Pure Functional Programming

Local Reasoning and Controlled Effects

Felix Mulder

April 2018

Who am I?

- Scala 2.12 Docs Compiler
- Scala 3 Compiler Engineer @ EPFL w/ Martin Odersky
- Software Engineer @ Klarna Bank

Pure Functional Programming

What is pure FP?

“Programming using only functions”

or

“Programming without “side-effects””

Referential Transparency

An expression is said to be referentially transparent if it can be replaced with its corresponding value without changing the program's behavior.

Equational Reasoning

```
x = 5
y = x + x
z = 2 * y + x
```

Equational Reasoning

```
x = 5
y = 5 + 5
z = 2 * y + x
```

Equational Reasoning

```
x = 5
y = 10
z = 2 * y + x
```

Equational Reasoning

```
x = 5
y = 10
z = 2 * 10 + 5
```

Equational Reasoning

```
x = 5

y = 10

z = 25
```

Equational Reasoning

```scala
val const5 = 5

const5 + const5
// res0: Int = 10
```

Equational Reasoning

```scala
val const5  = Future(5)
// const5: scala.concurrent.Future[Int] = Future(<not completed>)

val const10 = const5.flatMap(x => const5.map(x + _))
// const10: scala.concurrent.Future[Int] = Future(<not completed>)

Await.result(const10, 1.second)
// res1: Int = 10
```

Equational Reasoning

```scala
val read = Future(io.StdIn.readInt())
// read: scala.concurrent.Future[Int] = Future(<not completed>)

// > 10

val read2 = read.flatMap(x => read.map(x + _))
// read2: scala.concurrent.Future[Int] = Future(<not completed>)

Await.result(read2, 5.seconds)
// res4: Int = 20
```

Equational Reasoning

```
val read2 = Future(io.StdIn.readInt()).flatMap {
  x ⇒ Future(io.StdIn.readInt()).map(x + _)
}
// read2: scala.concurrent.Future[Int] = Future(<not completed>)

// > 10

// > 20

Await.result(read2, 5.seconds)
// res9: Int = 30
```

Side Effects

- Future is not pure
- Neither is `StdIn.readInt`

The shapes themselves are.

Shapes

```scala
trait Future[A] {
  def result: A
}
```

Category Theory

A Monad is just a Monoid in the category of Endo-Functors, so what’s the problem?

Category Theory

The study of how these shapes behave and how they relate to each other

Category Theory

You already know a lot of these shapes and relations.

```scala
def map[B](f: A ⇒ B): List[B]
```

```scala
def map[B](f: A ⇒ B): F[B]
```

Functor

Type Classes

- Ad-hoc polymorphism

Type Classes

- Ad-hoc polymorphism
- Provided via implicits

Let’s implement this

Functor Laws

- Identity

Functor Laws

- Identity
- Composition

Cats

Cats

```scala
import cats._, cats.implicits._
// import cats._
// import cats.implicits._

Functor[List].map(List(1, 2, 3))(_ - 1)
// res10: List[Int] = List(0, 1, 2)
```

Cats

```scala
def parseInt(s: String) =
  Validated.catchOnly[NumberFormatException](s.toInt).toValidatedNel

List("1", "2", "a", "b").traverse(parseInt).show
// res11: String =
// Invalid(NEL(
//   java.lang.NumberFormatException: For input string: "a"
//   java.lang.NumberFormatException: For input string: "b"
// ))
```

Applicative

Applicative

Adds two functions:

- `pure`
- `ap`

Applicative - `pure`

```scala
trait Applicative[F[_]] {
  def pure[A](a: A): F[A]
  // ...
}
```

- `List.apply`
- `Option.apply`

Applicative - ap

```scala
trait Applicative[F[_]] {
  // ...

  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
}
```

ap in action

```scala
import cats._, cats.implicits._

val fa = List(1,2,3)
val ff: List[Int => String] = List(_.toString, x => (x * 2).toString)

ff.ap(fa)
// res12: List[String] = List(1, 2, 3, 2, 4, 6)

// or simply:

ff <*> fa
// res15: List[String] = List(1, 2, 3, 2, 4, 6)
```

Cartesian

Applicative Laws

- Identity
- Composition
- Homomorphism
- Interchange

Identity

```scala
((x: Int) => x).pure[List] <*> List(1, 2, 3) == List(1, 2, 3)
// res16: Boolean = true
```

Composition

```scala
val compose: (B => C) => (A => B) => (A => C) =
  bc => ab => bc compose ab

compose.pure[List] <*> u <*> v <*> w == u <*> (v <*> w)
// res17: Boolean = true
```

Homomorphism

```scala
val f: Int ⇒ String = _.toString
// f: Int ⇒ String = $$Lambda$3700/245829500@51112385

f.pure[List] <*> 1.pure[List] == f(1).pure[List]
// res18: Boolean = true
```

Interchange

```scala
val y: Int = 1
// y: Int = 1

val u: List[Int => String] = List(_.toString)
// u: List[Int => String] = List($$Lambda$3703/1518276566@4df9cb59)

u <*> y.pure[List] == ((f: Int => String) => f(y)).pure[List] <*> u
// res19: Boolean = true
```

Exercise

Two things to remember about Applicative

- “Disjoint” - i.e. parallel
- `pure` - lift a value into `F[_]`

Monads

Monads

```scala
val c1 =
  Future { /** let's assume we're making an async call here */ 1 }

val c2: Int ⇒ Future[Int] =
  x ⇒ Future { /** let's assume we're making another async call here */ x * 2 }
```

Now there's a dependency between `c2` and some value `x`. What if we want to chain calls to `c1` with a call to `c2`?

Monads

```
c1.map(c2)
// res20: scala.concurrent.Future[scala.concurrent.Future[Int]] = Future(<not completed>)
```

Monads

```
c1.map(c2).flatten
// res21: scala.concurrent.Future[Int] = Future(<not completed>)
```

Monads

```scala
c1.flatMap(c2)
// res22: scala.concurrent.Future[Int] = Future(<not completed>)
```

Monads

Monads are applicative functors with a continuation function referred to as “bind” and in scala `flatMap`

They can be thought of as “*composable* computation descriptions”

Monads

“Monads are things with `flatMap`” <- this isn’t completely wrong - ergo it’s not completely right.

Monad Laws

- Left identity

```
a.pure[M].flatMap(f) == f(a)
```

- Right identity

```
m.flatMap(pure) == m
```

- Associativity

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

Monad Laws - Left identity

```scala
val f: Int ⇒ List[Int] = x ⇒ List(x)
val g: Int ⇒ List[Int] = x ⇒ List(x * 2)

1.pure[List].flatMap(f)
// res23: List[Int] = List(1)

f(1)
// res24: List[Int] = List(1)

1.pure[List].flatMap(f) == f(1)
// res25: Boolean = true
```

Monad Laws - Right identity

```scala
List(1).flatMap(_.pure[List])
// res26: List[Int] = List(1)

List(1).flatMap(_.pure[List]) == List(1)
// res27: Boolean = true
```

Monad Laws - Associativity

```scala
List(1, 2).flatMap(f).flatMap(g)
// res28: List[Int] = List(2, 4)

List(1, 2).flatMap(x => f(x).flatMap(g))
// res29: List[Int] = List(2, 4)

List(1, 2).flatMap(f).flatMap(g) == List(1, 2).flatMap(x => f(x).flatMap(g))
// res30: Boolean = true
```

IO

IO

```scala
import cats.effect.IO
// import cats.effect.IO

val read = IO { io.StdIn.readInt() }
// read: cats.effect.IO[Int] = IO$764191693

(read, read).mapN(_ + _)
// res31: cats.effect.IO[Int] = <function1>
```

IO

```
(read, read).mapN(_ + _).unsafeRunSync() // > 1, > 2
// res32: Int = 3
```

Exercise

Traverse

Traverse

```scala
trait Traverse[F[_]] {
  def traverse[G[_]: Applicative, A, B](fa: F[G[A]])(f: A ⇒ G[B]): G[F[B]]
}
```

Traverse

```scala
List(Option(1), None, Option(3)).traverse(identity)
// res33: Option[List[Int]] = None

List(Option(1), Option(2), Option(3)).traverse(identity)
// res34: Option[List[Int]] = Some(List(1, 2, 3))
```

Traverse

```scala
def traverse[G[_]: Applicative, A, B](fa: List[A])(f: A ⇒ G[B]): G[List[B]] = {
  fa.foldRight(List.empty[B].pure[G]){ case (a, acc) ⇒
    Applicative[G].map2(f(a), acc)(_ :: _)
  }
}

traverse(List(Option(1), Option(2), Option(3)))(identity)
// res35: Option[List[Int]] = Some(List(1, 2, 3))
```

Traverse

```
List(Option(1), Option(2), Option(3)).traverse(identity)
// res36: Option[List[Int]] = Some(List(1, 2, 3))

List(Option(1), Option(2), Option(3)).sequence
// res37: Option[List[Int]] = Some(List(1, 2, 3))
```

Stream

Stream

```
fs2.Stream[F[_], A]
```

Why do we need another stream?

Purity & Control Flow

`Stream[F[_], A]`

- Emits `n` values of the type `A` where `n = 0, 1, ...`
- `A` is evaluated in the context of `F[_]`

Akka-Streams

Stream[F[_], A]

```scala
import fs2.Stream
// import fs2.Stream

Stream.emit(1, 2, 3, 4, 5).toList
// res38: List[(Int, Int, Int, Int, Int)] = List((1,2,3,4,5))
```

Stream[F[_], A]

```scala
val concat = Stream.eval(IO(1)) ++ Stream.eval(IO(2))

concat.repeat
```

Concatenating for Effect

```
Stream.eval(IO(println("hello!"))) ++ concat
// res40: fs2.Stream[cats.effect.IO,AnyVal] = Stream(..)
```

Concatenating for Effect

```scala
val hello = Stream.eval(IO(println("hello!"))).drain
// hello: fs2.Stream[cats.effect.IO,Nothing] = Stream(..)

hello ++ concat
// res41: fs2.Stream[cats.effect.IO,Int] = Stream(..)
```

Concurrency Primitives

- `Ref[F[_], A]`
- `Signal[F[_], A]`
- `Scheduler`

Principled vs Unprincipled

Control Flow

Exercise

Putting it all together - http4s

Kleisli Tripple

```scala
case class Kleisli[F[_], A, B](val run: A ⇒ F[B])
```

Kleisli is a Monad

http4s

```scala
import cats.data.Kleisli
import cats.effect.IO
import org.http4s.{Request, Response}

type HttpService[F[_]] = Kleisli[F, Request[F], Response[F]]
```

http4s

```scala
type HttpService[F[_]] = Kleisli[F, Request[F], Option[Response[F]]]
```

http4s

```scala
import cats.data.OptionT

type HttpService[F[_]] = Kleisli[OptionT[F, ?], Request[F], Response[F]]
```

http4s

```scala
import org.http4s._
import org.http4s.dsl.io._

val service = HttpService[IO] {
  case req => Ok("hello, whatever!")
}

val resp = service(Request[IO]())
// resp: cats.data.OptionT[cats.effect.IO,org.http4s.Response[cats.effect.IO]] = OptionT(

resp.value.unsafeRunSync()
// res0: Option[org.http4s.Response[cats.effect.IO]] = Some(Response(status=200, headers=
```