paidy

INTRODUCTION TO MONADS

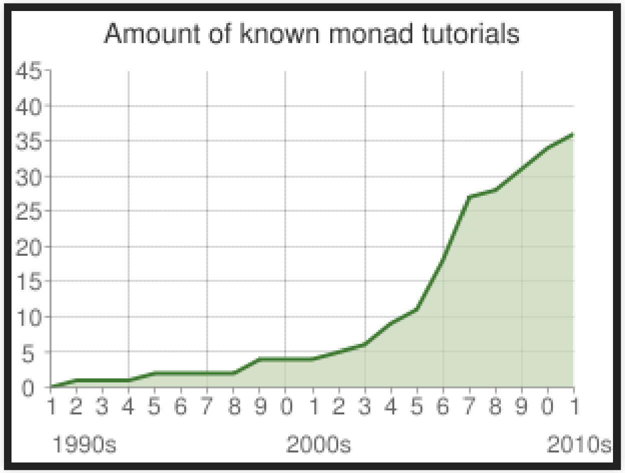
`>>=`

WHOAMI

Haemin Yoo

Github: @yoohaemin

- Engineering Intern at Paidy
- Enjoying Scala and FP
- From Seoul, Japanese is OK (a little)
- “A ‘newbie’, in Haskell, is someone who hasn’t yet implemented a compiler. They’ve only written a monad tutorial” - Pseudonym
- One lucky newbie.



Source: HaskellWiki

AGENDA

- Try to write better code
- Arrive at `Monad`
- Ask questions.

BEFORE WE START

- Who uses the monadic interface `flatMap` and `pure` in daily work?
- Who are familiar with scala `Future`s?

BUSINESS LOGIC

- Get a String from the outside
- Append "monad"
- echo it back
- We're using the console right now

```scala
import scala.io.StdIn

def echo(): Unit = {
  val input = StdIn.readLine()
  val appended = input + "monad"
  println(appended)
}
```

What's wrong with this?

Mixing business logic with implementation detail

- Testing is hard
- Changing is hard
- Reading is hard

INTERFACES!

```scala
trait ConsoleIO {
  def readLine(): String
  def printLine(str: String): Unit
}

object Terminal extends ConsoleIO {
  import scala.io.StdIn
  def readLine(): String = StdIn.readLine()
  def printLine(str: String): Unit = println(str)
}

def echo(console: ConsoleIO): Unit = {
  val input = console.readLine()
  val appended = input + "monad"
  console.printLine(appended)
}
```

MORE IMPLEMENTATIONS!

```scala
object DummyIO extends ConsoleIO {
  /* ... */
}

class FileIO(path: String) extends ConsoleIO {
  /* ... */
}
```

New business requirement:

async

No problem:

```
class SomeAsyncIO(path: String) extends ConsoleIO {
  def readLine(): Future[String] = //...
  def printLine(str: String): Future[Unit] = //...
}
```

Hmm...

```
trait ConsoleIO {
  def readLine(): String
  def printLine(str: String): Unit
}
```

Should we change the interface?

- Application code must be rewritten. (okay..?)
- Test for business logic must be rewritten into async. (Arrgh)

WHAT IS THE PROBLEM?

- What's so special about **going async**?
- Language assumes synchronous execution!
- How to describe **pure** business logic without an execution strategy?

ENCODING EFFECTS INTO TYPES

```scala
trait ConsoleIO[F[_]] {
  def readLine(): F[String]
  def printLine(str: String): F[Unit]
}

object SyncTerminalIO extends ConsoleIO[Id] { //No-op
  def readLine(): String = //...
  def printLine(str: String): Unit = //...
}

object AsyncTerminalIO extends ConsoleIO[Future] {
  def readLine(): Future[String] = //...
  def printLine(str: String): Future[Unit] = //...
}
```

BACK TO OUR `echo` METHOD...

```
// Caller of echo decides which strategy to use
def echo[F[_]](console: ConsoleIO[F]): ??? = {
  val input: ??? = console.readLine()
  //Hmm......
}
```

- what should the types be?
- `F[Unit]`
- `F[String]`

```scala
def echo[F[_]](console: ConsoleIO[F]): F[Unit] = {
  val input: F[String] = console.readLine()
  val appended = ??? //Hmm......
  console.printLine(appended) //Type mismatch
}
```

Ok, so we're stuck.

- How do we append the string?
- How do we pass a `F[String]` to a method that receives a `String`?
- We need some constraint for F (F needs to implement an interface.)

WHAT METHODS SHOULD THE INTERFACE HAVE?

Manipulation of the data stored inside the context.

Let's just call this, `extractAndManipulate`.

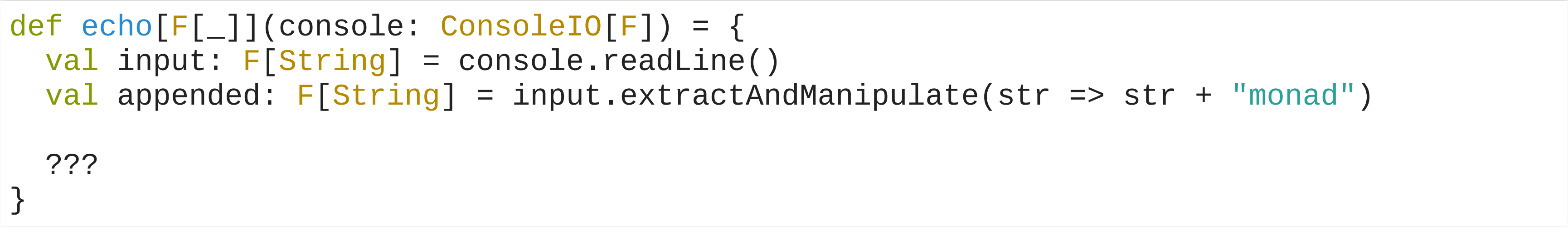
```
def echo[F[_]](console: ConsoleIO[F]) = {
  val input: F[String] = console.readLine()
  val appended: ??? = input.extractAndManipulate(str => str + "monad")

  ???
}
```

What is the type of `appended`?

The context still stays the same.

`F[String]`



```
def echo[F[_]](console: ConsoleIO[F]) = {
  val input: F[String] = console.readLine()
  val appended: F[String] = input.extractAndManipulate(str => str + "monad")

  ???
}
```

SEQUENCING OPERATIONS

```
def echo[F[_]](console: ConsoleIO[F]): F[Unit] = {
  val input: F[String] = console.readLine()
  val appended = input.extractAndManipulate(str => str + "monad")
  // do Something with these two:
  // appended
  // console.printLine()

  ???
}
```

WHAT METHODS SHOULD THE INTERFACE HAVE?

Shoving effectful data into an effectful operation

Let's just call this, `extractAndDo`.

```scala
def echo[F[_]](console: ConsoleIO[F]): F[Unit] = {
  val input: F[String] = console.readLine()
  val appended = input.extractAndManipulate(str => str + "monad")
  appended.extractAndDo(result => console.printLine(result))
}
```

In order to chain operations...

```scala
def extractAndManipulate(effectfulData: F[A])(manipulate: A => B): F[B]
def extractAndDo(effectfulData: F[A])(effectfulOperation: A => F[B]): F[B]
```

Real names!

```
def map(a: F[A])(f: A => B): F[B]
def flatMap(a: F[A])(f: A => F[B]): F[B]
```

IS THIS ALL?

New business logic:

- Echo and return the string.
- But if the string starts with '#', don't echo and just return the string.

```scala
def echo[F[_]](console: ConsoleIO[F]): F[String] = {
  val input: F[String] = console.readLine()
  val appended = input.extractAndManipulate(str => str + "monad")
  appended.extractAndDo { (result: String) =>
    if (result.head == '#') console.printLineAndReturn(result)
    else result //uh oh
  }
}
```

Need a way to lift a preexisting value into the context.

```
def pure(a: A): F[A]
```

```scala
def echo[F[_]](console: ConsoleIO[F]): F[String] = {
  val input: F[String] = console.readLine()
  val appended = input.map(str => str + "monad")
  appended.flatMap { (result: String) =>
    if (result.head == '#')
      console.printLine(result).map(_ => result)
    else result.pure
  }
}
```

We now have an interface called Monad.

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

Monad is just an interface for generic types specifically allowing chaining of effectful operations.

FOR-YIELD

Special syntax for `map` and `flatMap`

```
def echo[F[_]: Monad](console: ConsoleIO[F]): F[String] =
  for {
    input <- console.readLine()
    appended = input + "monad"
    result <- if (appended == '#')
                console.printLine(result).map(_ => result)
              else
                appended.pure
  } yield result
```

map IN TERMS OF flatMap AND pure

In Scala, fields or methods inside traits can have concrete implementations.

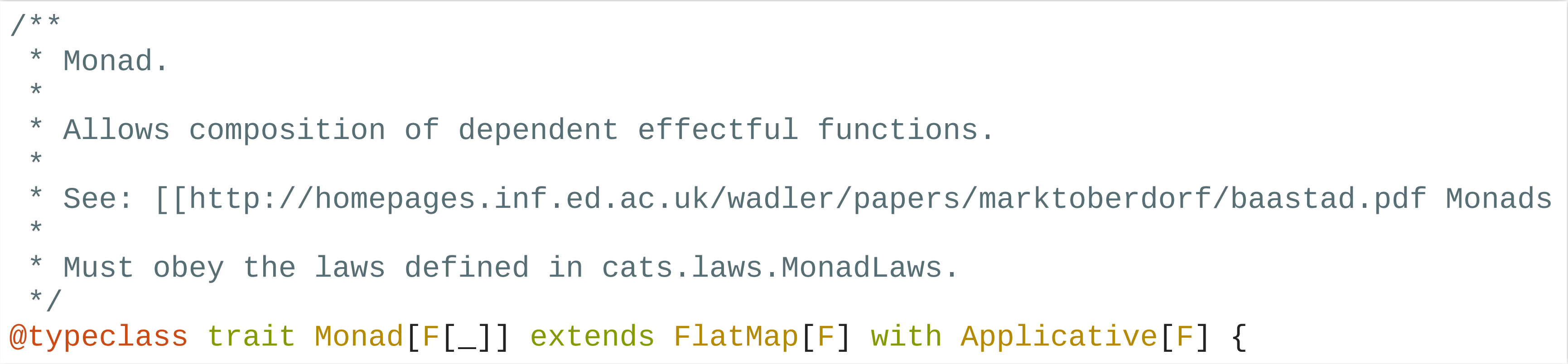
It turns out map can be defined only with flatMap and pure, making implementation of the Monad trait easier.

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

  //No need to reimplement
  def map(fa: F[A])(f: A => B): F[B] =
    fa.flatMap(a => f(a).pure)
}
```

So, anything that defines `pure` and `flatMap` may be called a `Monad`?

No.



```
/**
 * Monad.
 *
 * Allows composition of dependent effectful functions.
 *
 * See: [[http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf Monads
 *
 * Must obey the laws defined in cats.laws.MonadLaws.
 */
@typeclass trait Monad[F[_]] extends FlatMap[F] with Applicative[F] {
```

- There's something more to `Monad`s than just exposing sequencing interfaces.

The values describing business logic should only **DESCRIBE** the operation, not **DO** anything

Referential Transparency

Whole point is to decouple logic with an execution strategy

Any `Monad` implementation needs to pass certain tests.

Monad Laws

For example, using `scala.concurrent.Future` to do IO or other side effects breaks referential transparency.

It is not possible to even test lawfulness in such cases.

Why is RT and law important?

```
//printAndReturnOne: scala.concurrent.Future[Int]
for {
  first  <- printAndReturnOne
  second <- printAndReturnOne
} yield first + second
```

```scala
def printAndReturnOne: Future[Int] =
  Future { println(1); 1 }

for {
  first  <- printAndReturnOne
  second <- printAndReturnOne
} yield first + second
```

```scala
val printAndReturnOne: Future[Int] =
  Future { println(1); 1 }

for {
  first  <- printAndReturnOne
  second <- printAndReturnOne
} yield first + second
```

Hard to reason about that.

```scala
import cats.effect.IO

def printAndReturnOne: IO[Int] =
  IO { println(1); 1 }

for {
  first  <- printAndReturnOne
  second <- printAndReturnOne
} yield first + second
```

What happens?

Nothing.

```scala
import cats.effect.IO

def printAndReturnOne: IO[Int] =
  IO { println(1); 1 }

val program = for {
  first  <- printAndReturnOne
  second <- printAndReturnOne
} yield first + second

program.unsafeRunSync()
// 1
// 1
```

MORE EFFECTS

```
int addOnePrimitive(int i)

Integer addOneBox(Integer i)
```

```
scala.util.Option[A]
```

WHAT IF A VALUE IS NULLABLE AND ASYNC?

Just wait for the next talk!

QUESTIONS?

paidy

カードのいらないカンタン決済

WE ARE HIRING!

https://engineering.paidy.com/