡ f

// right identity
f >=> pure ≡ f

// associativity
f >=> (g >=> h) ≡ (f >=> g) >=> h
```

The Rules

```
// left identity
pure >=> f              ≡ f
```

The Rules

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

The Rules

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

The Rules

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

The Rules

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

// right identity
f >=> pure                ≡ f
```

The Rules

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

// right identity
a ⇒ f(a).flatMap(pure)   ≡ f
```

The Rules

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

// right identity
a ⇒ f(a).flatMap(pure)   ≡ a ⇒ f(a)
```

The Rules

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

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

The Rules

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

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

The Rules

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

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

// left-associative composition
(f >=> g) >=> h
```

The Rules

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

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

// left-associative composition
a ⇒ (f ⟾ g)(a).flatMap(h)
```

The Rules

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

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

// left-associative composition
a ⇒ (b ⇒ f(b).flatMap(g))(a).flatMap(h)
```

The Rules

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

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

// left-associative composition
a ⇒ f(a).flatMap(g).flatMap(h)
```

The Rules

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

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

// left-associative composition
a ⇒ f(a).flatMap(g).flatMap(h)
≡
// right-associative composition
f >=> (g >=> h)
```

The Rules

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

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

// left-associative composition
a => f(a).flatMap(g).flatMap(h)
≡
// right-associative composition
a => f(a).flatMap(g >=> h)
```

The Rules

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

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

// left-associative composition
a ⇒ f(a).flatMap(g).flatMap(h)
≡
// right-associative composition
a ⇒ f(a).flatMap(b ⇒ g(b).flatMap(h))
```

The Rules

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

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

// left-associative composition
f(a).flatMap(g).flatMap(h)
≡
// right-associative composition
f(a).flatMap(b ⇒ g(b).flatMap(h))
```

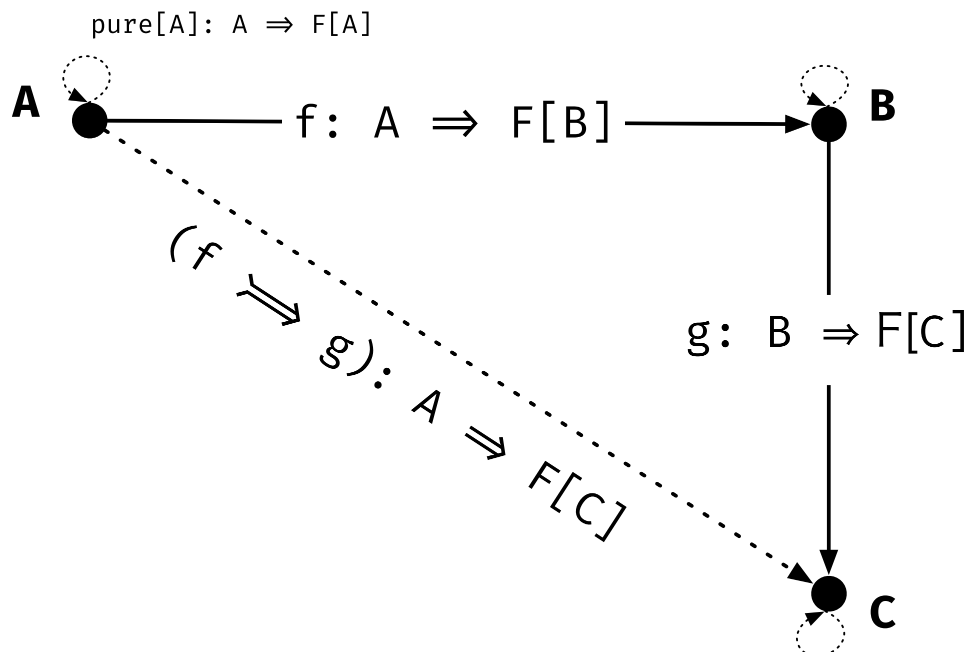
The Rules

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

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

// left-associative composition
m.flatMap(g).flatMap(h)
                                   ≡
// right-associative composition
m.flatMap(b ⇒ g(b).flatMap(h))
```

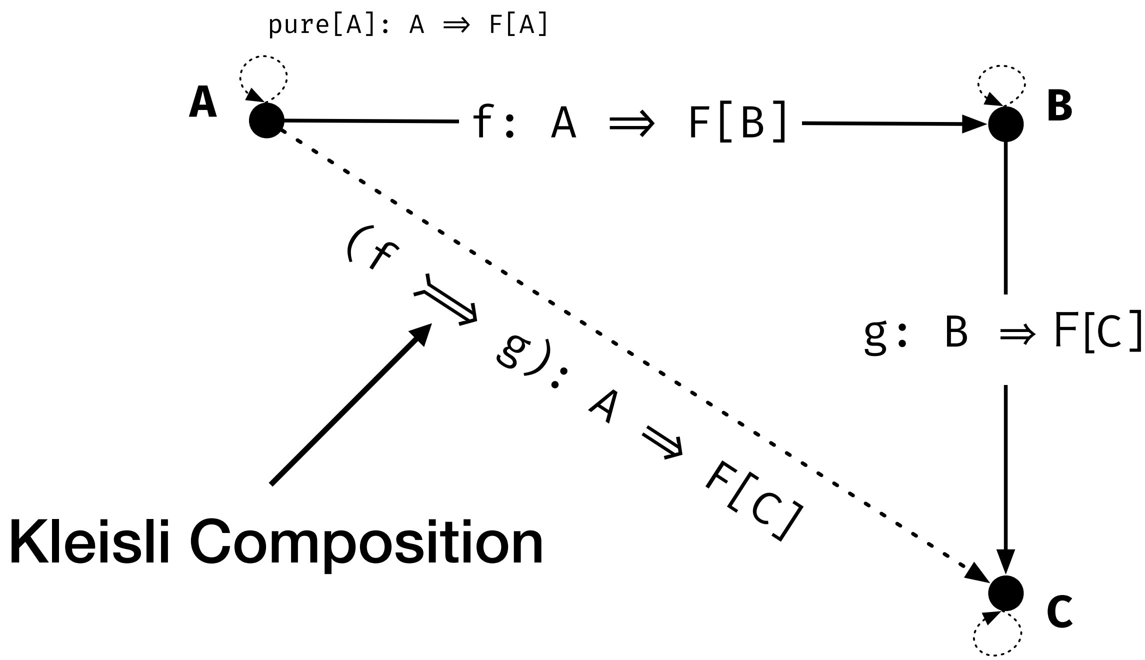
The Rules

```
// 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



Kleisli Category for F



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]
}
```

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

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

```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

```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]

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

}
```

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]

  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)))

}
```

Monad

```scala
// 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)

}
```

Monad

```scala
// 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)

  // Derived syntax
  def <*[B](fb: F[B]): F[A] = ev.map(tuple(fb))(_._1)
  def *>[B](fb: F[B]): F[B] = ev.map(tuple(fb))(_._2)

}
```

Our effects, again.

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]
```

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

```scala
def validate(s: String) = if (s.nonEmpty) Some(s) else None

scala> validate("Bob") tuple validate("Dole")
res13: Option[(String, String)] = Some((Bob,Dole))

scala> validate("") tuple validate("Dole")
res14: Option[(Nothing, String)] = None

scala> validate("Bob") *> validate("Dole")
res15: Option[String] = Some(Dole)

scala> validate("") *> validate("Dole")
res16: Option[String] = None

scala> validate("Dole") <* validate("Bob")
res17: Option[String] = Some(Dole)
```

Let's talk about Either Again

```
// 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]
```

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)
      }
  }
```

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 List Again

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

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)
  }
```

Let's talk about List Again

```
scala> List(1,2,3) tuple List('x', 'y')
res28: List[(Int, Char)] = List((1,x), (1,y), (2,x), (2,y), (3,x), (3,y))

scala> List(1,2,3) *> List('x', 'y')
res29: List[Char] = List(x, y, x, y, x, y)

scala> List('x', 'y') <* List(1,2,3)
res30: List[Char] = List(x, x, x, y, y, y)
```

Let's talk about Reader Again

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

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)
      }
  }
```

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 Writer Again

(let's not)

Let's talk about State Again

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

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′)
      }
  }
```

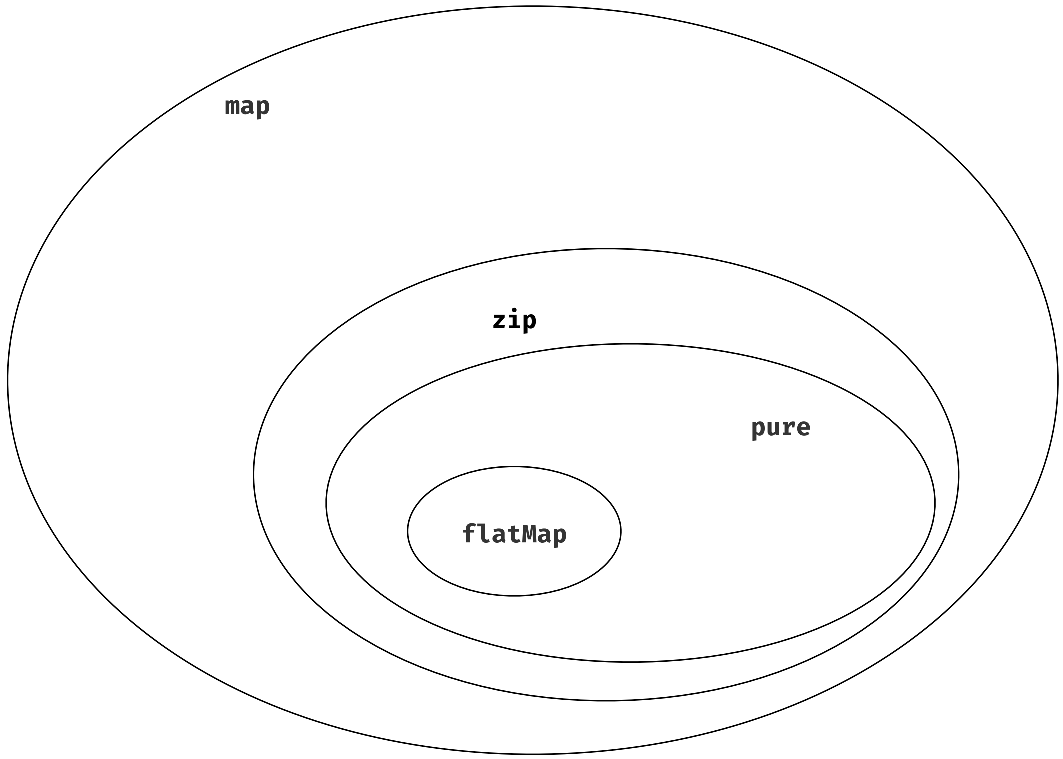
Let's talk about State Again

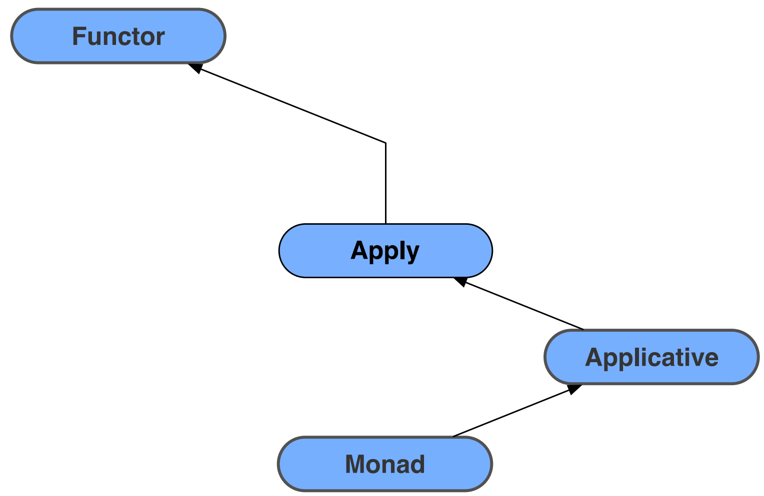
```scala
type Seed = Int
val rnd: State[Seed, Int] =
  State { s ⇒
    val next = ((s.toLong * 16807) % Int.MaxValue).toInt
    (next, next)
  }

val d6 = rnd.map(_ % 6)

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

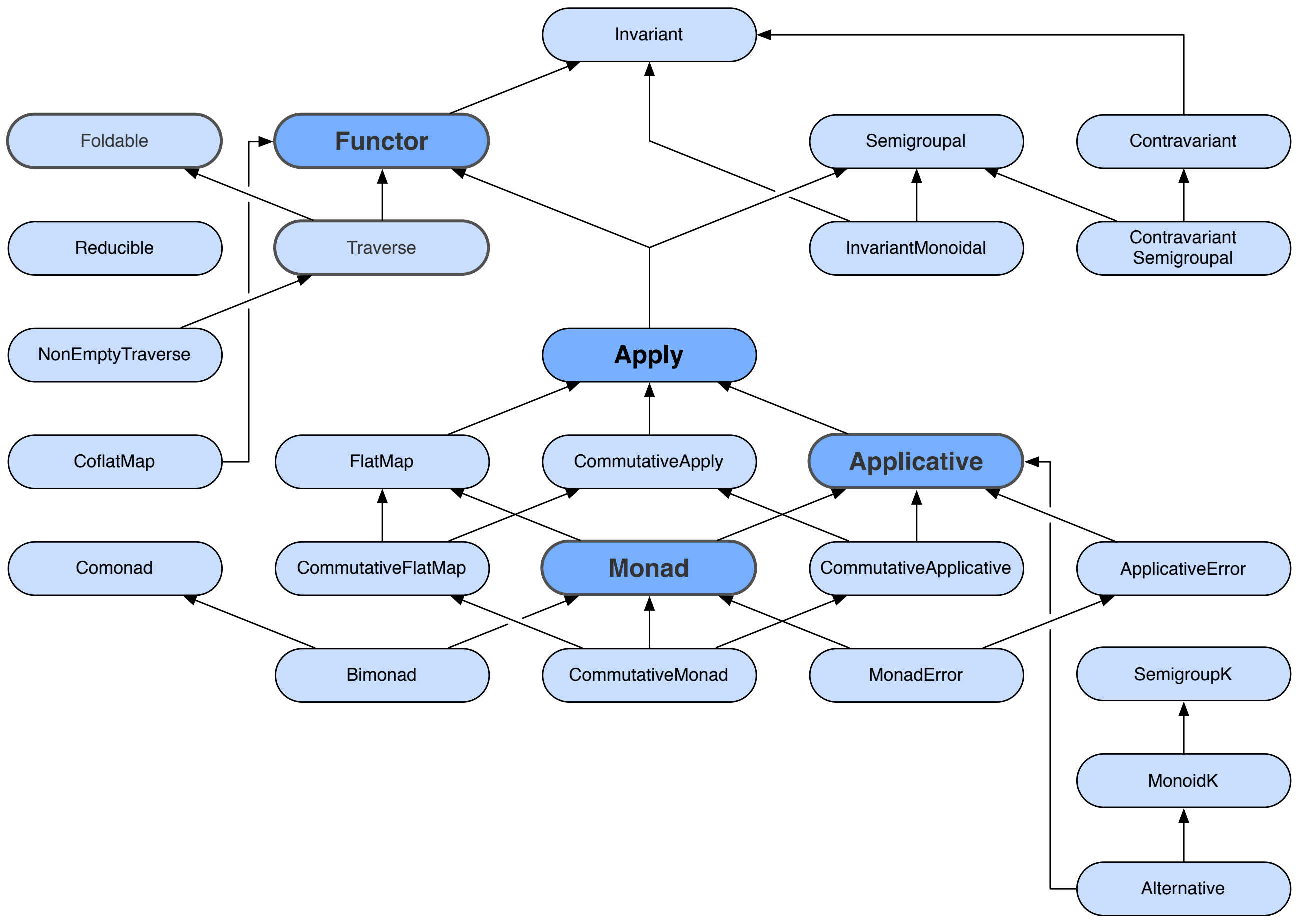
Let's talk about State Again

```
scala> damage.run(17)
res60: (Int, Seed) = (10,507111939)

scala> for { a ← damage; b ← damage } yield s"damages: $a, $b"
res61: State[Seed,String] = State($anon$1$$Lambd ...

scala> res61.run(17)._1
res62: String = damages: 10, 6

scala> (damage tuple damage).run(7)
res63: ((Int, Int), Seed) = ((4,7),452154665)
```

Is that it?


map
zip
pure
flatMap


Functor
Apply
Applicative
Monad



Invariant
Foldable
Functor
Semigroupal
Contravariant
Reducible
Traverse
InvariantMonoidal
Contravariant Semigroupal
NonEmptyTraverse
Apply
CoflatMap
FlatMap
CommutativeApply
Applicative
Comonad
CommutativeFlatMap
Monad
CommutativeApplicative
ApplicativeError
Bimonad
CommutativeMonad
MonadError
SemigroupK
MonoidK
Alternative

Thanks!

- Resources
  - Cats - see also link to Gitter channel.
  - *Functional Programming Essentials* by Kelley Robinson
  - *Functional Programming in Scala* by Rúnar and Chiusano