Functional programming is...

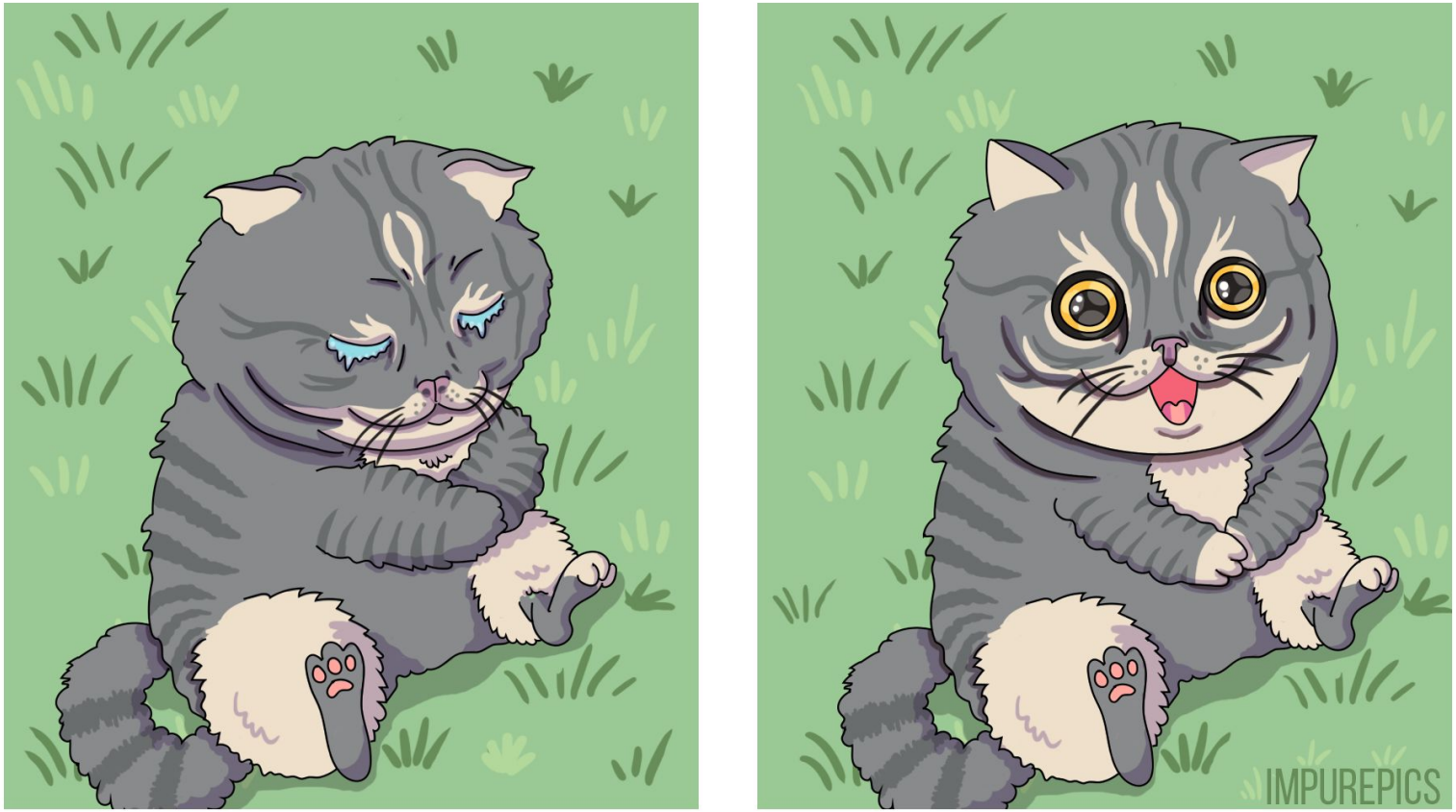
Aristov Ivan, KINOPLAN. October 2019.

...all about *functions*!

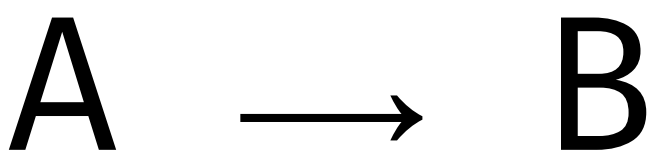
Agenda

- Functions itself
- Function properties, Referential transparency, Side-effects, Pure functions
- Functions composition
- Partial Function, Problems with Partial Functions
- Higher-kinded Types
- High-order functions
- Abstraction over Types, Typeclasses
- Typeclasses family
- Abstraction over Function types, Arrow Typeclass
- Few words about testing

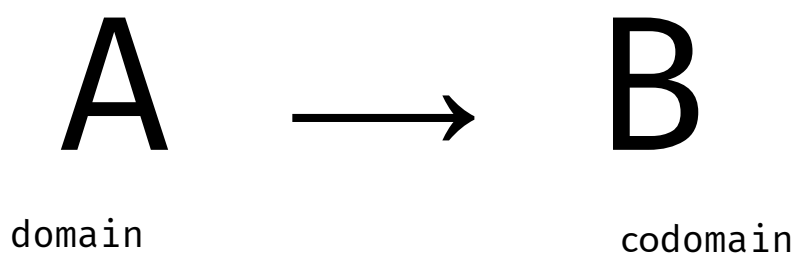
BEFORE AND AFTER **FUNCTIONAL PROGRAMMING** CAME INTO MY LIFE



Function



Function



INTRODUCTION TO FP
LECTURE 1. WHY FP?

REFERENTIAL TRANSPARENCY

So, function

- Everything can be represented as a function (values, classes, state, …)

So, function

- Everything can be represented as a function (values, classes, state, …)
- Total (computes at finite time)

So, function

- Everything can be represented as a function (values, classes, state, …)
- Total (computes at finite time)
- Referentially transparent (each computation get same result)

So, function

- Everything can be represented as a function (values, classes, state, …)
- Total (computes at finite time)
- Referentially transparent (each computation get same result)
- Immutable value (can’t change)

So, function

- Everything can be represented as a function (values, classes, state, …)
- Total (computes at finite time)
- Referentially transparent (each computation get same result)
- Immutable value (can’t change)
- Stateless (no dependencies)

So, function

- Everything can be represented as a function (values, classes, state, …)
- Total (computes at finite time)
- Referentially transparent (each computation get same result)
- Immutable value (can’t change)
- Stateless (no dependencies)
- Composable (g: A ⟶ B, f: B ⟶ C, g <<< f : A ⟶ C)

So, function

- Everything can be represented as a function (values, classes, state, …)
- Total (computes at finite time)
- Referentially transparent (each computation get same result)
- Immutable value (can’t change)
- Stateless (no dependencies)
- Composable (g: A ⟶ B, f: B ⟶ C, g <<< f : A ⟶ C)
- Easy to test (because of things from above)

Referential transparency

Expression can be replaced by its value without changing the behavior

Referential transparency

```
object MainTest extends App {
  def compute1 = { println("1") ; 1 }
  def compute2 = { println("2") ; 2 }

  compute1
  compute2
  compute1
}
```

Referential transparency

```
object MainTest extends App {
  def compute1 = { println("1") ; 1 }
  def compute2 = { println("2") ; 2 }

  compute1
  compute2
  compute1
}
```

Referential transparency

```
object MainTest extends App {
  val compute1 = { println("1") ; 1 }
  val compute2 = { println("2") ; 2 }

  compute1
  compute2
  compute1
}
```

Referential transparency

```
object MainTest extends App {
  val compute1 = { println("1") ; 1 }
  val compute2 = { println("2") ; 2 }

  compute1
  compute2
  compute1
}
```

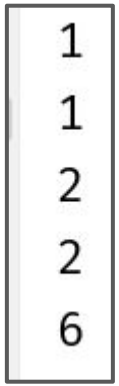
```
1
2
```

Referential transparency

```
object MainTest extends App {
  import scala.concurrent.{Future, Await}
  import scala.concurrent.duration.Duration
  import scala.concurrent.ExecutionContext.Implicits.global

  def compute1 = { println("1") ; 1 }
  def compute2 = { println("2") ; 2 }

  def fa = Future(compute1)
  def fb = Future(compute2)

  Await.result(
    for {
      a1 <- fa
      a2 <- fa
      b1 <- fb
      b2 <- fb
    } yield println(a1 + a2 + b1 + b2), Duration.Inf)
}
```

Referential transparency

```
object MainTest extends App {
  import scala.concurrent.{Future, Await}
  import scala.concurrent.duration.Duration
  import scala.concurrent.ExecutionContext.Implicits.global

  def compute1 = { println("1") ; 1 }
  def compute2 = { println("2") ; 2 }

  def fa = Future(compute1)
  def fb = Future(compute2)

  Await.result(
    for {
      a1 <- fa
      a2 <- fa
      b1 <- fb
      b2 <- fb
    } yield println(a1 + a2 + b1 + b2), Duration.Inf)
}
```

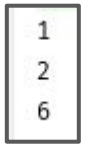


Referential transparency

```
object MainTest extends App {
  import scala.concurrent.{Future, Await}
  import scala.concurrent.duration.Duration
  import scala.concurrent.ExecutionContext.Implicits.global

  val compute1 = { println("1") ; 1 }
  val compute2 = { println("2") ; 2 }

  val fa = Future(compute1)
  val fb = Future(compute2)

  Await.result(
    for {
      a1 <- fa
      a2 <- fa
      b1 <- fb
      b2 <- fb
    } yield println(a1 + a2 + b1 + b2), Duration.Inf)
}
```

Referential transparency

```
object MainTest extends App {
  import scala.concurrent.{Future, Await}
  import scala.concurrent.duration.Duration
  import scala.concurrent.ExecutionContext.Implicits.global

  val compute1 = { println("1") ; 1 }
  val compute2 = { println("2") ; 2 }

  val fa = Future(compute1)
  val fb = Future(compute2)

  Await.result(
    for {
      a1 <- fa
      a2 <- fa
      b1 <- fb
      b2 <- fb
    } yield println(a1 + a2 + b1 + b2), Duration.Inf)
}
```



Referential transparency

```
object MainTest extends App {
  import scala.concurrent.{Future, Await}
  import scala.concurrent.duration.Duration
  import scala.concurrent.ExecutionContext.Implicits.global
  import java.util.concurrent.atomic.AtomicInteger

  val counter = new AtomicInteger(0)

  Await.result(
    for {
      a1 <- Future(counter.get)
      _  <- Future(counter.set(a1 + 1))
      a2 <- Future(counter.get)
    } yield println(a1 + a2), Duration.Inf)
}
```

Referential transparency

```
object MainTest extends App {
  import scala.concurrent.{Future, Await}
  import scala.concurrent.duration.Duration
  import scala.concurrent.ExecutionContext.Implicits.global
  import java.util.concurrent.atomic.AtomicInteger

  val counter = new AtomicInteger(0)

  Await.result(
    for {
      a1 <- Future(counter.get)
      _  <- Future(counter.set(a1 + 1))
      a2 <- Future(counter.get)
    } yield println(a1 + a2), Duration.Inf)
}
```

1

Referential transparency

```
object MainTest extends App {
  import scala.concurrent.{Future, Await}
  import scala.concurrent.duration.Duration
  import scala.concurrent.ExecutionContext.Implicits.global
  import java.util.concurrent.atomic.AtomicInteger

  val counter = new AtomicInteger(0)
  val get = Future(counter.get)

  Await.result(
    for {
      a1 <- get
      _  <- Future(counter.set(a1 + 1))
      a2 <- get
    } yield println(a1 + a2), Duration.Inf)
}
```

Referential transparency

```
object MainTest extends App {
  import scala.concurrent.{Future, Await}
  import scala.concurrent.duration.Duration
  import scala.concurrent.ExecutionContext.Implicits.global
  import java.util.concurrent.atomic.AtomicInteger

  val counter = new AtomicInteger(0)
  val get = Future(counter.get)

  Await.result(
    for {
      a1 <- get
      _  <- Future(counter.set(a1 + 1))
      a2 <- get
    } yield println(a1 + a2), Duration.Inf)
}
```

Referential transparency

```
object MainTest extends App {
  import cats.effect.IO
  import java.util.concurrent.atomic.AtomicInteger

  val counter = new AtomicInteger(0)
  val get = IO.delay(counter.get)

  (for {
    a1 <- get
    _  <- IO.delay(counter.set(a1 + 1))
    a2 <- get
    _  <- IO.delay(println(a1 + a2))
  } yield ()).unsafeRunSync()
}
```

Referential transparency

```
object MainTest extends App {
  import cats.effect.IO
  import java.util.concurrent.atomic.AtomicInteger

  val counter = new AtomicInteger(0)
  val get = IO.delay(counter.get)

  (for {
    a1 <- get
    _  <- IO.delay(counter.set(a1 + 1))
    a2 <- get
    _  <- IO.delay(println(a1 + a2))
  } yield ()).unsafeRunSync()
}
```

1

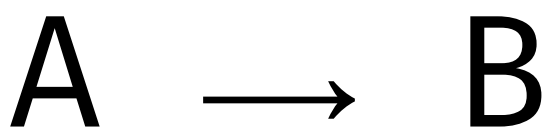

смотри сэнок ето чистая
функция
Господи как красива

Referential transparency

```
object MainTest extends IOApp {
  def run(args: List[String]): IO[ExitCode] = {
    for {
      ref <- Ref[IO].of(0)
      (a1, a2) <- ref.modify(a => (a + 1, a)).product(ref.get)
      _   <- IO.delay(println(a1 + a2))
    } yield ExitCode.Success
  }
}
```

Referential transparency

```
object MainTest extends IOApp {
  def run(args: List[String]): IO[ExitCode] = {
    for {
      ref <- Ref[IO].of(0)
      (a1, a2) <- ref.modify(a => (a + 1, a)).product(ref.get)
      _   <- IO.delay(println(a1 + a2))
    } yield ExitCode.Success
  }
}
```


A
B

Pure Function, Composition

$$f: A \longrightarrow B$$

$$g: B \longrightarrow C$$

Pure Function, Composition

$$f: A \longrightarrow B$$

$$g: B \longrightarrow C$$

$$j: A \longrightarrow C = g \lll f$$

Pure Function, Composition

$$f: A \longrightarrow B$$

$$g: B \longrightarrow C$$

$$j: A \longrightarrow C = g \lll f$$

$$(B \longrightarrow C) \lll (A \longrightarrow B): (A \longrightarrow C)$$

Pure Function, Composition

```
f: A → B

g: B → C

j: A → C = g <<< f

(B → C) <<< (A → B): (A → C)

(f: (A → B)) >>> (g: (B → C)): (A → C) =

g <<< f
```

Pure Function, Composition

```
object MainTest extends App {
  val a: Int => Boolean = _ % 2 == 0
}
```

a: Int => Boolean

Pure Function, Composition

```
object MainTest extends App {
  val a: Int => Boolean = _ % 2 == 0
  val b: Boolean => String = b => s"?$b?"
}
```

a: Int => Boolean

b: Boolean => String

? : Int => String

Pure Function, Composition

```
object MainTest extends App {
  val a: Int => Boolean = _ % 2 == 0
  val b: Boolean => String = b => s"?$b?"
  val c: String => String = s => s"\\$s\\"
}
```

a: Int => Boolean
b: Boolean => String
c: String => String

?: Int => String

Pure Function, Composition

```
object MainTest extends App {
  val a: Int => Boolean = _ % 2 == 0
  val b: Boolean => String = b => s"?$b?"
  val c: String => String = s => s"\\$s\\"

  val d: Int => String = c compose b compose a

  println(d.apply(2))
  println(d.apply(1))
}
```

a: Int => Boolean
b: Boolean => String
c: String => String

d: Int => String

Pure Function, Composition

```
object MainTest extends App {
  val a: Int => Boolean = _ % 2 == 0
  val b: Boolean => String = b => s"?$b?"
  val c: String => String = s => s"\\$s\\"

  val d: Int => String = c compose b compose a

  println(d.apply(2))
  println(d.apply(1))
}
```

a: Int => Boolean
b: Boolean => String
c: String => String

d: Int => String

```
\?true?\
\?false?\
```

$$A \longrightarrow ? \; B$$

$$A \longrightarrow^{?} B$$

$$\texttt{Int} \longrightarrow^{?} \texttt{Odd Int}$$

Partial Function

```
object MainTest extends App {
  val pfOddInt: PartialFunction[Int, Int] = {
    case int if int % 2 == 0 => int
  }

  println(pfOddInt.apply(3))
}
```

Partial Function

```
object MainTest extends App {
  val
    c
  }

  pri
}
```

```
Exception in thread "main" scala.MatchError: 3 (of class java.lang.Integer)
    at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:259)
    at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:257)
    at MainTest$$anonfun$1.applyOrElse(MainTest.scala:4)
    at MainTest$$anonfun$1.applyOrElse(MainTest.scala:4)
    at scala.runtime.AbstractPartialFunction$mcII$sp.apply$mcII$sp(AbstractPartialFunction.scala:38)
    at MainTest$.delayedEndpoint$MainTest$1(MainTest.scala:8)
    at MainTest$delayedInit$body.apply(MainTest.scala:2)
    at scala.Function0.apply$mcV$sp(Function0.scala:39)
    at scala.Function0.apply$mcV$sp$(Function0.scala:39)
    at scala.runtime.AbstractFunction0.apply$mcV$sp(AbstractFunction0.scala:17)
    at scala.App.$anonfun$main$1$adapted(App.scala:80)
    at scala.collection.immutable.List.foreach(List.scala:392)
```

Partial Function

```
object MainTest extends App {
  val pfOddInt: PartialFunction[Int, Int] = {
    case int if int % 2 == 0 => int
  }

  val lifted: Int => Option[Int] = pfOddInt.lift

  println(lifted.apply(3))
}
```

Partial Function

```
object MainTest extends App {
  val pfOddInt: PartialFunction[Int, Int] = {
    case int if int % 2 == 0 => int
  }

  val lifted: Int => Option[Int] = pfOddInt.lift

  println(lifted.apply(3))
}
```

None

Partial Function, Composition

```
object MainTest extends App {
  val pf1: PartialFunction[Int, Int] = {
    case int if int % 2 == 0 => int
  }

  val pf2: PartialFunction[Int, String] = {
    case int if int == 4 => "Success"
  }

  val pf3 = pf2 compose pf1

  println(pf3.apply(2))
}
```

Partial Function, Composition

```
object MainTest extends App {
  val pf1: PartialFunction[Int, Int] = {
    case int if int % 2 == 0 => int
  }

  val pf2:
    case i
  }

  val pf3

  println(
}
```

```
Exception in thread "main" scala.MatchError: 1 (of class java.lang.Integer)
    at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:259)
    at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:257)
    at MainTest$$anonfun$1.applyOrElse(MainTest.scala:4)
    at MainTest$$anonfun$1.applyOrElse(MainTest.scala:4)
    at scala.runtime.AbstractPartialFunction$mcII$sp.apply$mcII$sp(AbstractPartialFunction.scala:38)
    at scala.runtime.AbstractPartialFunction$mcII$sp.apply(AbstractPartialFunction.scala:38)
    at scala.runtime.AbstractPartialFunction$mcII$sp.apply(AbstractPartialFunction.scala:30)
    at scala.Function1.$anonfun$compose$1(Function1.scala:49)
    at MainTest$.delayedEndpoint$MainTest$1(MainTest.scala:14)
    at MainTest$delayedInit$body.apply(MainTest.scala:2)
    at scala.Function0.apply$mcV$sp(Function0.scala:39)
    at scala.Function0.apply$mcV$sp$(Function0.scala:39)
```

Partial Function, Composition

```
object MainTest extends App {
  val pf1: PartialFunction[Int, Int] = {
    case int if int % 2 == 0 => int
  }

  val pf2: PartialFunction[Int, String] = {
    case int if int == 4 => "Success"
  }

  val pf3: PartialFunction[Int, String] = pf2 compose pf1

  println(pf3.apply(2))
}
```

Partial Function, Composition

```
object MainTest extends App {
  val pf1: PartialFunction[Int, Int] = {
    case int if int % 2 == 0 => int
  }
  val pf2: PartialFunction[Int, String] = {
    case int if int % 2 == 1 => "odd"
  }
  val pf3: PartialFunction[Int, String] = pf2 compose pf1

  println(pf3.apply(2))
}
```

```
sbt:talks> compile
[info] Compiling 1 Scala source to /home/for/proj/talks/15-10-19-kinoplan/target/scala-2.12/classes ...
[error] /home/for/proj/talks/15-10-19-kinoplan/src/main/scala/MainTest.scala:10:47: type mismatch;
[error]  found   : Int => String
[error]  required: PartialFunction[Int,String]
[error]   val pf3: PartialFunction[Int, String] = pf2 compose pf1
[error]                                           ^
[error] one error found
[error] (talk / Compile / compileIncremental) Compilation failed
[error] Total time: 1 s, completed 13.10.2019 20:03:36
```

Composition

```
object MainTest extends App {
  val pf1: PartialFunction[Int, Int] = {
    case int if int % 2 == 0 => int
  }

  val pf2: PartialFunction[Int, String] = {
    case int if int == 4 => "Success"
  }

  val pfLifted1: Int => Option[Int] = pf1.lift
  val pfLifted2: Int => Option[String] = pf2.lift
  val pfLifted3: Int => Option[String] = pfLifted2 compose pfLifted1
}
```

Composition

```scala
object MainTest extends App {
  val pf1: PartialFunction[Int, Int] = {
    case int if int % 2 == 0 => int
  }
```

```
[error] /home/for/proj/talks/15-10-19-kinoplan/src/main/scala/MainTest.scala:12:60: type mismatch;
[error]  found   : Int => Option[Int]
[error]  required: Int => Int
[error]   val pfLifted3: Int => Option[String] = pfLifted2 compose pfLifted1
[error]                                                            ^
```

```scala
  val pfLifted1: Int => Option[Int] = pf1.lift
  val pfLifted2: Int => Option[String] = pf2.lift
  val pfLifted3: Int => Option[String] = pfLifted2 compose pfLifted1
}
```

Composition

```
object MainTest extends App {
  val f: Int => Option[Int] = ???
  val g: Int => Option[String] = ???

  val composed: Int => Option[String] = {
    pfLifted1.apply(_).flatMap(pfLifted2)
  }
}
```

Composition

```
object MainTest extends App {
  val f: Int => Option[Int] = ???
  val g: Int => Option[String] = ???
  val j: String => Option[List[Char]] = ???

  val composed: Int => Option[List[Char]] = {
    f.apply(_).flatMap(v => g(v).flatMap(j))
  }
}
```

Abstraction, Composition

```
object MainTest extends App {
  type =>?[A, B] = A => Option[B]

  val f: Int =>? Int = ???
  val g: Int =>? String = ???
  val j: String =>? List[Char] = ???

  def compose[A, B, C](f: B =>? C)(g: A =>? B): A =>? C = {
    v => g.apply(v).flatMap(f)
  }

  val composed: Int =>? List[Char] = compose(j)(compose(g)(f))
}
```

Abstraction, Composition

```
object MainTest extends App {
  trait A
  trait B { val list: List[A] }
  trait C { val list: List[B] }
  trait D { val list: List[C] }

  val extract: D => List[A] = ???
}
```

Abstraction, Composition

```
object MainTest extends App {
  trait A
  trait B { val list: List[A] }
  trait C { val list: List[B] }
  trait D { val list: List[C] }

  def compose[A, B, C](g: B => List[C])(f: A => List[B]): A => List[C] = {
    a => f.apply(a).flatMap(g)
  }

  val d2c: D => List[C] = _.list
  val c2b: C => List[B] = _.list
  val b2a: B => List[A] = _.list

  val extract: D => List[A] = compose(b2a)(compose(c2b)(d2c))
}
```

Abstraction, Composition

```
object MainTest extends App {
  type =>?[A, B] = A => Option[B]
  def compose[A, B, C](g: B =>? C)(f: A =>? B): A =>? C = f(_).flatMap(g)
  def compose[A, B, C](g: B => List[C])(f: A => List[B]): A => List[C] = {
    f.apply(_).flatMap(g)
  }
}
```

Abstraction, Composition

```
object MainTest extends App {
  type =>?[A, B] = A => Option[B]
  def compose[A, B, C](g: B =>? C)(f: A =>? B): A =>? C = f(_).flatMap(g)
  def compose[A, B, C](g: B => Future[C])(f: A => Future[B]): A =>? C = {
    f(_).flatMap(g)
  }

  def compose[A, B, C](g: B => List[C])(f: A => List[B]): A => List[C] = {
    f(_).flatMap(g)
  }

  def compose[A, B, C](g: B => Either[?, C])(f: A => Either[?, B]): A => Either[?, C] = {
    f(_).flatMap(g)
  }
}
```

Gotta go deeper…

Type is also a function

A: *

Type is also a function

$A : *$

$F[\_] : * \longrightarrow *$

Type is also a function

$A : *$

$F[\_] : * \longrightarrow *$

$F[\_, \_] : * \longrightarrow * \longrightarrow *$

Type is also a function

$A : *$

$F[\_] : * \longrightarrow *$

$F[\_, \_] : * \longrightarrow * \longrightarrow *$

$F[\_[\_], \_] : (* \longrightarrow *) \longrightarrow * \longrightarrow *$

Abstraction over types

```scala
trait MyFlatMap[F[_]] {
  def flatMap[A, B](a: F[A])(f: A => F[B]): F[B]
}
```

Abstraction over types

```
trait MyFlatMap[F[_]] {
  def flatMap[A, B](a: F[A])(f: A => F[B]): F[B]
}
```

```
object MainTest extends App {
  def compose[A, B, C, F[_]](g: B => F[C])(f: A => F[B])(
    implicit ins: MyFlatMap[F]
  ): A => F[C] = v => ins.flatMap(f(v))(g)
}
```

Abstraction over types

```
object MyFlatMap {
  implicit val optionFlatMap = new MyFlatMap[Option] {
    def flatMap[A, B](a: Option[A])(f: A => Option[B]) = {
      a.flatMap(f)
    }
  }

  implicit val listMyFlatMap = new MyFlatMap[List] {
    def flatMap[A, B](a: List[A])(f: A => List[B]) = {
      a.flatMap(f)
    }
  }

  implicit def mapMyFlatMap[K] = new MyFlatMap[Map[K, ?]] {
    def flatMap[A, B](a: Map[K, A])(f: A => Map[K, B]) = {
      a.flatMap { case (k, v) => f(v).get(k).map(k -> _) }
    }
  }
}
```

Previous examples

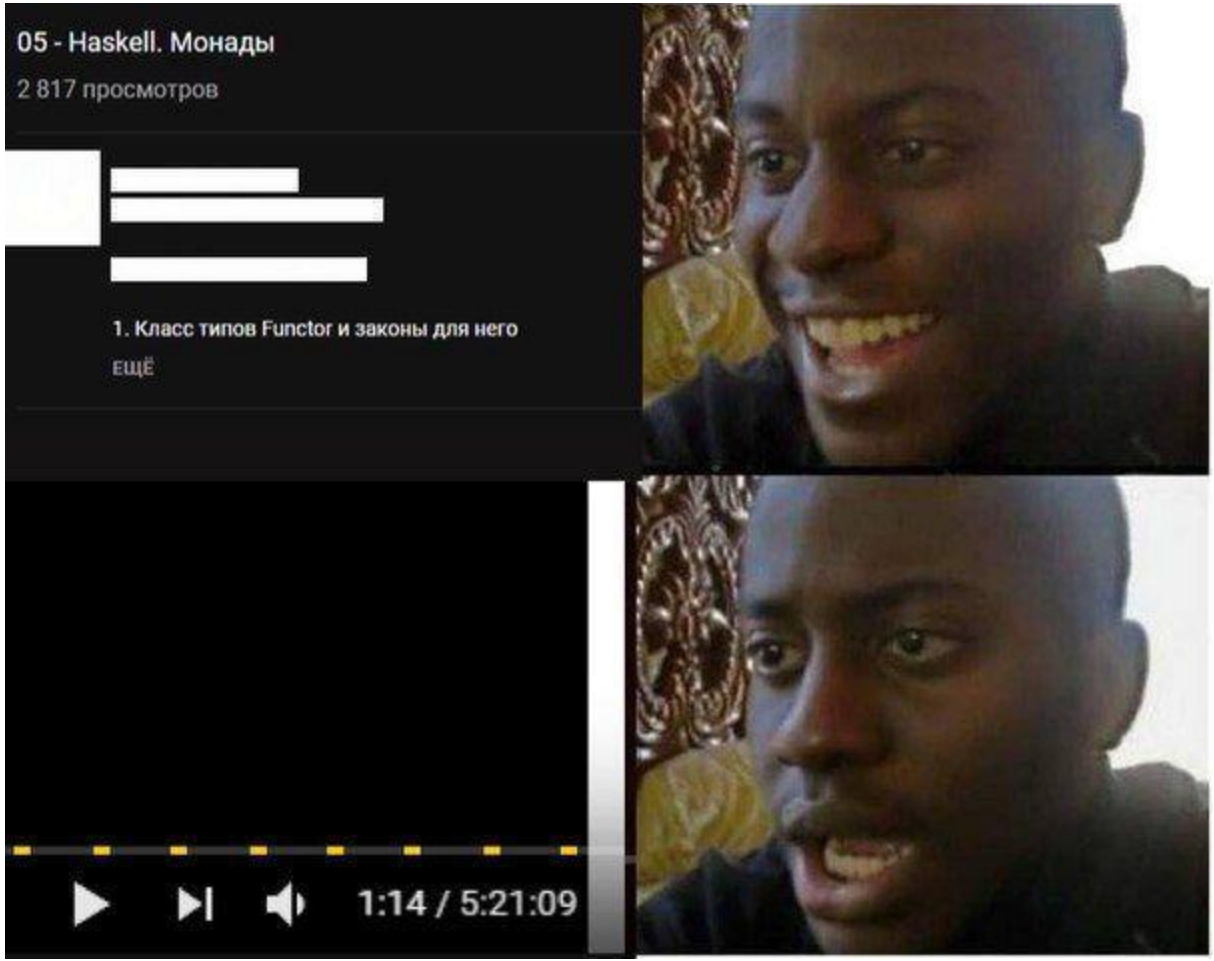
```
object MainTest extends App {
  def compose[A, B, C, F[_]](g: B => F[C])(f: A => F[B])(
    implicit ins: MyFlatMap[F]
  ): A => F[C] = v => ins.flatMap(f(v))(g)

  val f: Int => Option[Int] = ???
  val g: Int => Option[String] = ???
  val j: String => Option[List[Char]] = ???

  val composed: Int => Option[List[Char]] = compose(j)(compose(g)(f))
}
```

Previous examples

```
object MainTest extends App {
  def compose[A, B, C, F[_]](g: B => F[C])(f: A => F[B])(
    implicit ins: MyFlatMap[F]
  ): A => F[C] = v => ins.flatMap(f(v))(g)

  trait A
  trait B { val list: List[A] }
  trait C { val list: List[B] }
  trait D { val list: List[C] }

  val d2c: D => List[C] = _.list
  val c2b: C => List[B] = _.list
  val b2a: B => List[A] = _.list

  val extract: D => List[A] = compose(b2a)(compose(c2b)(d2c))
}
```

Introducing Monad Typeclass

```
trait Monad[F[_]] {
  def flatMap[A, B]: F[A] => (A => F[B]) => F[B]
}

object Monad {
  implicit val optionMonad = new Monad[Option] {
    def flatMap[A, B] = a => f => a.flatMap(f)
  }

  implicit val listMonad = new Monad[List] {
    def flatMap[A, B] = a => f => a.flatMap(f)
  }

  implicit def mapMonad[K] = new Monad[Map[K, ?]] {
    def flatMap[A, B] = a => f => a.flatMap { case (k, v) =>
      f(v).get(k).map(k -> _)
    }
  }
}
```


05 - Haskell. Монады
2 817 просмотров
1. Класс типов Functor и законы для него
ЕЩЁ
1:14 / 5:21:09

Rethinking

```
object MainTest extends App {
  def compose[A, B, C, F[_]](g: B => F[C])(f: A => F[B])(
    implicit ins: MyFlatMap[F]
  ): A => F[C] = v => ins.flatMap(f(v))(g)

  val f: Int => Option[Int] = ???
  val g: Int => Option[String] = ???
  val j: String => Option[List[Char]] = ???

  val composed: Int => Option[List[Char]] = compose(j)(compose(g)(f))
}
```

Rethinking

```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A]): MyFunc[F, C, B] = ???
}
```

Rethinking

```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A]): MyFunc[F, C, B] = ???
}
```

```
object MainTest extends App {
  val f: MyFunc[Option, Int, Int] = ???
  val g: MyFunc[Option, Int, String] = ???
  val j: MyFunc[Option, String, List[Char]] = ???

  val composed: MyFunc[Option, Int, List[Char]] = j.compose(g.compose(f))
}
```

Rethinking

```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A]): MyFunc[F, C, B] = ???
}
```

Rethinking

```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A]): MyFunc[F, C, B] = ???

  def andThen[C](g: MyFunc[F, B, C]): MyFunc[F, A, C] = {
    g.compose(this)
  }
}
```

Rethinking

```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A])(implicit ins: Monad[F]): MyFunc[F, C, B] = {
    MyFunc(c => ins.flatMap(g.run(c))(run))
  }

  def andThen[C](g: MyFunc[F, B, C])(implicit ins: Monad[F]): MyFunc[F, A, C] = {
    g.compose(this)
  }
}
```

Rethinking

```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A])(implicit ins: Monad[F]): MyFunc[F, C, B] = {
    MyFunc(c => ins.flatMap(g.run(c))(run))
  }

  def andThen[C](g: MyFunc[F, B, C])(implicit ins: Monad[F]): MyFunc[F, A, C] = {
    g.compose(this)
  }

  def map[C](f: B => C)(implicit ins: Monad[F]): MyFunc[F, A, C] = {
    MyFunc(a => ins.flatMap(run(a))(f))
  }
}
```

Rethinking

```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A])(implicit ins: Monad[F]): MyFunc[F, C, B] = {
    MyFunc(c => ins.flatMap(g.run(c))(run))
  }
```

```
[error] /home/for/proj/talks/15-10-19-kinoplan/src/main/scala/MyFunc.scala:11:43: type mismatch;
[error]  found   : B => C
[error]  required: B => F[C]
[error]     MyFunc(a => ins.flatMap[B, C](run(a))(f))
[error]                                           ^
```

```
  def map[C](f: B => C)(implicit ins: Monad[F]): MyFunc[F, A, C] = {
    MyFunc(a => ins.flatMap(run(a))(f))
  }
}
```

Introducing Functor Typeclass

```
trait Functor[F[_]] {
  def map[A, B]: F[A] => (A => B) => F[B]
}
```

Introducing Functor Typeclass

```
trait Functor[F[_]] {
  def map[A, B]: F[A] => (A => B) => F[B]
}
```

```
object Functor {
  implicit val optionFunctor = new Functor[Option] {
    def map[A, B] = a => f => a.map(f)
  }

  implicit val listFunctor = new Functor[List] {
    def map[A, B] = a => f => a.map(f)
  }

  implicit def mapFunctor[K] = new Functor[Map[K, ?]] {
    def map[A, B] = a => f => a.map { case (k, v) => (k, f(v)) }
  }
}
```

Introducing Functor Typeclass

```
trait Functor[F[_]] {
  def map[A, B]: F[A] => (A => B) => F[B]
}

case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A])(implicit ins: Monad[F]): MyFunc[F, C, B] = {...}

  def andThen[C](g: MyFunc[F, B, C])(implicit ins: Monad[F]): MyFunc[F, A, C] = {...}

  def map[C](f: B => C)(implicit ins: Functor[F]): MyFunc[F, A, C] = {
    MyFunc(a => ins.map(run(a))(f))
  }
}
```

Introducing Applicative Typeclass

```
trait Applicative[F[_]] {
  def pure[A]: A => F[A]

  def ap[A, B]: F[A => B] => F[A] => F[B]
}
```

Introducing Applicative Typeclass

```
trait Applicative[F[_]] extends Functor[F] {
  def pure[A]: A => F[A]

  def ap[A, B]: F[A => B] => F[A] => F[B]
}
```

Introducing Applicative Typeclass

```
trait Applicative[F[_]] extends Functor[F] {
  def pure[A]: A => F[A]

  def ap[A, B]: F[A => B] => F[A] => F[B]

  override def map[A, B] = a => f => ap(pure(f))(a)
}
```

Rethinking Monad Typeclass

```
trait Monad[F[_]] extends Applicative[F] {
  def flatMap[A, B]: F[A] => (A => F[B]) => F[B]

  override def map[A, B]: F[A] => (A => B) => F[B] = a => f => {
    flatMap(a)(v => pure(f(v)))
  }

  override def ap[A, B]: F[A => B] => F[A] => F[B] = fa => a => {
    flatMap(fa)(map(a)(_))
  }
}
```

Rethinking Monad Typeclass

```
object Monad {
  implicit val optionMonad = new Monad[Option] {
    def flatMap[A, B] = a => f => a.flatMap(f)

    def pure[A] = Some(_)
  }

  implicit val listMonad = new Monad[List] {
    def flatMap[A, B] = a => f => a.flatMap(f)

    def pure[A] = List(_)
  }

  implicit def mapMonad[K] = new Monad[Map[K, ?]] {
    def flatMap[A, B] = a => f => a.flatMap { case (k, v) =>
      f(v).get(k).map(k -> _)
    }

    def pure[A] = ???
  }
}
```

Introducing Apply Typeclass

```
trait Apply[F[_]] {
  def ap[A, B]: F[A => B] => F[A] => F[B]
}
```

Introducing FlatMap Typeclass

```
trait Apply[F[_]] {
  def ap[A, B]: F[A => B] => F[A] => F[B]
}
```

```
trait FlatMap[F[_]] extends Apply[F] with Functor[F] {
  def flatMap[A, B]: F[A] => (A => F[B]) => F[B]

  override def ap[A, B]= fa => a => flatMap(fa)(map(a)(_))
}
```

Rethinking Applicative Typeclass

```
trait Applicative[F[_]] extends Apply[F] with Functor[F] {
  def pure[A]: A => F[A]

  override def map[A, B] = a => f => ap(pure(f))(a)
}
```

Rethinking Monad Typeclass

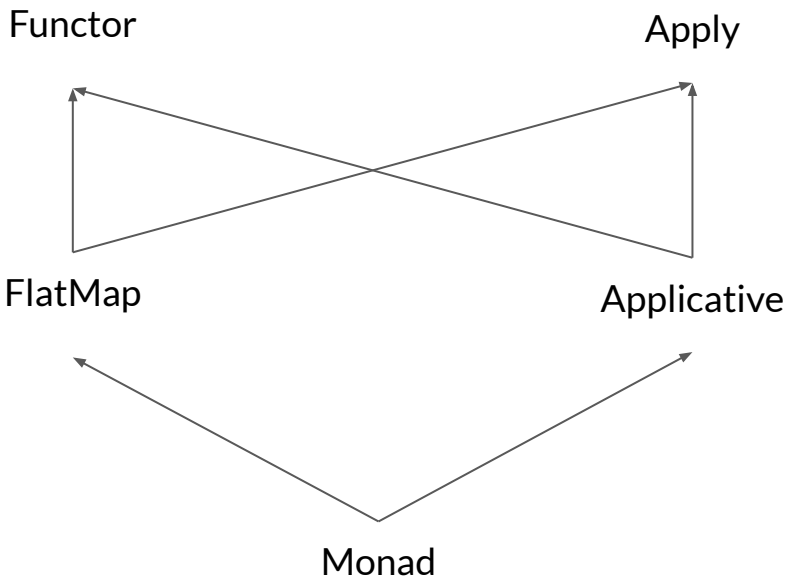
```
trait Monad[F[_]] extends Applicative[F] with FlatMap[F] {
  override def map[A, B] = a => f => {
    flatMap(a)(v => pure(f(v)))
  }

  override def ap[A, B] = fa => a => {
    flatMap(fa)(map(a)(_))
  }
}
```

FlatMap instance for Map

```
object FlatMap {
  implicit def mapFlatMap[K] = new FlatMap[Map[K, ?]] {
    def flatMap[A, B] = a => f => a.flatMap { case (k, v) =>
      f(v).get(k).map(k -> _)
    }

    def map[A, B] = a => f => a.map { case (k, v) => (k, f(v)) }
  }
}
```

Rethinking MyFunc

```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A])(implicit ins: FlatMap[F]): MyFunc[F, C, B] = {
    MyFunc(c => ins.flatMap(g.run(c))(run))
  }

  def andThen[C](g: MyFunc[F, B, C])(implicit ins: FlatMap[F]): MyFunc[F, A, C] = {
    g.compose(this)
  }

  def map[C](f: B => C)(implicit ins: Functor[F]): MyFunc[F, A, C] = {
    MyFunc(a => ins.map(run(a))(f))
  }
}
```


Functor
Apply
FlatMap
Applicative
Monad


DID YOU USE EVERY ABSTRACTION POSSIBLE?
YES.
AND WHAT DID IT COST?
EVERYTHING.
imgflip.com

Dive deeper…

Can we create a Monad instance for functions?

```
implicit def functionMonad[Y] = new Monad[Y => ?] {
  def pure[A]: A => Y => A = ???

  def flatMap[A, B]: (Y => A) => (A => Y => B) => Y => B = ???
}
```

Can we create a Monad instance for functions?

```
implicit def functionMonad[Y] = new Monad[Y => ?] {
  def pure[A]: A => Y => A = a => _ => a

  def flatMap[A, B] = a => f => arg => f(a(arg))(arg)
}
```

Can we create a Monad instance for MyFunc?

```
implicit def myFuncMonad[F[_], Y](implicit ins: Monad[F]) = {
  new Monad[MyFunc[F, Y, ?]] {
    def pure[A] = ???
    def flatMap[A, B] = ???
  }
}
```

Can we create a Monad instance for MyFunc?

```
implicit def myFuncMonad[F[_], Y](implicit ins: Monad[F]) = {
  new Monad[MyFunc[F, Y, ?]] {
    def pure[A] = a => MyFunc(_ => ins.pure(a))
    def flatMap[A, B] = a => f => MyFunc { y =>
      ins.flatMap(a.run(y))(f(_).run(y))
    }
  }
}
```

Rethinking again...

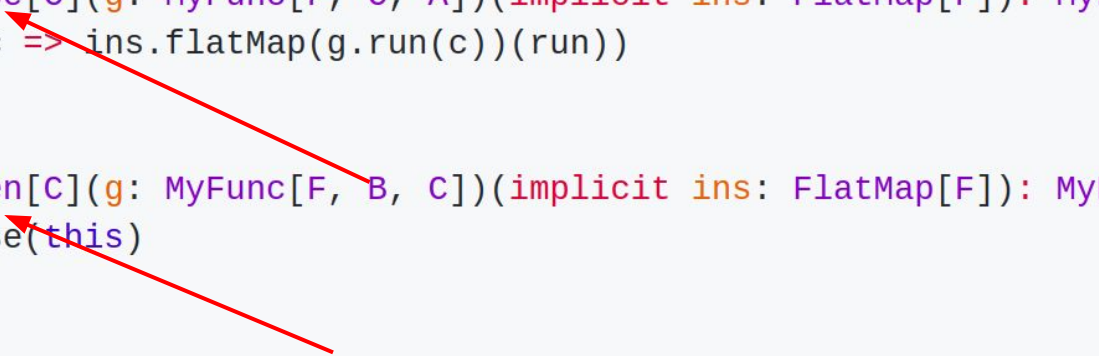
```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A])(implicit ins: FlatMap[F]): MyFunc[F, C, B] = {
    MyFunc(c => ins.flatMap(g.run(c))(run))
  }

  def andThen[C](g: MyFunc[F, B, C])(implicit ins: FlatMap[F]): MyFunc[F, A, C] = {
    g.compose(this)
  }
}
```

Rethinking again...

```
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A])(implicit ins: FlatMap[F]): MyFunc[F, C, B] = {
    MyFunc(c => ins.flatMap(g.run(c))(run))
  }

  def andThen[C](g: MyFunc[F, B, C])(implicit ins: FlatMap[F]): MyFunc[F, A, C] = {
    g.compose(this)
  }
}
```

Rethinking again...

```scala
case class MyFunc[F[_], A, B](run: A => F[B]) {
  def compose[C](g: MyFunc[F, C, A])(implicit ins: FlatMap[F]): MyFunc[F, C, B] = {
    MyFunc(c => ins.flatMap(g.run(c))(run))
  }

  def andThen[C](g: MyFunc[F, B, C])(implicit ins: FlatMap[F]): MyFunc[F, A, C] = {
    g.compose(this)
  }
}
```



Boilerplate :`)

Arrow Typeclass

```
trait Arrow[F[_, _]] {
  def lift[A, B]: (A => B) => F[A, B]

  def compose[A, B, C]: F[B, C] => F[A, B] => F[A, C]

  def andThen[A, B, C]: F[A, B] => F[B, C] => F[A, C] = {
    fa => compose(_)(fa)
  }

  def first[A, B, C]: F[A, B] => F[(A, C), (B, C)]

  def second[A, B, C]: F[A, B] => F[(C, A), (C, B)]
}
```

Arrow Typeclass instance for Function

```
implicit val functionArrow = new Arrow[Function1] {
  def lift[A, B]: (A => B) => A => B = ???
  def compose[A, B, C]: (B => C) => (A => B) => A => C = ???
  def first[A, B, C]: (A => B) => ((A, C)) => (B, C) = ???
  def second[A, B, C]: (A => B) => ((C, A)) => (C, B) = ???
}
```

Arrow Typeclass instance for Function

```
implicit val functionArrow = new Arrow[Function1] {
  def lift[A, B] = identity
  def compose[A, B, C] = g => f => g compose f
  def first[A, B, C] = f => { case (a, c) => (f(a), c) }
  def second[A, B, C] = f => { case (c, a) => (c, f(a)) }
}
```

Arrow Typeclass instance for MyFunc

```
implicit def myFuncArrow[F[_]](implicit ins: Monad[F]) = {
  new Arrow[MyFunc[F, ?, ?]] {
    def lift[A, B]: (A => B) => MyFunc[F, A, B] = ???
    def compose[A, B, C]: MyFunc[F, B, C] => MyFunc[F, A, B] => MyFunc[F, A, C] = ???
    def first[A, B, C]: MyFunc[F, A, B] => MyFunc[F, (A, C), (B, C)] = ???
    def second[A, B, C]: MyFunc[F, A, B] => MyFunc[F, (C, A), (C, B)] = ???
  }
}
```

Arrow Typeclass instance for MyFunc

```
implicit def myFuncArrow[F[_]](implicit ins: Monad[F]) = {
  new Arrow[MyFunc[F, ?, ?]] {
    def lift[A, B] = f => MyFunc(arg => ins.pure(f(arg)))
    def compose[A, B, C]: MyFunc[F, B, C] => MyFunc[F, A, B] => MyFunc[F, A, C] = ???
    def first[A, B, C]: MyFunc[F, A, B] => MyFunc[F, (A, C), (B, C)] = ???
    def second[A, B, C]: MyFunc[F, A, B] => MyFunc[F, (C, A), (C, B)] = ???
  }
}
```

Arrow Typeclass instance for MyFunc

```
implicit def myFuncArrow[F[_]](implicit ins: Monad[F]) = {
  new Arrow[MyFunc[F, ?, ?]] {
    def lift[A, B] = f => MyFunc(arg => ins.pure(f(arg)))

    def compose[A, B, C] = g => f => MyFunc { arg =>
      ins.flatMap(f.run(arg))(g.run)
    }

    def first[A, B, C]: MyFunc[F, A, B] => MyFunc[F, (A, C), (B, C)] = ???

    def second[A, B, C]: MyFunc[F, A, B] => MyFunc[F, (C, A), (C, B)] = ???
  }
}
```

Arrow Typeclass instance for MyFunc

```
implicit def myFuncArrow[F[_]](implicit ins: Monad[F]) = {
  new Arrow[MyFunc[F, ?, ?]] {
    def lift[A, B] = f => MyFunc(arg => ins.pure(f(arg)))

    def compose[A, B, C] = g => f => MyFunc { arg =>
      ins.flatMap(f.run(arg))(g.run)
    }

    def first[A, B, C] = f => MyFunc {
      case (a, c) => ins.map(f.run(a))((_, c))
    }

    def second[A, B, C]: MyFunc[F, A, B] => MyFunc[F, (C, A), (C, B)] = ???
  }
}
```

Arrow Typeclass instance for MyFunc

```
implicit def myFuncArrow[F[_]](implicit ins: Monad[F]) = {
  new Arrow[MyFunc[F, ?, ?]] {
    def lift[A, B] = f => MyFunc(arg => ins.pure(f(arg))
    def compose[A, B, C] = g => f => MyFunc { arg =>
      ins.flatMap(f.run(arg))(g.run)

    def first[A, B, C] = f => MyFunc {
      case (a, c) => ins.map(f.run(a))((_, c))

    def second[A, B, C] = f => MyFunc {
      case (c, a) => ins.map(f.run(a))((c, _))
    }
  }
}
```


DID YOU USE EVERY ABSTRACTION POSSIBLE?
YES
AND WHAT DID IT COST?
NOTHING
imgflip.com

But honestly...

- You can find our typeclasses in Cats/Scalaz library

But honestly...

- You can find our typeclasses in Cats/Scalaz library
- Our ***MyFunc*** data type is ***Kleisli*** data type from Cats/Scalaz library

But honestly...

- You can find our typeclasses in Cats/Scalaz library
- Our ***MyFunc*** data type is ***Kleisli*** data type from Cats/Scalaz library
- ***Kleisli*** is part of ***ReaderT*** pattern

But honestly...

- You can find our typeclasses in Cats/Scalaz library
- Our ***MyFunc*** data type is ***Kleisli*** data type from Cats/Scalaz library
- ***Kleisli*** is part of ***ReaderT*** pattern
- ***ReaderT*** is pretty common and most used pattern in FP

Testing

Testing (old)

```
def updateByParams(cinemaId: Int, saleRequest: ScheduleUpdateSaleStatusRequest): WriteResult =
    scheduleSeanceDAO.updateByParams(cinemaId, saleRequest)
```

```
"call service method with valid user.cinemaId" in {
  when(scheduleSeanceService.updateByParams(anyInt(), any())).thenAnswer(new Answer[WriteResult] {
    override def answer(invocation: InvocationOnMock) = {
      invocation.getArgument(0).asInstanceOf[Int] mustBe 0
      mock[WriteResult]
    }
  })
  controller.updateByParams().apply(FakeRequest().withJsonBody(json))
}
```

Testing (old)

Mockito matchers

```
def updateByParams(cinemaId: Int, saleRequest: ScheduleUpdateSaleStatusRequest): WriteResult =
    scheduleSeanceDAO.updateByParams(cinemaId, saleRequest)
```

```
"call service method with valid user.cinemaId" in {
  when(scheduleSeanceService.updateByParams(anyInt(), any())).thenAnswer(new Answer[WriteResult] {
    override def answer(invocation: InvocationOnMock) = {
      invocation.getArgument(0).asInstanceOf[Int] mustBe 0
      mock[WriteResult]
    }
  })
  controller.updateByParams().apply(FakeRequest().withJsonBody(json))
}
```

Testing (old)

```
def updateByParams(cinemaId: Int, saleRequest: ScheduleUpdateSaleStatusRequest): WriteResult =
    scheduleSeanceDAO.updateByParams(cinemaId, saleRequest)
```

```
"call service method with valid user.cinemaId" in {
  when(scheduleSeanceService.updateByParams(anyInt(), any())).thenAnswer(new Answer[WriteResult] {
    override def answer(invocation: InvocationOnMock) = {
      invocation.getArgument(0).asInstanceOf[Int] mustBe 0
      mock[WriteResult]
    }
  })
  controller.updateByParams().apply(FakeRequest().withJsonBody(json))
}
```

Mockito matchers

Lack types

Testing (old)

```
def updateByParams(cinemaId: Int, saleRequest: ScheduleUpdateSaleStatusRequest): WriteResult =
    scheduleSeanceDAO.updateByParams(cinemaId, saleRequest)
```

```
"call service method with valid user.cinemaId" in {
  when(scheduleSeanceService.updateByParams(anyInt(), any())).thenAnswer(new Answer[WriteResult] {
    override def answer(invocation: InvocationOnMock) = {
      invocation.getArgument(0).asInstanceOf[Int] mustBe 0
      mock[WriteResult]
    }
  })
  controller.updateByParams().apply(FakeRequest().withJsonBody(json))
}
```

Mockito matchers

Unknown arity

Lack types

Testing (old)

```
def updateByParams(cinemaId: Int, saleRequest: ScheduleUpdateSaleStatusRequest): WriteResult =
    scheduleSeanceDAO.updateByParams(cinemaId, saleRequest)
```

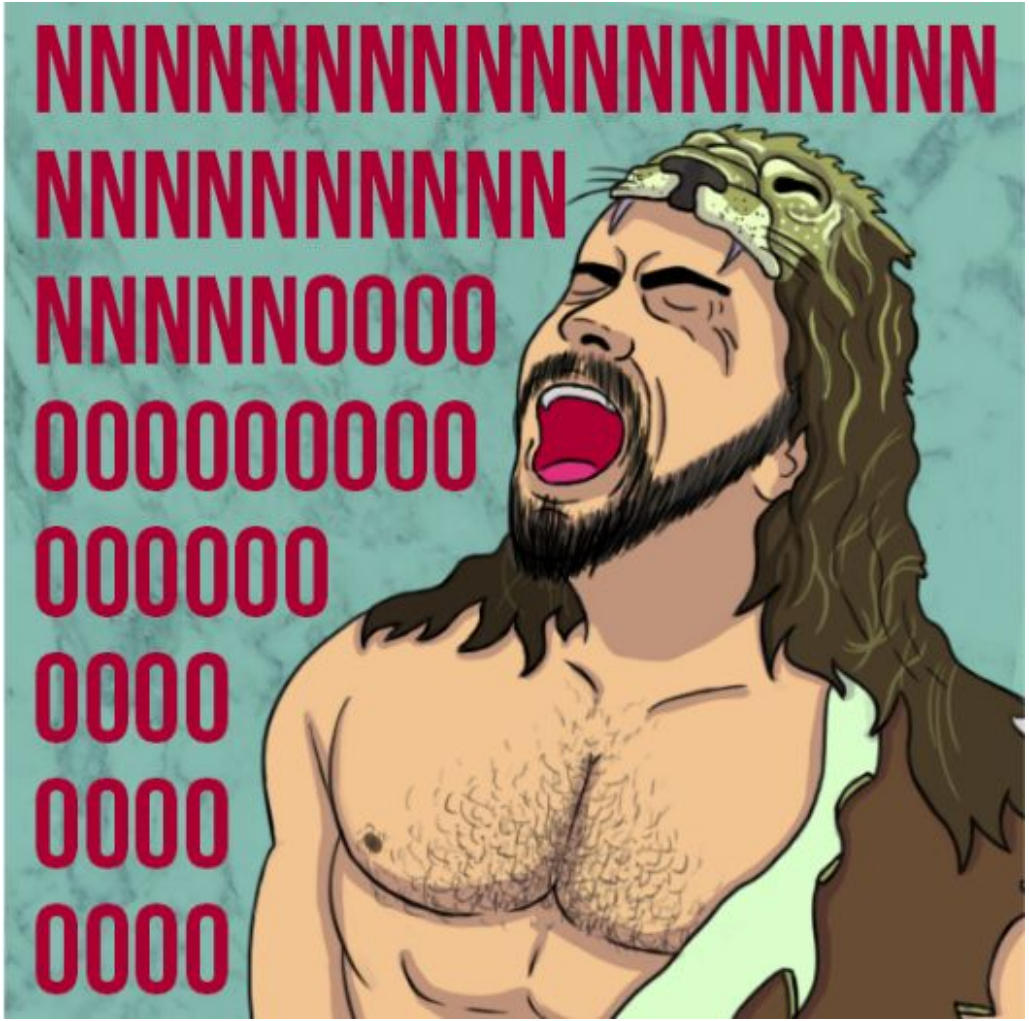
```
"call service method with valid user.cinemaId" in {
  when(scheduleSeanceService.updateByParams(anyInt(), any())).thenAnswer(new Answer[WriteResult] {
    override def answer(invocation: InvocationOnMock) = {
      invocation.getArgument(0).asInstanceOf[Int] mustBe 0
      mock[WriteResult]
    }
  })
  controller.updateByParams().apply(FakeRequest().withJsonBody(json))
}
```

Mockito matchers

Lack types

Unknown arity

Mock result?


NNNNNNNNNNNNNNNNNN
NNNNNNNNN
NNNNNOOOO
OOOOOOOOO
OOOOOO
OOOO
OOOO
OOOO

Testing (new)

```
val updateByParams: Kleisli[IO, (Int, ScheduleUpdateSaleStatusRequest), Unit] = {
  Kleisli { case (cinemaId, saleRequest) =>
    IO.delay(scheduleSeanceDAO.updateByParams(cinemaId, saleRequest))
  }
}
```

Testing (new)

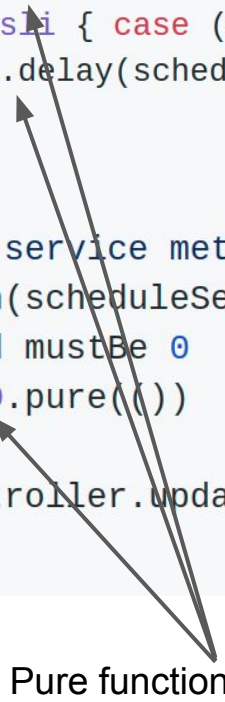
```
val updateByParams: Kleisli[IO, (Int, ScheduleUpdateSaleStatusRequest), Unit] = {
  Kleisli { case (cinemaId, saleRequest) =>
    IO.delay(scheduleSeanceDAO.updateByParams(cinemaId, saleRequest))
  }
}
```

```
"call service method with valid user.cinemaId" in {
  when(scheduleSeanceService.updateByParams).thenReturn(Kleisli { case (id, _) =>
    id mustBe 0
    IO.pure(())
  })
  controller.updateByParams().apply(FakeRequest().withJsonBody(json))
}
```

Testing (new)

```
val updateByParams: Kleisli[IO, (Int, ScheduleUpdateSaleStatusRequest), Unit] = {
  Kleisli { case (cinemaId, saleRequest) =>
    IO.delay(scheduleSeanceDAO.updateByParams(cinemaId, saleRequest))
  }
}
```

```
"call service method with valid user.cinemaId" in {
  when(scheduleSeanceService.updateByParams).thenReturn(Kleisli { case (id, _) =>
    id mustBe 0
    IO.pure(())
  })
  controller.updateByParams().apply(FakeRequest().withJsonBody(json))
}
```

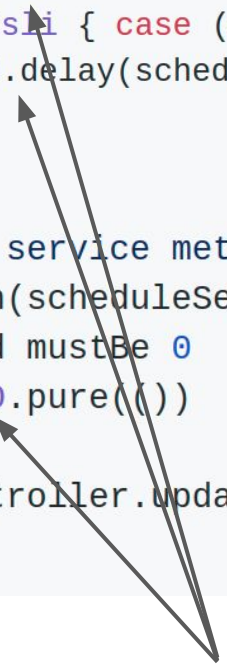


Pure function

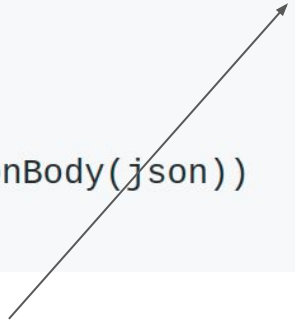
Testing (new)

```
val updateByParams: Kleisli[IO, (Int, ScheduleUpdateSaleStatusRequest), Unit] = {
  Kleisli { case (cinemaId, saleRequest) =>
    IO.delay(scheduleSeanceDAO.updateByParams(cinemaId, saleRequest))
  }
}
```

```
"call service method with valid user.cinemaId" in {
  when(scheduleSeanceService.updateByParams).thenReturn(Kleisli { case (id, _) =>
    id mustBe 0
    IO.pure(())
  })
  controller.updateByParams().apply(FakeRequest().withJsonBody(json))
}
```



Pure function



Typesafe, fully-known arity

Testing (new)

No more mockito matchers

```
val updateByParams: Kleisli[IO, (Int, ScheduleUpdateSaleStatusRequest), Unit] = {
  Kleisli { case (cinemaId, saleRequest) =>
    IO.delay(scheduleSeanceDAO.updateByParams(cinemaId, saleRequest))
  }
}

"call service method with valid user.cinemaId" in {
  when(scheduleSeanceService.updateByParams).thenReturn(Kleisli { case (id, _) =>
    id mustBe 0
    IO.pure(())
  })
  controller.updateByParams().apply(FakeRequest().withJsonBody(json))
}
```

Pure function

Typesafe, fully-known arity

Testing (new)

```
class AccessService @Inject()(accessDao: AccessDAO) {
  val add: Kleisli[IO, Access, Boolean] = writeResultRunner(AddError)(accessDao.add)
}

object AccessService {
  def writeResultRunner[T](e: => Throwable) = (f: Kleisli[IO, T, WriteResult]) => {
    f.map(_.ok).flatMapF(_.fold(IO.raiseError(e), IO.pure(true)))
  }
}
```

Testing (new)

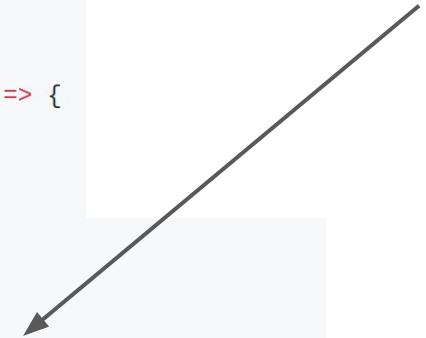
```
class AccessService @Inject()(accessDao: AccessDAO) {
  val add: Kleisli[IO, Access, Boolean] = writeResultRunner(AddError)(accessDao.add)
}

object AccessService {
  def writeResultRunner[T](e: => Throwable) = (f: Kleisli[IO, T, WriteResult]) => {
    f.map(_.ok).flatMapF(_.fold(IO.raiseError(e), IO.pure(true)))
  }
}
```

```
"return 200" in {
  val (service, controller) = setup
  when(service.add).thenReturn(Kleisli((_: Access) => IO.delay(true)))

  val result = controller.add(FakeRequest().withJsonBody(json))
  status(result) mustBe OK
  contentType(result) mustBe Some("application/json")
  contentAsJson(result) mustBe Json.obj("status" -> "ok")
}
```

Testing (new)

```
class AccessService @Inject()(accessDao: AccessDAO) {
  val add: Kleisli[IO, Access, Boolean] = writeResultRunner(AddError)(accessDao.add)
}

object AccessService {
  def writeResultRunner[T](e: => Throwable) = (f: Kleisli[IO, T, WriteResult]) => {
    f.map(_.ok).flatMapF(_.fold(IO.raiseError(e), IO.pure(true)))
  }
}
```

Pure function



```
"return 200" in {
  val (service, controller) = setup
  when(service.add).thenReturn(Kleisli((_: Access) => IO.delay(true)))

  val result = controller.add(FakeRequest().withJsonBody(json))
  status(result) mustBe OK
  contentType(result) mustBe Some("application/json")
  contentAsJson(result) mustBe Json.obj("status" -> "ok")
}
```

Testing (new)

No mockito matchers

Pure function

```
class AccessService @Inject()(accessDao: AccessDAO) {
  val add: Kleisli[IO, Access, Boolean] = writeResultRunner(AddError)(accessDao.add)
}

object AccessService {
  def writeResultRunner[T](e: => Throwable) = (f: Kleisli[IO, T, WriteResult]) => {
    f.map(_.ok).flatMapF(_.fold(IO.raiseError(e), IO.pure(true)))
  }
}
```

```
"return 200" in {
  val (service, controller) = setup
  when(service.add).thenReturn(Kleisli((_: Access) => IO.delay(true)))

  val result = controller.add(FakeRequest().withJsonBody(json))
  status(result) mustBe OK
  contentType(result) mustBe Some("application/json")
  contentAsJson(result) mustBe Json.obj("status" -> "ok")
}
```

Testing (new)

```
class AccessService @Inject()(accessDao: AccessDAO) {
  val add: Kleisli[IO, Access, Boolean] = writeResultRunner(AddError)(accessDao.add)
}

object AccessService {
  def writeResultRunner[T](e: => Throwable) = (f: Kleisli[IO, T, WriteResult]) => {
    f.map(_.ok).flatMapF(_.fold(IO.raiseError(e), IO.pure(true)))
  }
}
```

```
"return 200" in {
  val (service, controller) = setup
  when(service.add).thenReturn(Kleisli((_: Access) => IO.delay(true)))

  val result = controller.add(FakeRequest().withJsonBody(json))
  status(result) mustBe OK
  contentType(result) mustBe Some("application/json")
  contentAsJson(result) mustBe Json.obj("status" -> "ok")
}
```

No mockito matchers

Pure function

Typesafe,
fully-known arity

Further reading

- FP in Scala
- FP for mortals
- Scala with Cats
- Herding cats
- typelevel.org

Credits

- @impurepics


PURE FP
IS DE WAY