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 impl I was very much inspired by Monix *Task* which I studied at that time.

The API of my IO is very similar to the basics of Monix *Task*. The implementation also helped me to understand the IO Monad and Monix *Task* (which itsself is an impl of the IO Monad).

Interop with *Future* is also supported. You can convert *IO* to a *Future*. Vise 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

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?

```
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))
```

```
// 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?

```
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))
```

```
// 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

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

program()
```

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

2. IO Program without 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!")
  }
```

2. IO Program without 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"
```

- 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)

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

```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. Add *map* and *flatMap*

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

4. Add *map* and *flatMap*

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

```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. Add *map* and *flatMap*

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

```
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 ()
```

```
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. Add *IO.pure* and *IO.eval* to companion

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

object IO {
  def pure[A](value: A): IO[A] = IO { () => value }
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }
}
```

5. Add *IO.pure* and *IO.eval* to companion

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

object IO {
  def pure[A](value: A): IO[A] = IO { () => value }
  def eval[A](thunk: => A): IO[A] = IO { () => thunk }
}
```

```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 }
  _       <- IO.eval { println(s"Well hello, $name!") }
} yield ()

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

- *IO.pure* is for pure values.
- *IO.eval* simplifies the creation of *IO* instances (*Function0* no longer visible).

6. More synchronous run* methods

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

  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }

  // ----- impure sync run* methods

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

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

6. More synchronous run* methods

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

  def map[B](f: A => B): IO[B] = IO { () => f(run()) }
  def flatMap[B](f: A => IO[B]): IO[B] = IO { () => f(run()).run() }

  // ----- impure sync run* methods

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

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

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

val v1: Unit = program.run()                    // may throw an exception

val v2: Try[Unit] = program.runToTry

val v3: Either[Throwable, Unit] = program.runToEither
```

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

7. Other example: Authenticate Maggie

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

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

// running checkMaggie synchronously ...

println("\n>>> IO#run:")
println(checkMaggie.run())                //=> true, may throw an Exception

println("\n>>> IO#runToTry:")
printAuthTry(checkMaggie.runToTry)        //=> true

println("\n>>> IO#runToEither:")
printAuthEither(checkMaggie.runToEither)  //=> true
```

8. Add asynchronous 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 =
    ec.execute(new Runnable {
      override def run(): Unit = callback(runToTry)
    })

  def runAsync(callback: Either[Throwable, A] => Unit)
                                (implicit ec: ExecutionContext): Unit =
    ec.execute(() => callback(runToEither))   // SAM for a Runable
}
```

Using the async run* methods

```scala
// check username and password for correctness
def authenticate(username: String, password: String): IO[Boolean] = ???

// check Maggie's username and password for correctness
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
```

Callbacks

```scala
def printAuthTry[A](tryy: Try[A]): Unit = println(
  tryy.fold(
    ex => ex.toString,
    isAuthenticated => s"isAuthenticated = $isAuthenticated")
)

def printAuthEither[A](either: Either[Throwable, A]): Unit =
  println(either.fold(
    ex => ex.toString,
    isAuthenticated => s"isAuthenticated = $isAuthenticated")
  )


def authCallbackTry[A]: Try[A] => Unit = printAuthTry

def authCallbackEither[A]: Either[Throwable, A] => Unit = printAuthEither
```

9. Helper method *runAsync0*

- allows for refactoring of *runAsync* and *runOnComplete*

```
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 =
    // convert Try based callback into an Either based callback
    runAsync(ea => callback(ea.toTry))

  def runAsync(callback: Either[Throwable, A] => Unit)
                                  (implicit ec: ExecutionContext): Unit =
    runAsync0(ec, callback)

  private def runAsync0(ec: ExecutionContext,
                        callback: Either[Throwable, A] => Unit): Unit =
    ec.execute(() => callback(runToEither))   // SAM for a Runable
}
```

10. Add callback *foreach*

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

  // Triggers async evaluation of this IO, executing the given function for the ge
  // WARNING: Will not be called if this IO is never completed or if it is complet
  // Since this method executes asynchronously and does not produce a return value
  // any non-fatal exceptions thrown will be reported to the ExecutionContext.
  def foreach(f: A => Unit)(implicit ec: ExecutionContext): Unit =
    runAsync {
      case Left(ex) => ec.reportFailure(ex)
      case Right(value) => f(value)
    }
}
```

10. Add callback *foreach*

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

  // Triggers async evaluation of this IO, executing the given function for the ge
  // WARNING: Will not be called if this IO is never completed or if it is complet
  // Since this method executes asynchronously and does not produce a return value
  // any non-fatal exceptions thrown will be reported to the ExecutionContext.
  def foreach(f: A => Unit)(implicit ec: ExecutionContext): Unit =
    runAsync {
      case Left(ex) => ec.reportFailure(ex)
      case Right(value) => f(value)
    }
}
```

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

- *foreach* run asynchronously
- *foreach* swallows exceptions. Prefer *runAsync*!

11. Change case class to trait, make *IO* an ADT

```
sealed trait IO[A] {

  def run: () => A

  // ...
}
```

11. Change case class to trait, make *IO* an ADT

```scala
sealed trait IO[A] {

  def run: () => A

  // ...
}
```

```scala
object IO {

  private case class Pure[A](run: () => A) extends IO[A]
  private case class Eval[A](run: () => A) extends IO[A]

  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)
}
```

- The app works as before.

12. Implement concrete *run* method

- pattern match over ADT subtypes.
- make *run* (which may throw exeptions) private.

```
sealed trait IO[A] {

  private def run(): A = this match {
    case Pure(thunk) => thunk()
    case Eval(thunk) => thunk()
  }
}
```

12. Implement concrete *run* method

- pattern match over ADT subtypes.
- make *run* (which may throw exeptions) private.

```
sealed trait IO[A] {

  private def run(): A = this match {
    case Pure(thunk) => thunk()
    case Eval(thunk) => thunk()
  }
}
```

```
object IO {
  // ... as before
}
```

- The app works as before.

13. 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
```

14. Monad instance for *IO*

```scala
sealed trait IO[A] {

  def map[B](f: A => B): IO[B] = IO { f(run()) }
  def flatMap[B](f: A => IO[B]): IO[B] = IO { f(run()).run() }
}

object IO {

  // Monad instance defined in implicit context
  implicit def 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
    override def tailRecM[A, B](a: A)(f: A => IO[Either[A, B]]): IO[B] = ???
  }
}
```

Convert computations into monadic code

```
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***

```
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***

```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 *cats.Id***

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

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

```scala
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***

```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 *cats.Id***

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

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

```scala
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***

```scala
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
```

15. Extend ADT with *Suspend*

```
sealed trait IO[A] {

  private def run(): A = this match {
    case Pure(thunk) => thunk()
    case Eval(thunk) => thunk()
    case Suspend(thunk) => thunk().run()
  }
}

object IO {

  private case class Pure[A](thunk: () => A) extends IO[A]
  private case class Eval[A](thunk: () => A) extends IO[A]
  private case class Suspend[A](thunk: () => IO[A]) extends IO[A]

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

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

Applying *suspend* or *defer*

- These methods defer the (possibly immediate) side effect of the inner *IO*.

Applying *suspend* or *defer*

- These methods defer the (possibly immediate) side effect of the inner *IO*.

IO.pure

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

Applying *suspend* or *defer*

- These methods defer the (possibly immediate) side effect of the inner *IO*.

IO.pure

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

IO.defer

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

16. Extend ADT with *FlatMap*

- … and implement *map* in terms of *flatMap*.

```
sealed trait IO[A] {

  private def run(): A = this match {
    case Pure(thunk) => thunk()
    case Eval(thunk) => thunk()
    case Suspend(thunk) => thunk().run()
    case FlatMap(src, f) => f(src.run()).run()
  }

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

object IO {

  private case class Pure[A](thunk: () => A) extends IO[A]
  private case class Eval[A](thunk: () => A) extends IO[A]
  private case class Suspend[A](thunk: () => IO[A]) extends IO[A]
  private case class FlatMap[A, B](src: IO[A], f: A => IO[B]) extends IO[B]
}
```

- The app (Authenticate Maggie) works as before.

17. Create *IO[A]* from *Try[A]* or from *Either[Throwable, A]*

```scala
object IO {

  def fromTry[A](tryy: Try[A]): IO[A] = IO {
    tryy match {
      case Failure(t) => throw t
      case Success(value) => value
    }
  }

  def fromEither[A](either: Either[Throwable, A]): IO[A] = IO {
    either match {
      case Left(t) => throw t
      case Right(value) => value
    }
  }
}
```

Using *fromTry[A]* and *fromEither[A]*

```
val tryy = Try { User.getUsers }
val io1 = IO.fromTry(tryy)
io1 runOnComplete {                // can use runAsync as well
  case Failure(t) => println(t.toString)
  case Success(users) => users foreach println
}
```

```
val either = Try { User.getUsers }.toEither
val io2 = IO.fromEither(either)
io2 runAsync  {                    // can use runOnComplete as well
  case Left(t) => println(t.toString)
  case Right(users) => users foreach println
}
```

18. *IO.fromFuture* converts *Future* to *IO*

```
object IO {

  def fromFuture[A](future: Future[A]): IO[A] =
    future.value match {
      case Some(try0) => fromTry(try0)
      case None => IO.eval { Await.result(future, Duration.Inf) } // BLOCKING!!!
    }
}
```

18. *IO.fromFuture* converts *Future* to *IO*

```
object IO {

  def fromFuture[A](future: Future[A]): IO[A] =
    future.value match {
      case Some(try0) => fromTry(try0)
      case None => IO.eval { Await.result(future, Duration.Inf) } // BLOCKING!!!
    }
}
```

- Attention: The implementation of *fromFuture* is a bit simplistic. Waiting for the *Future* to complete might block a thread! Should be FIXED!!! That is not easy!

Using *IO.fromFuture*

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

Using *IO.fromFuture*

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

- *fromFuture* is eager. (The *Future* runs eagerly before *fromFuture* is invoked.)

```
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
  }
```

- *fromFuture* is eager. (The *Future* runs eagerly before *fromFuture* is invoked.)

```
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 }
```

- *fromFuture* wrapped in *defer* is lazy. (The *Future* will run when the *IO* is run.)

```
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. *deferFuture* turns eager *Future* into lazy *IO*

```
object IO {

  def fromFuture[A](future: Future[A]): IO[A] =
    future.value match {
      case Some(try0) => fromTry(try0)
      case None => IO.eval { Await.result(future, Duration.Inf) } // BLOCKING!!!
    }

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

- An *ExecutionContext* is still required to create the *Future*!
- The question is: How can that be avoided?
- We want to create an *IO* from a *Future* without providing an EC. The EC will be provided when we run the *IO*.
- See steps 20 and 21.

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
  }
```

- *fromFuture* wrapped in *defer* is lazy. (The *Future* will run when the *IO* is run.)

```
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*

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

- *fromFuture* wrapped in *defer* is lazy. (The *Future* will run when the *IO* is run.)

```
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 *IO.defer(IO.fromFuture(f))*.

```
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"
```

20. Synchronous run* methods take an EC

```
sealed trait IO[A] {

  private def run(implicit ec: ExecutionContext): A = this match {
    case Pure(thunk) => thunk()
    case Eval(thunk) => thunk()
    case Suspend(thunk) => thunk().run
    case FlatMap(src, f) => f(src.run).run
  }

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

  def runToEither(implicit ec: ExecutionContext): Either[Throwable, A] = runToTry.
}
```

- The function wrapped in *IO* may create a *Future* (without running it).
- Even if the *IO* runs synchronously the wrapped *Future* runs async.
- An implicit *ExecutionContext* is also required in the sync run* functions.
- This prepares for the impl of *deferFutureAction*.

21. Implement *IO#deferFutureAction*

```
sealed trait IO[A] {
  private def run(implicit ec: ExecutionContext): A = this match {
    case  ...
    case FutureToTask(ec2Future) => fromFuture(ec2Future(ec)).run(ec)
  }
}
object IO {
  private case class ...
  private case class FutureToTask[A](f: ExecutionContext => Future[A]) extends IO[A

  def deferFutureAction[A](ec2Future: ExecutionContext => Future[A]): IO[A] =
    FutureToTask(ec2Future)
}
```

21. Implement *IO#deferFutureAction*

```
sealed trait IO[A] {
  private def run(implicit ec: ExecutionContext): A = this match {
    case  ...
    case FutureToTask(ec2Future) => fromFuture(ec2Future(ec)).run(ec)
  }
}
object IO {
  private case class ...
  private case class FutureToTask[A](f: ExecutionContext => Future[A]) extends IO[A

  def deferFutureAction[A](ec2Future: ExecutionContext => Future[A]): IO[A] =
    FutureToTask(ec2Future)
}
```

- ADT *IO* is extended with a new sub type *FutureToTask*.
- *FutureToTask* takes a function *ExecutionContext => Future[A]*.
- *deferFutureAction* takes a function *ExecutionContext => Future[A]* and passes it to *FutureToTask*.
- Thus the creation of the *Future* is defered to the central *run* method.

Using *IO#deferFutureAction*

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

Using *IO#deferFutureAction*

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

- With *IO.deferFuture* an EC is required when <u>creating</u> the *IO*.

```
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"
```

Using *IO#deferFutureAction*

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

- With *IO.deferFuture* an EC is required when creating the *IO*.

```
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"
```

- With *IO.deferFutureAction* an EC is required when running the *IO*.

```
val io = IO.deferFutureAction { implicit ec: ExecutionContext => futureGetUsers }

implicit val ec: ExecutionContext = ExecutionContext.global

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

22. Implement *IO.raiseError*

```
sealed trait IO[A] {
  private def run(implicit ec: ExecutionContext): A = this match {
    case  ...
    case Error(exception) => throw exception
  }
}
object IO {
  private case class ...
  private case class Error[A](exception: Throwable) extends IO[A]

  def raiseError[A](exception: Exception): IO[A] = Error[A](exception)
  def failed[A](exception: Exception): IO[A] = raiseError(exception)
              // analogous to Future.failed
}
```

22. Implement *IO.raiseError*

```scala
sealed trait IO[A] {
  private def run(implicit ec: ExecutionContext): A = this match {
    case  ...
    case Error(exception) => throw exception
  }
}
object IO {
  private case class ...
  private case class Error[A](exception: Throwable) extends IO[A]

  def raiseError[A](exception: Exception): IO[A] = Error[A](exception)
  def failed[A](exception: Exception): IO[A] = raiseError(exception)
              // analogous to Future.failed
}
```

- ADT *IO* is extended with a new sub type *Error* that wraps a *Throwable*.

22. Implement *IO.raiseError*

```
sealed trait IO[A] {
  private def run(implicit ec: ExecutionContext): A = this match {
    case  ...
    case Error(exception) => throw exception
  }
}
object IO {
  private case class ...
  private case class Error[A](exception: Throwable) extends IO[A]

  def raiseError[A](exception: Exception): IO[A] = Error[A](exception)
  def failed[A](exception: Exception): IO[A] = raiseError(exception)
              // analogous to Future.failed
}
```

- ADT *IO* is extended with a new sub type *Error* that wraps a *Throwable*.

```
val ioError: IO[Int] = IO.raiseError[Int](
                            new IllegalStateException("illegal state"))

implicit val ec: ExecutionContext = ExecutionContext.global
println(ioError.runToEither)
//=> Left(java.lang.IllegalStateException: illegal state)
```

23. Impl *IO#failed* (returns a failed projection).

```
sealed trait IO[A] {
  private def run(implicit ec: ExecutionContext): A = this match {
    case  ...
    case failed: Failed[A] => failed.get(ec).asInstanceOf[A]
  }

  // Returns a failed projection of this IO.
  def failed: IO[Throwable] = Failed(this)
}
object IO {
  private case class ...
  private case class Failed[A](io: IO[A]) extends IO[Throwable] {
    def get(implicit ec: ExecutionContext): Throwable = try {
      io.run
      throw new NoSuchElementException("failed")
    } catch {
      case nse: NoSuchElementException if nse.getMessage == "failed" => throw nse
      case t: Throwable => t
    }
  }
}
```

23. Impl *IO#failed* (returns a failed projection).

```
sealed trait IO[A] {
  private def run(implicit ec: ExecutionContext): A = this match {
    case  ...
    case failed: Failed[A] => failed.get(ec).asInstanceOf[A]
  }

  // Returns a failed projection of this IO.
  def failed: IO[Throwable] = Failed(this)
}
object IO {
  private case class ...
  private case class Failed[A](io: IO[A]) extends IO[Throwable] {
    def get(implicit ec: ExecutionContext): Throwable = try {
      io.run
      throw new NoSuchElementException("failed")
    } catch {
      case nse: NoSuchElementException if nse.getMessage == "failed" => throw nse
      case t: Throwable => t
    }
  }
}
```

- ADT *IO* is extended with a new sub type *Failed* that wraps the other *IO* to project.

Using *IO#failed*

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

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

Using *IO#failed*

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

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.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*.

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 3.x Documentation
  https://monix.io/docs/3x/
- Monix 3.x API Documentation
  https://monix.io/api/3.0/
- 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