Implementing the IO Monad

https://github.com/hermannhueck/implementing-io-monad

Abstract

My implementation of the IO Monad is just a feasibility study, not production code.

When coding this IO impl I was very much inspired by *cats.effect.IO* and *monix.eval.Task* which I studied at that time. Both are implementions of the IO Monad.

The API of my IO is very similar to the basics of Monix *Task*. This IO implementation also helped me to understand the IO Monad (of *cats-effect*) and Monix *Task*.

Interop with *Future* is also supported. You can convert *IO* to a *Future*. Vice versa you can convert a *Future* to an *IO*.

The development of my impl can be followed step by step in the files in package *iomonad*.

Agenda

1. Referential Transparency
2. Is Future referentially transparent?
3. The IO Monad
4. Resources

1. Referential Transparency

Referential Transparency

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

This requires that the expression is ***pure***, that is to say the expression value must be the same for the same inputs and its evaluation must have ***no side effects***.

https://en.wikipedia.org/wiki/Referential_transparency

Referential Transparency Benefits

- (Equational) Reasoning about code
- Refactoring is easier
- Testing is easier
- Separate pure code from impure code
- Potential compiler optimizations (more in Haskell than in Scala)
  (e.g. memoization, parallelisation, compute expressions at compile time)

"What Referential Transparency can do for you"
Talk by Luka Jacobowitz at ScalaIO 2017
https://www.youtube.com/watch?v=X-cEGEJMx_4

func is not referentially transparent!

```
def func(ioa1: Unit, ioa2: Unit): Unit = {
  ioa1
  ioa2
}

func(println("hi"), println("hi"))        // prints "hi" twice
//=> hi
//=> hi

println("-----")

val x: Unit = println("hi")
func(x, x)                                // prints "hi" once
//=> hi
```

func is referentially transparent!

```
def putStrLn(line: String): IO[Unit] =
IO.eval { println(line) }

def func(ioa1: IO[Unit], ioa2: IO[Unit]): IO[Unit] =
  for {
    _ <- ioa1
    _ <- ioa2
  } yield ()

func(putStrLn("hi"), putStrLn("hi")).run()   // prints "hi" twice
//=> hi
//=> hi

println("-----")

val x: IO[Unit] = putStrLn("hi")
func(x, x).run()                             // prints "hi" twice
//=> hi
//=> hi
```

2. Is *Future* referentially transparent?

Is *Future* referentially transparent?

```scala
val future1: Future[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val future: Future[Int] = Future { atomicInt.incrementAndGet }
  for {
    x <- future
    y <- future
  } yield (x, y)
}

future1 onComplete println     // Success((1,1))
```

```scala
// same as future1, but inlined
val future2: Future[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- Future { atomicInt.incrementAndGet }
    y <- Future { atomicInt.incrementAndGet }
  } yield (x, y)
}

future2 onComplete println     // Success((1,2))    <-- not the same result
```

Is *Future* referentially transparent?

```
val future1: Future[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val future: Future[Int] = Future { atomicInt.incrementAndGet }
  for {
    x <- future
    y <- future
  } yield (x, y)
}

future1 onComplete println     // Success((1,1))
```

```
// same as future1, but inlined
val future2: Future[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- Future { atomicInt.incrementAndGet }
    y <- Future { atomicInt.incrementAndGet }
  } yield (x, y)
}

future2 onComplete println     // Success((1,2))    <-- not the same result
```

No!

Is Monix *Task* referentially transparent?

```scala
val task1: Task[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val task: Task[Int] = Task { atomicInt.incrementAndGet }
  for {
    x <- task
    y <- task
  } yield (x, y)
}

task1 runAsync println     // Success((1,2))
```

```scala
// same as task1, but inlined
val task2: Task[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- Task { atomicInt.incrementAndGet }
    y <- Task { atomicInt.incrementAndGet }
  } yield (x, y)
}

task2 runAsync println     // Success((1,2))    <-- same result
```

Is Monix *Task* referentially transparent?

```
val task1: Task[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val task: Task[Int] = Task { atomicInt.incrementAndGet }
  for {
    x <- task
    y <- task
  } yield (x, y)
}

task1 runAsync println     // Success((1,2))
```

```
// same as task1, but inlined
val task2: Task[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- Task { atomicInt.incrementAndGet }
    y <- Task { atomicInt.incrementAndGet }
  } yield (x, y)
}

task2 runAsync println     // Success((1,2))    <-- same result
```

Yes!

Is my IO Monad referentially transparent?

```scala
val io1: IO[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val io: IO[Int] = IO { atomicInt.incrementAndGet }
  for {
    x <- io
    y <- io
  } yield (x, y)
}

io1.runToFuture onComplete println     // Success((1,2))
```

```scala
// same as io1, but inlined
val io2: IO[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- IO { atomicInt.incrementAndGet }
    y <- IO { atomicInt.incrementAndGet }
  } yield (x, y)
}

io2.runToFuture onComplete println     // Success((1,2))    <-- same result
```

Is my IO Monad referentially transparent?

```scala
val io1: IO[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  val io: IO[Int] = IO { atomicInt.incrementAndGet }
  for {
    x <- io
    y <- io
  } yield (x, y)
}

io1.runToFuture onComplete println     // Success((1,2))
```

```scala
// same as io1, but inlined
val io2: IO[(Int, Int)] = {
  val atomicInt = new AtomicInteger(0)
  for {
    x <- IO { atomicInt.incrementAndGet }
    y <- IO { atomicInt.incrementAndGet }
  } yield (x, y)
}

io2.runToFuture onComplete println     // Success((1,2))    <-- same result
```

Yes!

3. The IO Monad

1. Impure IO Program with side effects

```scala
// impure program
def program(): Unit = {
  print("Welcome to Scala!  What's your name?   ")
  val name = scala.io.StdIn.readLine
  println(s"Well hello, $name!")
}
```

1. Impure IO Program with side effects

```scala
// impure program
def program(): Unit = {
  print("Welcome to Scala!  What's your name?   ")
  val name = scala.io.StdIn.readLine
  println(s"Well hello, $name!")
}
```

```scala
program()
```

1. Impure IO Program <u>with</u> side effects

```scala
// impure program
def program(): Unit = {
  print("Welcome to Scala!  What's your name?   ")
  val name = scala.io.StdIn.readLine
  println(s"Well hello, $name!")
}
```

```scala
program()
```

- Whenever a method or a function returns *Unit* it is ***<u>impure</u>*** (or it is a noop). It's intension is to produce a side effect.
- A ***<u>pure</u>*** function always returns a value of some type
  (and doesn't produce a side effect inside).

2. Pure IO Program <u>without</u> side effects

```scala
// pure program
val program: () => Unit =  // () => Unit  is syntactic sugar for:  Function0[Unit]
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
```

2. Pure IO Program <u>without</u> side effects

```scala
// pure program
val program: () => Unit =  // () => Unit  is syntactic sugar for:  Function0[Unit]
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
```

```scala
program()    // producing the side effects "at the end of the world"
```

2. Pure IO Program <u>without</u> side effects

```
// pure program
val program: () => Unit =  // () => Unit  is syntactic sugar for:  Function0[Unit]
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
```

```
program()    // producing the side effects "at the end of the world"
```

- Make the program a function returning *Unit*: *Function0[Unit]*
- Free of side effects in it's definition
- Produces side effects only when run (at the end of the world)
- *program* can be a <u>val</u> (if no parameters are passed to it)

3. Wrap Function0[A] in a case class

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

3. Wrap Function0[A] in a case class

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

```scala
// pure program
val program: IO[Unit] = IO {
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
}
```

3. Wrap Function0[A] in a case class

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

```
// pure program
val program: IO[Unit] = IO {
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
}
```

```
program.run()    // producing the side effects "at the end of the world"
```

3. Wrap Function0[A] in a case class

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

```scala
// pure program
val program: IO[Unit] = IO {
  () => {
    print("Welcome to Scala!  What's your name?   ")
    val name = scala.io.StdIn.readLine
    println(s"Well hello, $name!")
  }
}
```

```scala
program.run()    // producing the side effects "at the end of the world"
```

- *IO[A]* wraps a *Function0[A]* in a case class.
- This is useful to implement further extensions on that case class.

4. *IO#map* and *IO#flatMap*

```
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
```

4. *IO#map* and *IO#flatMap*

```scala
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
```

```scala
val program: IO[Unit] = for {
  _       <- IO { () => print(s"Welcome to Scala!  What's your name?   ") }
  name    <- IO { () => scala.io.StdIn.readLine }
  _       <- IO { () => println(s"Well hello, $name!") }
} yield ()
```

4. *IO#map* and *IO#flatMap*

```scala
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
```

```scala
val program: IO[Unit] = for {
  _       <- IO { () => print(s"Welcome to Scala!  What's your name?   ") }
  name    <- IO { () => scala.io.StdIn.readLine }
  _       <- IO { () => println(s"Well hello, $name!") }
} yield ()
```

```scala
program.run()    // producing the side effects "at the end of the world"
```

4. *IO#map* and *IO#flatMap*

```scala
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
```

```scala
val program: IO[Unit] = for {
  _       <- IO { () => print(s"Welcome to Scala!  What's your name?   ") }
  name    <- IO { () => scala.io.StdIn.readLine }
  _       <- IO { () => println(s"Well hello, $name!") }
} yield ()
```

```scala
program.run()    // producing the side effects "at the end of the world"
```

- With *map* and *flatMap* *IO[A]* is monadic (but it is not yet a Monad).
- *IO* is ready for for-comprehensions.
- This allows the composition of programs from smaller components.

5. Companion object *IO* with *pure* and *eval*

```
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
object IO {
  def pure[A](value: A): IO[A] = IO { () => value }         // eager
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }      // lazy
}
```

5. Companion object *IO* with *pure* and *eval*

```scala
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
object IO {
  def pure[A](value: A): IO[A] = IO { () => value }         // eager
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }      // lazy
}
```

```scala
val program: IO[Unit] = for {
  welcome <- IO.pure("Welcome to Scala!")
  _       <- IO.eval { print(s"$welcome  What's your name?   ") }
  name    <- IO.eval { scala.io.StdIn.readLine }         // simpler with IO.eval
  _       <- IO.eval { println(s"Well hello, $name!") }
} yield ()
```

5. Companion object *IO* with *pure* and *eval*

```
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
object IO {
  def pure[A](value: A): IO[A] = IO { () => value }         // eager
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }      // lazy
}
```

```
val program: IO[Unit] = for {
  welcome <- IO.pure("Welcome to Scala!")
  _       <- IO.eval { print(s"$welcome  What's your name?   ") }
  name    <- IO.eval { scala.io.StdIn.readLine }         // simpler with IO.eval
  _       <- IO.eval { println(s"Well hello, $name!") }
} yield ()
```

```
program.run()    // producing the side effects "at the end of the world"
```

5. Companion object *IO* with *pure* and *eval*

```scala
case class IO[A](run: () => A) {
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
  def flatten[B](implicit ev: A <:< IO[B]): IO[B] = flatMap(a => a)
}
object IO {
  def pure[A](value: A): IO[A] = IO { () => value }         // eager
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }      // lazy
}
```

```scala
val program: IO[Unit] = for {
  welcome <- IO.pure("Welcome to Scala!")
  _       <- IO.eval { print(s"$welcome  What's your name?   ") }
  name    <- IO.eval { scala.io.StdIn.readLine }         // simpler with IO.eval
  _       <- IO.eval { println(s"Well hello, $name!") }
} yield ()
```

```scala
program.run()    // producing the side effects "at the end of the world"
```

- *IO.pure* is eager and accepts a pure value.
- *IO.eval* is lazy and accepts a computation.
- *map* can be written in terms of *flatMap* and *pure*.

6. ADT *IO*

```scala
sealed trait IO[+A] {
  def run(): A

  def flatMap[B](f: A => IO[B]): IO[B] = IO { f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
}
object IO {
  // ADT sub types
  private case class Pure[A](thunk: () => A) extends IO[A] {
    override def run(): A = thunk()
  }
  private case class Eval[A](thunk: () => A) extends IO[A] {
    override def run(): A = thunk()
  }

  def pure[A](a: A): IO[A] = Pure { () => a }
  def now[A](a: A): IO[A] = pure(a)

  def eval[A](a: => A): IO[A] = Eval { () => a }
  def delay[A](a: => A): IO[A] = eval(a)
  def apply[A](a: => A): IO[A] = eval(a)
}
```

6. ADT *IO*

```scala
sealed trait IO[+A] {
  def run(): A

  def flatMap[B](f: A => IO[B]): IO[B] = IO { f(run()).run() }
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
}
object IO {
  // ADT sub types
  private case class Pure[A](thunk: () => A) extends IO[A] {
    override def run(): A = thunk()
  }
  private case class Eval[A](thunk: () => A) extends IO[A] {
    override def run(): A = thunk()
  }

  def pure[A](a: A): IO[A] = Pure { () => a }
  def now[A](a: A): IO[A] = pure(a)

  def eval[A](a: => A): IO[A] = Eval { () => a }
  def delay[A](a: => A): IO[A] = eval(a)
  def apply[A](a: => A): IO[A] = eval(a)
}
```

- *IO.apply* is an alias for *IO.eval*. (Simplifies the creation of *IO* instances.)
- The app works as before.

7. ADT sub type *FlatMap*

```scala
sealed trait IO[+A] {

  def flatMap[B](f: A => IO[B]): IO[B] = FlatMap(this, f)
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
}

object IO {

  // ADT sub types
  private case class FlatMap[A, B](src: IO[A], f: A => IO[B]) extends IO[B] {
    override def run(): B = f(src.run()).run()
  }
}
```

7. ADT sub type *FlatMap*

```scala
sealed trait IO[+A] {

  def flatMap[B](f: A => IO[B]): IO[B] = FlatMap(this, f)
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
}

object IO {

  // ADT sub types
  private case class FlatMap[A, B](src: IO[A], f: A => IO[B]) extends IO[B] {
    override def run(): B = f(src.run()).run()
  }
}
```

- We gained stack-safety by trampolining. (*FlatMap* is a heap object.)
- The app works as before.

IO Monad - Basic Setup

- A *Function0* wrapped in a case class
- Impl of *map* and *flatMap* (and flatten)
- companion object with *pure* and other smart constructors
- case class converted to ADT

IO Monad - Basic Setup

- A *Function0* wrapped in a case class
- Impl of *map* and *flatMap* (and flatten)
- companion object with *pure* and other smart constructors
- case class converted to ADT

What is to come?

- Sync and async run methods
- More smart constructors
- Monad instance for *IO*
- Create *IO* from *Try*, *Either*, *Future*
- *MonadError* instance for *IO* (*IO.raiseError*, *IO#handleErrorWith*)
- *Bracket* instance for *IO* (*IO#bracket*)
- *Sync* instance for *IO* (*IO.delay*)

8. Sync run* methods

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

  def runToTry: Try[A] = Try { run() }

  def runToEither: Either[Throwable, A] = runToTry.toEither
}
```

8. Sync run* methods

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

  def runToTry: Try[A] = Try { run() }

  def runToEither: Either[Throwable, A] = runToTry.toEither
}
```

```
// running the program synchronously ...

val value: Unit = program.run()                                 // ()

val tryy: Try[Unit] = program.runToTry                          // Success(())

val either: Either[Throwable, Unit] = program.runToEither       // Right(())
```

8. Sync run* methods

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

  def runToTry: Try[A] = Try { run() }

  def runToEither: Either[Throwable, A] = runToTry.toEither
}
```

```scala
// running the program synchronously ...

val value: Unit = program.run()                               // ()

val tryy: Try[Unit] = program.runToTry                        // Success(())

val either: Either[Throwable, Unit] = program.runToEither     // Right(())
```

- *run* may throw an exception.
- *runToTry* and *runToEither* avoid that.

9. Other example: Authenticate Maggie

```scala
def authenticate(username: String, password: String): IO[Boolean] = ???

val checkMaggie: IO[Boolean] = authenticate("maggie", "maggie-pw")


// running checkMaggie synchronously ...

val value: Boolean = checkMaggie.run()
// true

val tryy: Try[Boolean] = checkMaggie.runToTry
// Success(true)

val either: Either[Throwable, Boolean] = checkMaggie.runToEither
// Right(true)
```

9. Other example: Authenticate Maggie

```scala
def authenticate(username: String, password: String): IO[Boolean] = ???

val checkMaggie: IO[Boolean] = authenticate("maggie", "maggie-pw")


// running checkMaggie synchronously ...

val value: Boolean = checkMaggie.run()
// true

val tryy: Try[Boolean] = checkMaggie.runToTry
// Success(true)

val either: Either[Throwable, Boolean] = checkMaggie.runToEither
// Right(true)
```

- The previous example was interactive … not well suited for async IO.

10. Async run* methods

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

  def runToFuture(implicit ec: ExecutionContext): Future[A] =
    Future { run() }

  def runOnComplete(callback: Try[A] => Unit)
                                     (implicit ec: ExecutionContext): Unit =
    runToFuture onComplete callback

  def runAsync(callback: Either[Throwable, A] => Unit)
                                     (implicit ec: ExecutionContext): Unit =
    runOnComplete(tryy => callback(tryy.toEither))
}
```

10. Async run* methods

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

  def runToFuture(implicit ec: ExecutionContext): Future[A] =
    Future { run() }

  def runOnComplete(callback: Try[A] => Unit)
                                    (implicit ec: ExecutionContext): Unit =
    runToFuture onComplete callback

  def runAsync(callback: Either[Throwable, A] => Unit)
                                    (implicit ec: ExecutionContext): Unit =
    runOnComplete(tryy => callback(tryy.toEither))
}
```

- All async run* methods take an implicit EC.
- Unlike *Future* the EC is not needed to create an *IO*, only to run it.

Using the async run* methods

```scala
def authenticate(username: String, password: String): IO[Boolean] = ???

val checkMaggie: IO[Boolean] = authenticate("maggie", "maggie-pw")
```

Using the async run* methods

```
def authenticate(username: String, password: String): IO[Boolean] = ???

val checkMaggie: IO[Boolean] = authenticate("maggie", "maggie-pw")
```

```
// running 'checkMaggie' asynchronously ...

implicit val ec: ExecutionContext = ExecutionContext.global

val future: Future[Boolean] = checkMaggie.runToFuture
future onComplete tryCallback                           //=> true

checkMaggie runOnComplete tryCallback                   //=> true

checkMaggie runAsync eitherCallback                     //=> true
```

Using the async run* methods

```scala
def authenticate(username: String, password: String): IO[Boolean] = ???

val checkMaggie: IO[Boolean] = authenticate("maggie", "maggie-pw")
```

```scala
// running 'checkMaggie' asynchronously ...

implicit val ec: ExecutionContext = ExecutionContext.global

val future: Future[Boolean] = checkMaggie.runToFuture
future onComplete tryCallback                          //=> true

checkMaggie runOnComplete tryCallback                  //=> true

checkMaggie runAsync eitherCallback                    //=> true
```

```scala
def tryCallback[A]: Try[A] => Unit = tryy =>
  println(tryy.fold(ex => ex.toString, value => value.toString))

def eitherCallback[A]: Either[Throwable, A] => Unit = either =>
  println(either.fold(ex => ex.toString, value => value.toString))
```

11. Async method *foreach*

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

  def foreach(f: A => Unit)(implicit ec: ExecutionContext): Unit =
    runAsync {
      case Left(ex) => ec.reportFailure(ex)
      case Right(value) => f(value)
    }
}
```

11. Async method *foreach*

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

  def foreach(f: A => Unit)(implicit ec: ExecutionContext): Unit =
    runAsync {
      case Left(ex) => ec.reportFailure(ex)
      case Right(value) => f(value)
    }
}
```

```
authenticate("maggie", "maggie-pw") foreach println       //=> true
authenticate("maggieXXX", "maggie-pw") foreach println    //=> false
authenticate("maggie", "maggie-pwXXX") foreach println    //=> false
```

11. Async method *foreach*

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

  def foreach(f: A => Unit)(implicit ec: ExecutionContext): Unit =
    runAsync {
      case Left(ex) => ec.reportFailure(ex)
      case Right(value) => f(value)
    }
}
```

```
authenticate("maggie", "maggie-pw") foreach println       //=> true
authenticate("maggieXXX", "maggie-pw") foreach println    //=> false
authenticate("maggie", "maggie-pwXXX") foreach println    //=> false
```

- *foreach* runs asynchronously.
- It takes a callback for the successful result value.
- *foreach* swallows exceptions. Prefer *runAsync*!

12. *IO.raiseError*

```
object IO {
  // ADT sub types
  private case class Error[A](exception: Throwable) extends IO[A] {
    override def run(): A = throw exception
  }

  def raiseError[A](exception: Exception): IO[A] = Error[A](exception)
}
```

12. *IO.raiseError*

```scala
object IO {
  // ADT sub types
  private case class Error[A](exception: Throwable) extends IO[A] {
    override def run(): A = throw exception
  }

  def raiseError[A](exception: Exception): IO[A] = Error[A](exception)
}
```

- ADT sub type *Error* wraps a *Throwable*.

12. *IO.raiseError*

```scala
object IO {
  // ADT sub types
  private case class Error[A](exception: Throwable) extends IO[A] {
    override def run(): A = throw exception
  }

  def raiseError[A](exception: Exception): IO[A] = Error[A](exception)
}
```

- ADT sub type *Error* wraps a *Throwable*.

```scala
val ioError: IO[Int] = IO.raiseError[Int](
                                new IllegalStateException("illegal state"))
println(ioError.runToEither)
//=> Left(java.lang.IllegalStateException: illegal state)
```

13. *IO#failed* (returns a failed projection).

```scala
sealed trait IO[+A] {
  def failed: IO[Throwable] = Failed(this)
}

object IO {
  // ADT sub types
  private case class Failed[A](io: IO[A]) extends IO[Throwable] {
    override def run(): Throwable = try {
      io.run()
      throw new NoSuchElementException("failed")
    } catch {
      case nse: NoSuchElementException if nse.getMessage == "failed" =>
        throw nse
      case throwable: Throwable =>
        throwable
    }
  }
}
```

13. *IO#failed* (returns a failed projection).

```
sealed trait IO[+A] {
  def failed: IO[Throwable] = Failed(this)
}

object IO {
  // ADT sub types
  private case class Failed[A](io: IO[A]) extends IO[Throwable] {
    override def run(): Throwable = try {
      io.run()
      throw new NoSuchElementException("failed")
    } catch {
      case nse: NoSuchElementException if nse.getMessage == "failed" =>
        throw nse
      case throwable: Throwable =>
        throwable
    }
  }
}
```

- ADT sub type *Failed* wraps the *IO* to project.

Using *IO#failed*

```
val ioError: IO[Int] = IO.raiseError[Int](
                                new IllegalStateException("illegal state"))
println(ioError.runToEither)
//=> Left(java.lang.IllegalStateException: illegal state)

val failed: IO[Throwable] = ioError.failed
println(failed.runToEither)
//=> Right(java.lang.IllegalStateException: illegal state)

val ioSuccess = IO.pure(5)
println(ioSuccess.runToEither)
//=> Right(5)

println(ioSuccess.failed.runToEither)
//=> Left(java.util.NoSuchElementException: failed)
```

Using *IO#failed*

```scala
val ioError: IO[Int] = IO.raiseError[Int](
                                new IllegalStateException("illegal state"))
println(ioError.runToEither)
//=> Left(java.lang.IllegalStateException: illegal state)

val failed: IO[Throwable] = ioError.failed
println(failed.runToEither)
//=> Right(java.lang.IllegalStateException: illegal state)

val ioSuccess = IO.pure(5)
println(ioSuccess.runToEither)
//=> Right(5)

println(ioSuccess.failed.runToEither)
//=> Left(java.util.NoSuchElementException: failed)
```

The failed projection is an *IO* holding a value of type *Throwable*, emitting the error yielded by the source, in case the source fails, otherwise if the source succeeds the result will fail with a *NoSuchElementException*.

14. Other example: pure computations

```scala
def sumIO(from: Int, to: Int): IO[Int] =
  IO { sumOfRange(from, to) }

def fibonacciIO(num: Int): IO[BigInt] =
  IO { fibonacci(num) }

def factorialIO(num: Int): IO[BigInt] =
  IO { factorial(num) }

def computeIO(from: Int, to: Int): IO[BigInt] =
  for {
    x <- sumIO(from, to)
    y <- fibonacciIO(x)
    z <- factorialIO(y.intValue)
  } yield z


val io: IO[BigInt] = computeIO(1, 4)

implicit val ec: ExecutionContext = ExecutionContext.global
io foreach { result => println(s"result = $result") }
//=> 6227020800
```

15. Monad instance for *IO*

```scala
sealed trait IO[+A] {

  def flatMap[B](f: A => IO[B]): IO[B] = FlatMap(this, f)
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
}

object IO {

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

  implicit val ioMonad: Monad[IO] = new Monad[IO] {
    override def pure[A](value: A): IO[A] = IO.pure(value)
    override def flatMap[A, B](fa: IO[A])(f: A => IO[B]): IO[B] = fa flatMap f
  }
}
```

15. Monad instance for *IO*

```scala
sealed trait IO[+A] {

  def flatMap[B](f: A => IO[B]): IO[B] = FlatMap(this, f)
  def map[B](f: A => B): IO[B] = flatMap(a => pure(f(a)))
}

object IO {

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

  implicit val ioMonad: Monad[IO] = new Monad[IO] {
    override def pure[A](value: A): IO[A] = IO.pure(value)
    override def flatMap[A, B](fa: IO[A])(f: A => IO[B]): IO[B] = fa flatMap f
  }
}
```

- Monad instance defined in companion object (implicit scope)

Computations that abstract over HKT: *F[_]: Monad*

```scala
import scala.language.higherKinds
import cats.syntax.flatMap._
import cats.syntax.functor._

def sumF[F[_]: Monad](from: Int, to: Int): F[Int] =
  Monad[F].pure { sumOfRange(from, to) }

def fibonacciF[F[_]: Monad](num: Int): F[BigInt] =
  Monad[F].pure { fibonacci(num) }

def factorialF[F[_]: Monad](num: Int): F[BigInt] =
  Monad[F].pure { factorial(num) }

def computeF[F[_]: Monad](from: Int, to: Int): F[BigInt] =
  for {
    x <- sumF(from, to)
    y <- fibonacciF(x)
    z <- factorialF(y.intValue)
  } yield z
```

- This code can be used with *IO* or any other Monad.

- **Reify *F[_] : Monad* with *IO***

```scala
import scala.concurrent.ExecutionContext.Implicits.global

val io: IO[BigInt] = computeF[IO](1, 4)
io foreach { result => println(s"result = $result") }          //=> 6227020800
```

- **Reify *F[_] : Monad* with *IO***

```
import scala.concurrent.ExecutionContext.Implicits.global

val io: IO[BigInt] = computeF[IO](1, 4)
io foreach { result => println(s"result = $result") }           //=> 6227020800
```

- **Reify *F[_] : Monad* with *cats.Id***

```
val result: cats.Id[BigInt] = computeF[cats.Id](1, 4)
println(s"result = $result")                                     //=> 6227020800
```

- **Reify *F[_] : Monad* with *IO***

```
import scala.concurrent.ExecutionContext.Implicits.global

val io: IO[BigInt] = computeF[IO](1, 4)
io foreach { result => println(s"result = $result") }          //=> 6227020800
```

- **Reify *F[_] : Monad* with *cats.Id***

```
val result: cats.Id[BigInt] = computeF[cats.Id](1, 4)
println(s"result = $result")                                   //=> 6227020800
```

- **Reify *F[_] : Monad* with *Option***

```
import cats.instances.option._

val maybeResult: Option[BigInt] = computeF[Option](1, 4)
maybeResult foreach { result => println(s"result = $result") }  //=> 6227020800
```

- **Reify *F[_] : Monad* with *IO***

```
import scala.concurrent.ExecutionContext.Implicits.global

val io: IO[BigInt] = computeF[IO](1, 4)
io foreach { result => println(s"result = $result") }          //=> 6227020800
```

- **Reify *F[_] : Monad* with *cats.Id***

```
val result: cats.Id[BigInt] = computeF[cats.Id](1, 4)
println(s"result = $result")                                     //=> 6227020800
```

- **Reify *F[_] : Monad* with *Option***

```
import cats.instances.option._

val maybeResult: Option[BigInt] = computeF[Option](1, 4)
maybeResult foreach { result => println(s"result = $result") }  //=> 6227020800
```

- **Reify *F[_] : Monad* with *Future***

```
import scala.concurrent.{Future, ExecutionContext}
import ExecutionContext.Implicits.global
import cats.instances.future._

val future: Future[BigInt] = computeF[Future](1, 4)
future foreach { result => println(s"result = $result") }      //=> 6227020800
```

16. *IO.defer* and *IO.suspend*

```scala
object IO {

  // ADT sub types
  private case class Suspend[A](thunk: () => IO[A]) extends IO[A] {
    override def run(): A = thunk().run()
  }

  def suspend[A](ioa: => IO[A]): IO[A] = Suspend(() => ioa)
  def defer[A](ioa: => IO[A]): IO[A] = suspend(ioa)
}
```

16. *IO.defer* and *IO.suspend*

```scala
object IO {

  // ADT sub types
  private case class Suspend[A](thunk: () => IO[A]) extends IO[A] {
    override def run(): A = thunk().run()
  }

  def suspend[A](ioa: => IO[A]): IO[A] = Suspend(() => ioa)
  def defer[A](ioa: => IO[A]): IO[A] = suspend(ioa)
}
```

- These methods defer the (possibly immediate) side effect of the inner *IO*.
- ADT sub type *Suspend* wraps another *IO*.

Using *IO.defer*

IO.pure(...) // without *IO.defer*

```
val io1 = IO.pure { println("immediate side effect"); 5 }
//=> immediate side effect
Thread sleep 2000L
io1 foreach println
//=> 5
```

Using *IO.defer*

IO.pure(...) // without *IO.defer*

```
val io1 = IO.pure { println("immediate side effect"); 5 }
//=> immediate side effect
Thread sleep 2000L
io1 foreach println
//=> 5
```

IO.defer(IO.pure(...))

```
val io2 = IO.defer { IO.pure { println("deferred side effect"); 5 } }
Thread sleep 2000L
io2 foreach println
//=> deferred side effect
//=> 5
```

17. *IO.fromTry* and *IO.fromEither*

```scala
object IO {

  def fromTry[A](tryy: Try[A]): IO[A] =
    tryy.fold(IO.raiseError, IO.pure)

  def fromEither[A](either: Either[Throwable, A]): IO[A] =
    either.fold(IO.raiseError, IO.pure)
}
```

17. *IO.fromTry* and *IO.fromEither*

```scala
object IO {

  def fromTry[A](tryy: Try[A]): IO[A] =
    tryy.fold(IO.raiseError, IO.pure)

  def fromEither[A](either: Either[Throwable, A]): IO[A] =
    either.fold(IO.raiseError, IO.pure)
}
```

```scala
val tryy: Try[Seq[User]] = Try { User.getUsers }
val io1: IO[Seq[User]] = IO.fromTry(tryy)

val either: Either[Throwable, Seq[User]] = tryy.toEither
val io2: IO[Seq[User]] = IO.fromEither(either)
```

18. *IO.fromFuture*

```
object IO {

  // ADT sub types
  private case class FromFuture[A](fa: Future[A]) extends IO[A] {
    override def run(): A = Await.result(fa, Duration.Inf) // BLOCKING!!!
  }

  def fromFuture[A](future: Future[A]): IO[A] = FromFuture(future)
}
```

18. *IO.fromFuture*

```scala
object IO {

  // ADT sub types
  private case class FromFuture[A](fa: Future[A]) extends IO[A] {
    override def run(): A = Await.result(fa, Duration.Inf) // BLOCKING!!!
  }

  def fromFuture[A](future: Future[A]): IO[A] = FromFuture(future)
}
```

- Attention: The implementation of *fromFuture* is a bit simplistic.
- Waiting for the *Future* to complete might block a thread!
- (A solution of this problem would require a redesign of my simple IO Monad, which doesn't support non-blocking async computations.)

Using *IO.fromFuture*

```scala
def futureGetUsers(implicit ec: ExecutionContext): Future[Seq[User]] =
  Future {
    println("side effect")
    User.getUsers
  }
```

Using *IO.fromFuture*

```scala
def futureGetUsers(implicit ec: ExecutionContext): Future[Seq[User]] =
  Future {
    println("side effect")
    User.getUsers
  }
```

- *IO.fromFuture(f)* is eager.

```scala
implicit val ec: ExecutionContext = ExecutionContext.global

val io = IO.fromFuture { futureGetUsers }

io foreach { users => users foreach println } //=> "side effect"
io foreach { users => users foreach println }
```

Using *IO.fromFuture*

```
def futureGetUsers(implicit ec: ExecutionContext): Future[Seq[User]] =
  Future {
    println("side effect")
    User.getUsers
  }
```

- *IO.fromFuture(f)* is eager.

```
implicit val ec: ExecutionContext = ExecutionContext.global

val io = IO.fromFuture { futureGetUsers }

io foreach { users => users foreach println } //=> "side effect"
io foreach { users => users foreach println }
```

- *IO.defer(IO.fromFuture(f))* is lazy.

```
implicit val ec: ExecutionContext = ExecutionContext.global

val io = IO.defer { IO.fromFuture { futureGetUsers } }

io foreach { users => users foreach println } //=> "side effect"
io foreach { users => users foreach println } //=> "side effect"
```

19. *IO.deferFuture*

```
object IO {

  def deferFuture[A](fa: => Future[A]): IO[A] =
    defer(IO.fromFuture(fa))
}
```

19. *IO.deferFuture*

```scala
object IO {

  def deferFuture[A](fa: => Future[A]): IO[A] =
    defer(IO.fromFuture(fa))
}
```

- *IO.deferFuture(f)* is an alias for *IO.defer(IO.fromFuture(f))*.
- An *ExecutionContext* is still required to create the *Future*!

Using *IO.deferFuture*

```
def futureGetUsers(implicit ec: ExecutionContext): Future[Seq[User]] =
  Future {
    println("side effect")
    User.getUsers
  }
```

Using *IO.deferFuture*

```
def futureGetUsers(implicit ec: ExecutionContext): Future[Seq[User]] =
  Future {
    println("side effect")
    User.getUsers
  }
```

- *IO.defer(IO.fromFuture(f))* is lazy.

```
implicit val ec: ExecutionContext = ExecutionContext.global

val io = IO.defer { IO.fromFuture { futureGetUsers } }

io foreach { users => users foreach println } //=> "side effect"
io foreach { users => users foreach println } //=> "side effect"
```

Using *IO.deferFuture*

```scala
def futureGetUsers(implicit ec: ExecutionContext): Future[Seq[User]] =
  Future {
    println("side effect")
    User.getUsers
  }
```

- *IO.defer(IO.fromFuture(f))* is lazy.

```scala
implicit val ec: ExecutionContext = ExecutionContext.global

val io = IO.defer { IO.fromFuture { futureGetUsers } }

io foreach { users => users foreach println } //=> "side effect"
io foreach { users => users foreach println } //=> "side effect"
```

- *IO.deferFuture(f)* is a shortcut for that.

```scala
implicit val ec: ExecutionContext = ExecutionContext.global

val io = IO.deferFuture { futureGetUsers }

io foreach { users => users foreach println } //=> "side effect"
io foreach { users => users foreach println } //=> "side effect"
```

4. Resources

Resources

- Code and Slides of this Talk:
  https://github.com/hermannhueck/implementing-io-monad
- Code and Slides for my Talk on: Future vs. Monix Task
  https://github.com/hermannhueck/future-vs-monix-task
- Monix Task 3.x Documentation (for comparison with IO)
  https://monix.io/docs/3x/eval/task.html
- Monix Task 3.x API Documentation (for comparison with IO)
  https://monix.io/api/3.0/monix/eval/Task.html
- Best Practice: "Should Not Block Threads"
  https://monix.io/docs/3x/best-practices/blocking.html
- What Referential Transparency can do for you
  Talk by Luka Jacobowitz at ScalaIO 2017
  https://www.youtube.com/watch?v=X-cEGEJMx_4

Thanks for Listening

Q & A

https://github.com/hermannhueck/implementing-io-monad