Purely Functional I/O in Scala

Rúnar Óli Bjarnason
@runarorama

Scala.IO, Paris 2013

What you should take away from this talk

- You do not need side-effects to do I/O.
- Purely functional I/O really is pure.
- It is also practical.
- How it's done and why it's done that way.

"Purely functional"

A pure function of type `(A => B)`
takes an argument of type `A`
and returns a value of type `B`.

And does nothing else.

Pure functions

A pure function always returns the same value given the same arguments.

Pure functions

A pure function has no dependencies other than its arguments.

Pure functions

The result of calling a pure function can be understood completely by looking at the returned value.

Pure functions are

- Compositional
- Modular
- Testable
- Scalable
- ***Comprehensible***

Pure functions are awesome. So why should we resort to side effects when doing I/O?

Problems with I/O side effects

- No separation of I/O code and logic
- Monolithic, non-modular, limited reuse
- Novel compositions are difficult
- Difficult to test
- Difficult to scale

```scala
class Cafe {
  def buyCoffee(cc: CreditCard): Coffee = {
    val cup = new Coffee()
    cc.charge(cup.price)
    cup
  }
}
```

```scala
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

```scala
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()
    (cup, new Charge(cc, cup.price))
  }
}
```

The big idea

Instead of performing I/O as a side effect,
return a value to the caller that describes
how we want to interact with the I/O system.

In short: embed an I/O scripting language in Scala.

```scala
abstract class Program {
  final def main(args: Array[String]): Unit =
    Program.unsafePerformIO(pureMain(args))

  def pureMain(args: IndexedSeq[String]): IO[Unit]
}

object Program {
  private def unsafePerformIO[A](a: IO[A]): A = ???
}
```

```
getLine: IO[String]

putLine: String => IO[Unit]
```

"Do I have to change all my functions to use `IO[T]` instead of `T`?"

```scala
trait IO[A] {
  def map[B](f: A => B): IO[B]


}

object IO {
  def pure[A](a: => A): IO[A]
}
```

```
trait IO[A] {
  def map[B](f: A => B): IO[B]

  def flatMap[B](f: A => IO[B]): IO[B]
}

object IO {
  def pure[A](a: => A): IO[A]
}
```

```
val ask: IO[Unit] = for {
  _    <- putLine("What is your name?")
  name <- getLine
  _    <- putLine("Hello, " ++ name)
} yield ()
```

```scala
val ask: IO[Unit] =
  putLine("What is your name?").flatMap { _ =>
    getLine.flatMap { name =>
      putLine("Hello, " ++ name)
    }
  }
```

Type safety

```
"Hello, " ++ getLine
```

```
error: type mismatch;
 found   : IO[String]
 required: String
              "Hello, " ++ getLine
                           ^
```

I/O monad

```scala
trait IO[A] {
  def map[B](f: A => B): IO[B] =
    flatMap(a => pure(a))

  def flatMap[B](f: A => IO[B]): IO[B]
}

object IO {
  def pure[A](f: => A): IO[A]
}
```

Monads

```scala
trait Monad[M[_]] {
  def flatMap[B](a: M[A])(f: A => M[B]): M[B]
  def pure[A](a: => A): M[A]
}
```

```scala
def sequence[A](ios: List[IO[A]]): IO[List[A]]

def traverse[A,B](as: List[A])(f: A => IO[B]): IO[List[B]]

def replicateM[A](n: Int, io: IO[A]): IO[List[A]]

def while(b: IO[Boolean]): IO[Unit]

def unzip(p: IO[(A,B)]): (IO[A], IO[B])

def join[A](a: IO[IO[A]]): IO[A]
```

```
val lines = List(
  "Háfrónskri og harðsoðinni",
  "hreintyngdur ég hefja megi",
  "brynfjörurpt með í mynni",
  "morgunmál á hverjum degi.")

val x = traverse(lines)(putLine)
```

What have we gained?

- Separation of I/O code from your logic
- Type safety
- First-class compositional I/O actions
- **Algebraic reasoning**
- Other benefits, depending on the implementation

The deferred effects model

```
class IO[A](run: () => A)
```

The deferred effects model

```scala
object IOMonad extends Monad[IO] {
  def pure[A](a: => A) = new IO(() => a)
  def flatMap[A,B](ma: IO[A])(
                   f: A => IO[B]): IO[B] =
    new IO { () => f(ma.run()).run() }
}
```

The world-as-state model

```
class IO[A](run: RealWorld => (A, RealWorld))
```

The world-as-state model

```
object IOMonad extends Monad[IO] {
  def pure[A](a: => A) = new IO(rw => (a, rw))
  def flatMap[A,B](ma: IO[A])(
                  f: A => IO[B]): IO[B] =
    new IO { rw =>
      val (a, rw1) = ma.run(rw)
      f(a).run(rw1)
    }
}
```

Example actions

```scala
def io[A](a: => A): IO[A] =
  new IO(() => a)

def putLine(s: String): IO[Unit] =
  io(println(s))

def getLine: IO[String] =
  io(readLine)
```

Problems

- A function is totally opaque.
- `RealWorld` is a lie.
- Conflates programs hang or crash with programs that remain productive.
- No story on concurrency.
- Haven't really gained any testability.
- StackOverflowError

Free monad model

```
sealed trait IO[F[_],A] { ... }

case class Return[F[_],A](a: A) extends IO[F,A]

case class Req[F[_],I,A](
  i: F[I],
  k: I => IO[F,A]) extends IO[F,A]
```

Free monad

```
sealed abstract class IO[F[_],A] {
  def flatMap[B](f: A => IO[F,B]): IO[F,B] =
    this match {
      case Return(a) => f(a)
      case Req(r, k) =>
        Req(r, k andThen (_ flatMap f))
    }
  def map[B](f: A => B): IO[F,B] =
    flatMap(a => Return(f(a)))
}
```

Console-only I/O

```
sealed trait Console[A]
case object GetLine extends Console[String]
case class PutLine(s: String) extends Console[Unit]

type ConsoleIO[A] = IO[Console,A]

val getLine: ConsoleIO[String] =
  Req(GetLine, s => Return(s))

def putLine(s: String): ConsoleIO[Unit] =
  Req(PutLine(s), _ => Return(()))
```

A console program

```
val ask: ConsoleIO[Unit] = for {
  _    <- putLine("What is your name?")
  name <- getLine
  _    <- putLine("Hello, " ++ name)
} yield ()
```

A console program

```
val ask: ConsoleIO[Unit] =
  Req(PutLine("What is your name?"), _ =>
    Req(GetLine, name =>
      Req(PutLine("Hello, " ++ name), _ =>
        Return(()))))
```

Any-effect I/O

```
type AnyIO[A] = IO[Function0, A]
```

Running actions

```scala
trait ~>[F[_],G[_]] {
  def apply[A](f: F[A]): G[A]
}

sealed abstract class IO[F[_],A] {
  ...
  def runIO[G[_]:Monad](f: F ~> G): G[A] = {
    val G = implicitly[Monad[G]]
    this match {
      case Return(a) => G.unit(a)
      case Req(r, k) =>
        G.bind(f(r))(k andThen (_.runIO(f)))
    }
  ...
}
```

Running actions

```
type Id[A] = A

object SideEffect extends (Function0 ~> Id) {
  def apply[A](f: Function0[A]): A = f()
}

def unsafePerformIO[A](io: IO[Function0, A]): A =
  io.runIO(SideEffect)
```

Running actions

```scala
implicit object ConsoleEffect extends (Console ~> Id) {
  def apply[A](c: Console[A]): A =
    r match {
      case GetLine => readLine
      case PutLine(s) => println(s)
    }
}
```

Running actions

```scala
case class InOut(in: List[String], out: List[String])
case class State[A](runState: InOut => (A, InOut))

object PureConsole extends (Console ~> State) {
  def apply[A](c: Console[A]): State[A] =
    State(s => (c, s) match {
      case (GetLine, InOut(in, out)) =>
        (in.head, InOut(in.tail, out))
      case (PutLine(l), InOut(in, out)) =>
        ((), InOut(in, l :: out))
    })
}
```

Running actions

```
scala> val ask = for {
     |   _    <- putLine("What is your name?")
     |   name <- getLine
     |   _    <- putLine("Hello, " ++ name)
     | } yield ()
ask: IO[Console, Unit] = IO@364032b7

scala> val s = ask.runIO(PureConsole)
s: State[Unit] = State(<function1>)

scala> val ls = s.runState(InOut(List("Alice"), Nil))
ls: InOut = InOut(Nil, List("Hello, Alice", "What is
your name?"))
```

Running actions

```
scala> val ask = for {
     |   _    <- putLine("What is your name?")
     |   name <- getLine
     |   _    <- putLine("Hello, " ++ name)
     | } yield ()
ask: IO[Console, Unit] = IO@364032b7

scala> val s = ask.runIO(ConsoleEffect)
What is your name?
```

Concurrency story

```
type AsyncIO[A] = IO[Future, A]
```

Breakpoints

```
def runUntilFailure[F[_],A](io: IO[F,A])(f: F ~> Id):
  Either[(Throwable, IO[F,A]), A] =
    io match {
      case Return(a) => Right(a)
      case Req(r, k) => try {
        runUntilFailure(k(f(r)))(f)
      } catch {
        case e: Throwable => Left((e, io))
      }
    }
```

What have we gained?

- An `IO` data type that we can inspect and is highly extensible.
- We can test programs without performing their I/O actions (e.g. `Console`).
- Concurrency: We simply build asynchronous requests into our `F` type.

StackOverflowError

- See ***Stackless Scala With Free Monads***, a paper from Scala Days 2012. `http://goo.gl/X0iO3M`
- See also `scalaz.Free`

SOE Problem

```
for {
  x <- a
  y <- b
  ...
} yield ()
```

SOE Problem

```
a.flatMap(av =>
  b.flatMap(bv =>
    c.flatMap(cv =>
      d.flatMap(dv =>
        e.flatMap(ev =>
          ...
```

SOE Solution

```
sealed abstract class IO[F[_],A]

case class Pure[F[_],A](a: A) extends IO[F,A]

case class Request[F[_],I,A](
  req: F[I],
  k: I => IO[A]) extends IO[A]

case class FlatMap[F[_],A,B](
  sub: IO[F,A],
  k: A => IO[F,B]
) extends IO[F,B]
```

Practical Streaming I/O

- The ***scalaz-stream*** library
- ***Advanced Stream Processing in Scala***
  Paul Chiusano, NEScala 2013.

Streaming I/O

```
sealed abstract class Process[+F[_],+O]

case class Emit[+F[_],+O](
  o: Seq[O],
  k: Process[F[_],O]) extends Process[F,O]

case class Await[+F[_],I,+O](
  req: F[I],
  k: I => Process[F,O],
  fallback: Process[F,O],
  cleanup: Process[F,O]) extends Process[F,O]

case class Halt(e: Throwable)
  extends Process[Nothing,Nothing]
```

Conclusion

- Purely functional I/O is possible and practical in Scala.
- It has a programming model vastly superior to relying on side-effects.
- The less powerful the representation, the more useful it is.