Practical

Pure I/O

in Scala with *ZIO*

Previously on Practical Pure I/O ...

Programs with side-effects on just one line ... are achievable.

The Promise of Functional Programming

- If we program with **functions**, defined as
  - **Deterministic**: same arguments implies same result
  - **Total**: result always available for arguments, no exception
  - **Pure**: only effect is computing result, **no side-effects**
- Then **typical refactorings cannot break a working program** 👍.

Programs as Pure Values

```
public interface Program<A> { // ...
    // Creating elementary programs
    static <A> Program<A> of(final Supplier<A> unsafeRun) { /* ... */ }
    static <A> Program<A> yield(final A a) { /* ... */ }

    // Combining programs to form larger programs
    default <B> Program<B> thenChain(final Function<A, Program<B>> f) { /* ... */ }
    default <B> Program<B> thenTransform(final Function<A, B> f) { /* ... */ }

    // Interpreting the resulting program
    static <A> A unsafeRun(final Program<A> program) { /* ... */ }
}
```

Impurity at the *Edge of the World*

```
public class HelloWorldApp {
    public static final Program<Unit> helloApp =
            putStrLn("What's your name?").thenChain(__ -> {
                return getStrLn().thenChain(name -> {
                    return putStrLn("Hello " + name + "!");
                });
            });

    public static void main(String[] args) {
        final Program<Unit> program = helloApp;
        // PURE, anything above done by creating and combining programs
        Program.unsafeRun(program); // IMPURE, only at the Edge of the World
    }
}
```

A Value Containing Void (`Unit`)

```
@Value.Immutable(singleton = true)
public abstract class Unit {
    public static Unit of() {
        return ImmutableUnit.of();
    }
}
```

Business-Ready Pure I/O

with *ZIO*

ZIO

Type-safe, composable, asynchronous and concurrent programming for Scala

— *https://scalaz.github.io/scalaz-zio/*

- Support for **synchronicity** and **asynchronicity**
- Support for **concurrency** with *fibers* and **interruptibility**
- Consistent **error model** (expected vs. unexpected), **resiliency** and **resource safety**
- Full **testability** with dependency injection
- Easy **debugging**
- **Performance** and **stack safety**

`IO[E, A]`

`IO[+E, +A] // IO<E, A>   E = error, A = Result`

- An immutable object that **describes a program** performing *side-effects*.
- An `IO` **does nothing**, it's pure.
- It must be interpreted by a **runtime system** or **RTS**
- Only when **run** by the RTS, it will either
  - **succeed** producing a **result** of type `A`,
  - or **fail** with an **error** of type `E`,
  - or **die** with an unexpected error.

Hello World!

```scala
object HelloWorldApp extends App {
  // Wraps synchronous (blocking) side-effecting code in an IO
  val helloWorld: IO[Nothing, Unit] =
    IO.effectTotal(/* () => */ Console.println("Hello World!"))
    // The IO just holds a lambda but does not run it for now.

  def run(args: List[String]): IO[Nothing, Int] = {
    helloWorld.either.fold(_ => 1, _ => 0)
  }
}
```

Simple `IO`

Success in IO

```
val success: IO[Nothing, Int] = IO.succeed(42)
val successLazy: IO[Nothing, Int] = IO.succeedLazy(/* () => */ 40 + 2)

// Will never fail (Nothing)
// Will always succeed with result 42 (Int)
```

Error Model

- `Fail`, **expected** error
  - **Domain** error, **business** error, transient error, anticipated error
  - Reflected in type as `E`
- `Die`, **unexpected** error
  - **System** error, **fatal** error, defect, unanticipated error
  - Not reflected in type

Failure in IO

```scala
val failure: IO[String, Nothing] = IO.fail("Failed")
// Will always fail with error "Failed" (String)
// will never succeed (Nothing)

val exceptionFailure: IO[FileNotFoundException, Nothing] =
  IO.fail(new FileNotFoundException("Expected"))
// Error can be an exception (but just as a value, never thrown!)
```

Death in `IO`

```scala
val death: IO[Nothing, Nothing] =
  IO.die(new IndexOutOfBoundsException("Unexpected"))
// Will never fail (Nothing)
// Will never succeed (Nothing)
// Will always die with error (not reflected in type)
// Error can only be an exception (but just as a value, never thrown!)

val deathMessage: IO[Nothing, Nothing] =
  IO.dieMessage("Unexpected")
// When just in need of a RuntimeException with a message
```

Wrapping in `IO`

Wrapping in `IO` as the Great Unifier

- `IO` integrates any kind of IO seamlessly into the same **unified model**.
- Abstracts over how **success** is returned and **failure** is signaled
  - Synchronous `return` and `throw`
  - Synchronous `Try` or `Either`
  - Asynchronous **callback**
  - Asynchronous `Future`
- Abstracts over **synchronicity**
- Abstracts over **interruptibility**

Synchronous in `IO`

```scala
def randomBetween(min: Int, max: Int): IO[Nothing, Int] = {
  // Side-effecting code updates the state of a random generator,
  // and returns a random number (Int).
  // It can never fail (Nothing).
  IO.effectTotal(/* () => */ Random.nextInt(max - min) + min)
}

def putStrLn(line: String): IO[Nothing, Unit] = {
  // Side-effecting code prints a line,
  // and returns void (Unit).
  // It can never fail (Nothing).
  IO.effectTotal(/* () => */ Console.println(line))
}
```

Synchronous, Exception-Throwing in IO

```scala
def getStrLn: IO[IOException, String] = {
  // Side-effecting code reads from keyboard until a line is available,
  // and returns the line (String).

  // In case an IOException is thrown, catch it
  // and fail with the exception (not thrown)
  // or die in case of any other exception.
  IO.effect(/* () => */ StdIn.readLine()) /* IO[Throwable, String] */
    .refineOrDie {
      case e: IOException => e
    }
}
```

Asynchronous, Callback-Based in `IO`

```
def addAsync(a: Int, b: Int,
  onSuccess: Int => Unit,
  onFailure: String => Unit
): Unit = ???

def add(a: Int, b: Int): IO[String, Int] = {
  IO.effectAsync { (notify: IO[String, Int] => Unit) =>
    addAsync(a, b,
      result => notify(IO.succeed(result)),
      error => notify(IO.fail(error))
    )
  }
}
```

Interruptible, Asynchronous, Callback-Based in `IO`

```scala
def addAsync(a: Int, b: Int,
  onSuccess: Int => Unit,
  onFailure: String => Unit
): () => Unit = ???

def add(a: Int, b: Int): IO[String, Int] = {
  IO.effectAsyncInterrupt { (notify: IO[String, Int] => Unit) =>
    val canceler = addAsync(a, b,
      result => notify(IO.succeed(result)),
      error => notify(IO.fail(error))
    )

    Left(IO.effectTotal(/* () => */ canceler()))
  }
}
```

Asynchronous `Future` in `IO`

```scala
def addAsync(a: Int, b: Int)(implicit ec: ExecutionContext): Future[Int] = ???

def add(a: Int, b: Int): IO[Throwable, Int] = {
  IO.fromFuture { implicit ec =>
    addAsync(a, b) /* (ec) */
  }
}
```

Combining `IO`s

Transforming `IO` (`map`)

```scala
val randomLetter: IO[Nothing, Char] =
  randomBetween('A', 'Z') /* IO[Nothing, Int] */
    .map { i /* Int */ =>
      i.toChar /* Char */
    }
```

Chaining IOs (broken map)

```scala
val printRolledDiceWRONG: IO[Nothing, IO[Nothing, Unit]] =
  randomBetween(1, 6) /* IO[Nothing, Int] */
    .map { dice /* Int */ =>
      putStrLn(s"Dice shows $dice") /* IO[Nothing, Unit] */
    }
```

- Wrong **nested** type `IO[Nothing, IO[Nothing, Unit]]`
- Needs to be made **flat** somehow as `IO[Nothing, Unit]`

Chaining IOs (flatMap)

```scala
val printRolledDice: IO[Nothing, Unit] =
  randomBetween(1, 6) /* IO[Nothing, Int] */
    .flatMap { dice /* Int */ =>
      putStrLn(s"Dice shows $dice") /* IO[Nothing, Unit] */
    }
```

Chaining and Transforming `IO`s

```
randomBetween(0, 20).flatMap { x =>
  randomBetween(0, 20).map { y =>
    Point(x, y)
  }
}
```

Pyramid of `maps` and `flatMaps` 😈

```
val welcomeNewPlayer: IO[IOException, Unit] =
  putStrLn("What's your name?").flatMap { _ =>
    getStrLn.flatMap { name =>
      randomBetween(0, 20).flatMap { x =>
        randomBetween(0, 20).flatMap { y =>
          randomBetween(0, 20).flatMap { z =>
            putStrLn(s"Welcome $name, you start at coordinates($x, $y, $z).")
          }
        }
      }
    }
  }
```

Flatten Them All 😇

```scala
val welcomeNewPlayer: IO[IOException, Unit] =
  for {
    _ <- putStrLn("What's your name?")
    name <- getStrLn
    x <- randomBetween(0, 20)
    y <- randomBetween(0, 20)
    z <- randomBetween(0, 20)
    _ <- putStrLn(s"Welcome $name, you start at coordinates($x, $y, $z).")
  } yield ()
```

Intermediary Variable

```scala
val printRandomPoint: IO[Nothing, Unit] =
  for {
    x <- randomBetween(0, 20)
    y <- randomBetween(0, 20)
    point = Point(x, y) // Not running an IO, '=' instead of '<-'
    _ <- putStrLn(s"point=$point")
  } yield ()
```

Anatomy of `for` Comprehension

`for` **comprehension is not a** `for` **loop**.

It can be a `for` loop...

But it can handle **many other things**

like `IO` and ... `Seq`, `Option`, `Future`...

for Comprehension **Types**

```scala
val printRandomPoint: IO[Nothing, Point] = {
  for {
    x     /* Int   */ <- randomBetween(0, 10)                /* IO[Nothing, Int]  */
    _     /* Unit  */ <- putStrLn(s"x=$x")                   /* IO[Nothing, Unit] */
    y     /* Int   */ <- randomBetween(0, 10)                /* IO[Nothing, Int]  */
    _     /* Unit  */ <- putStrLn(s"y=$y")                   /* IO[Nothing, Unit] */
    point /* Point */ =  Point(x, y)                         /* Point             */
    _     /* Unit  */ <- putStrLn(s"point.x=${point.x}")     /* IO[Nothing, Unit] */
    _     /* Unit  */ <- putStrLn(s"point.y=${point.y}")     /* IO[Nothing, Unit] */
  } yield point /* Point */
} /* IO[Nothing, Point] */
```

`for` Comprehension **Type Rules**

	`val` **type**	**operator**	**expression type**
generator	`A`	`<-`	`IO[E, A]`
assignment	`B`	`=`	`B`

	`for` **comprehension type**	`yield` **expression type**
production	`IO[E, R]`	`R`

- Combines **only `IO[E, T]`**, **no mix** with `Seq[T]`, `Option[T]`, `Future[T]`…
- But it could be **only** `Seq[T]`, **only** `Option[T]`, **only** `Future[T]`…

for Comprehension **Scopes**

```
val printRandomPoint: IO[Nothing, Point] = {
  for {
    x <- randomBetween(0, 10)                  /*  x                  */
    _ <- putStrLn(s"x=$x")                     /*  O                  */
    y <- randomBetween(0, 10)                  /*  |    y             */
    _ <- putStrLn(s"y=$y")                     /*  |    O             */
    point = Point(x, y)                        /*  O    O    point    */
    _ <- putStrLn(s"point.x=${point.x}")       /*  |    |    O        */
    _ <- putStrLn(s"point.y=${point.y}")       /*  |    |    O        */
  } yield point                                /*  |    |    O        */
}
```

`for` Comprehension **Implicit Nesting**

```
val printRandomPoint: IO[Nothing, Point] = {
  for {
         x <- randomBetween(0, 10)
    /* | */ _ <- putStrLn(s"x=$x")
    /* |    | */ y <- randomBetween(0, 10)
    /* |    |    | */ _ <- putStrLn(s"y=$y")
    /* |    |    |    | */ point = Point(x, y)
    /* |    |    |    |    | */ _ <- putStrLn(s"point.x=${point.x}")
    /* |    |    |    |    |    | */ _ <- putStrLn(s"point.y=${point.y}")
  } /* |    |    |    |    |    |    | */ yield point
}
```

Conditions and Loops

Behaving Conditionally

```scala
def describeNumber(n: Int): IO[Nothing, Unit] = {
  for {
    _ <- if (n % 2 == 0) putStrLn("Even") else putStrLn("Odd")
    _ <- if (n == 42) putStrLn("The Answer") else IO.unit
  } yield ()
}
```

Looping with **Recursion** 😨

```
def findName(id: Int): IO[Nothing, String] = ???

def findNames(ids: List[Int]): IO[Nothing, List[String]] = {
  ids match {
    case Nil => IO.succeed(Nil)

    case id :: restIds =>
      for {
        name      /* String       */ <- findName(id)       /* IO[Nothing, String]       */
        restNames /* List[String] */ <- findNames(restIds) /* IO[Nothing, List[String]] */
      } yield name :: restNames /* List[String] */
  }
}
```

Looping with `foreach`

```scala
def findName(id: Int): IO[Nothing, String] = ???

def findNames(ids: List[Int]): IO[Nothing, List[String]] =
  IO.foreach(ids) { id => findName(id) }
```

- Recursion can be hard to read
- Prefer using simpler alternatives whenever possible
  - `IO.foreach`, `IO.collectAll`, `IO.reduceAll`, `IO.mergeAll`
  - Or `IO.foreachPar`, `IO.collectAllPar`, `IO.reduceAllPar`, `IO.mergeAllPar` in **parallel** 👍

Further with *ZIO*

Keeping Resource Safe

```scala
class Resource {
  def close: IO[Nothing, Unit] = ???
  def read: IO[Int, String] = ???
}

object Resource {
  def open(name: String): IO[Int, Resource] = ???
}

val program: IO[Int, Unit] =
  IO.bracket(Resource.open("hello"))(_.close) { resource =>
    for {
      line <- resource.read
      _ <- putStrLn(line)
    } yield ()
  }
```

Retrying after Error

```scala
def findName(id: Int): IO[Int, String] = ???

val retrySchedule = Schedule.recurs(5) && Schedule.exponential(1.second)

val program: IO[Int, Unit] =
  for {
    name <- findName(1).retry(retrySchedule)
    _ <- putStrLn(s"name=$name")
  } yield ()
```

- `retry` repeats in case of **failure**.
- There also exists `repeat` that repeats in case of **success**.

Forking and Interrupting

```
val analyze: IO[Nothing, String] = ???
val validate: IO[Nothing, Boolean] = ???

val program: IO[Nothing, String] =
  for {
    analyzeFiber <- analyze.fork
    validateFiber <- validate.fork
    validated <- validateFiber.join
    _ <- if (validated) IO.unit else analyzeFiber.interrupt
    analysis <- analyzeFiber.join
    _ <- putStrLn(analysis)
  } yield analysis
```

Injecting Services with Environment

```
ZIO[-R, +E, +A] // R = Environment, E = Error, A = Result
```

- `R` is the type for the **environment** required to run the program.
- A set of **services** expressed as a *compound type* (using `with`)
- `Any` means that *any* environment is enough, so it requires *no* environment.

And `IO` is just a type alias

```
type IO[+E, +A] = ZIO[Any, E, A]
```

Using Standard Services

```scala
val program: ZIO[System with Clock with Random with Console, Throwable, Unit] = for {
  randomNumber <- random.nextInt(10)
  maybeJavaVersion <- system.property("java.version")
  millisSinceEpoch <- clock.currentTime(TimeUnit.MILLISECONDS)

  _ <- ZIO.foreach(maybeJavaVersion) { javaVersion =>
    console.putStrLn(s"Java Version is $javaVersion")
  }

  _ <- console.putStrLn(s"Milliseconds since epoch is $millisSinceEpoch")
  _ <- console.putStrLn(s"Random number is $randomNumber")
} yield ()
```

Powerful Testing and Debugging

- **Full Testability**
  - **Dependency injection** of services in environment
- **Lossless Error Traceability**
  - No error is lost
  - Concurrent errors are kept
  - No need to mindlessly log exceptions
- **Full Stack Traces**
  - Across *fibers* and *threads*

Loaded with Features

- **Streaming**
  - `Stream`, a lazy, concurrent, asynchronous source of values
  - `Sink`, a consumer of values, which may produces a value when it has consumed enough
- **Software Transactional Memory** (STM)
- **Low Level Concurrency**
  - `FiberLocal`, `FiberRef`, a variable whose value depends on the fiber that accesses it
  - `Promise`, a variable that may be set a single time, and awaited on by many fibers
  - `Queue`, an asynchronous queue that never blocks
  - `Ref`, a mutable reference to a value
  - `Semaphore`, a semaphore