Pour quelques monades de plus...

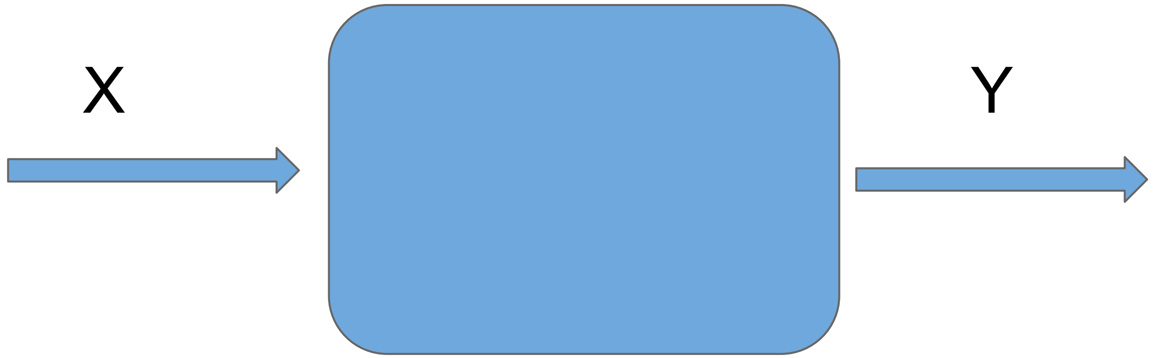
Apprendre Haskell vous fera le plus grand bien !

http://lyah.haskell.fr/

Un DSL pour ma base de données

exemple avec Aerospike

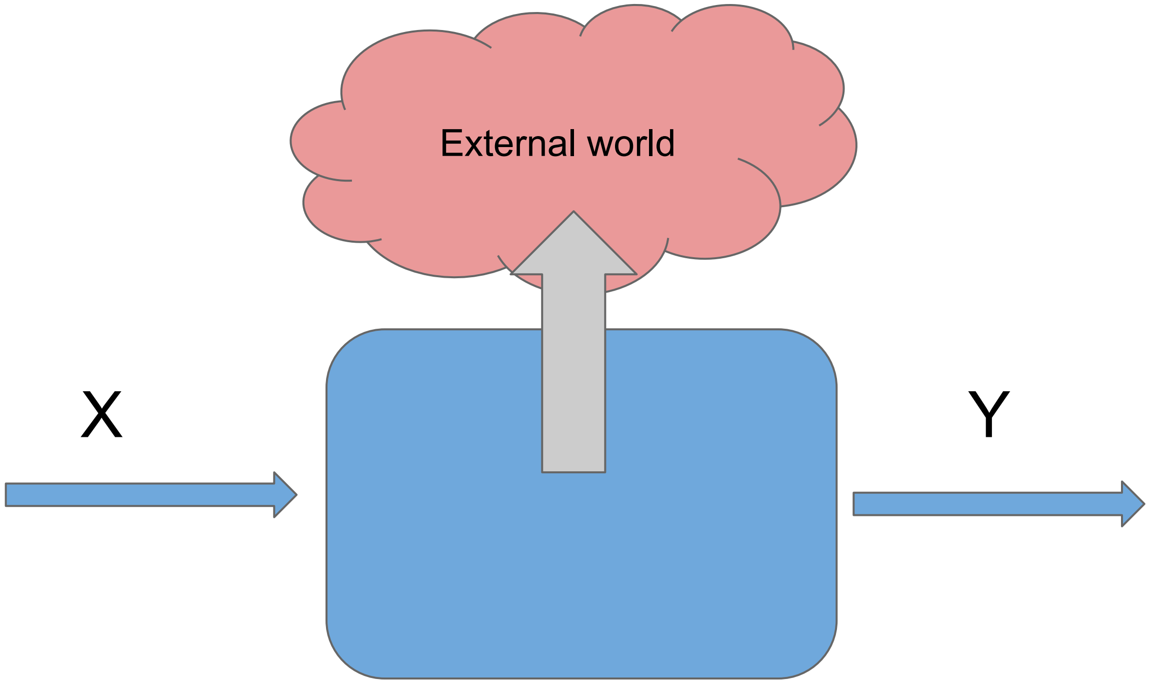
Pure function



Pure function

```
val increment: Int => Int = x => x + 1
```

Impure function



Impure function

```scala
val get: String => String = url => httpClient.get(url)
```

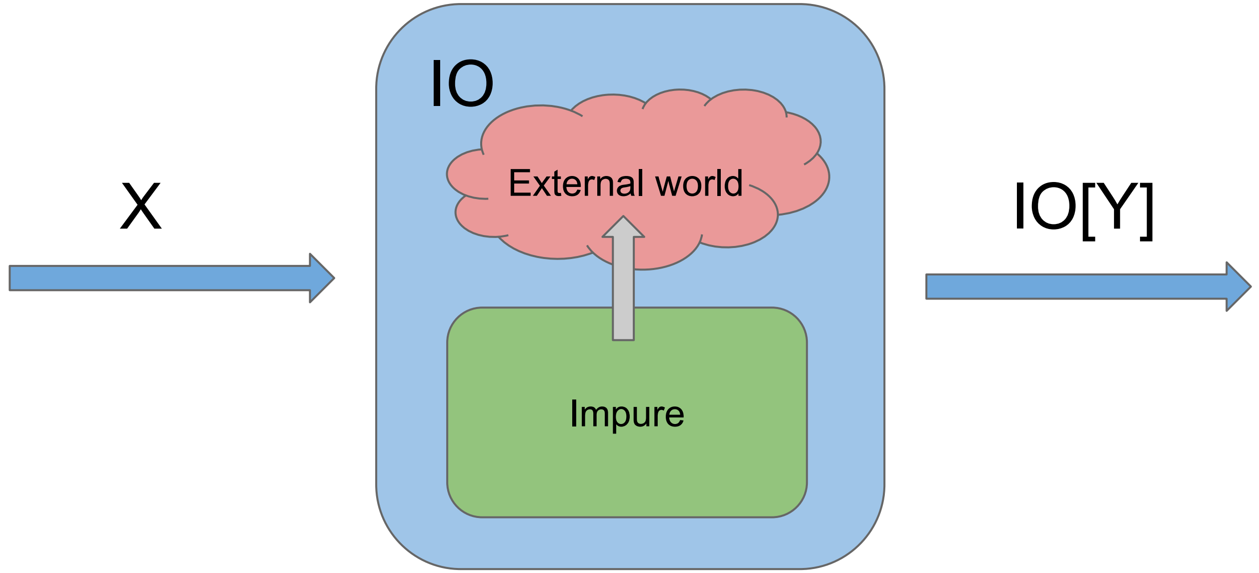
Real World

- La pureté c’est bien…
- … Mais dans le monde réel, c’est possible?

Real World

- La pureté c’est bien…
- … Mais dans une vraie app, c’est possible?

IO monad



Future[T]

- Approche concurrente
- Asynchrone
- Future[T] : Contient potentiellement une valeur T

```
val httpClient: HttpClient = ???
def getAsync(url: String): Future[String] = {
 val promise = Promise[String]()
 httpClient.get(url, AsyncHandler {
   content => promise.success(content),
   error => promise.failure(error)
 })
 promise.future
}
```

Future[T]

Future est une monade IO?

Future[T]

Future est une monade IO?

Referential transparency

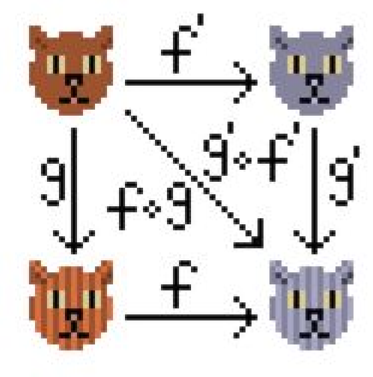
```
val future: Future[A] = ???

!=

def future: Future[A] = ???
```

Scala IO library

cats-effect



Monix

scalaz-effect

SCALAZ
PRINCIPLED FUNCTIONAL PROGRAMMING FOR SCALA

IO with Scala

- Confiner les effets de bords

```
val div: IO[Unit] = IO {
    val a = io.StdIn.readInt()
    val b = io.StdIn.readInt()
    if (b == 0) throw new IllegalArgumentException()
    else println(a / b)
}
div.runUnsafeSync //execute in real world
```

Real world app : Aerospike

- clé -> valeur
- In memory / SSD
- Clustering

AEROSPIKE

IO with Aerospike

```
def put[A](key: Key, value: A)(client: AerospikeClient): IO[Unit] = IO.async { cb =>
        val listener = new WriteListener {
                override def onFailure(exception: AerospikeException): Unit = cb(Left(exception))
                override def onSuccess(key: Key): Unit = cb(Right(()))
        }
        client.put(key, listener, bins)
}
val program: IO[Unit] = put(key, value)(client)
program.runUnsafeSync //execute in real world
```

IO with Aerospike

```scala
def put[A](key: Key, value: A)(client: AerospikeClient): IO[Unit] = IO.async { cb =>
        val listener = new WriteListener {
                override def onFailure(exception: AerospikeException): Unit = cb(Left(exception))
                override def onSuccess(key: Key): Unit = cb(Right(()))
        }
        client.put(key, listener, bins)
}
val program: IO[Unit] = put(key, value)(client)
program.runUnsafeSync //execute in real world
```

IO with Aerospike

```
def put[A](key: Key, value: A)(client: AerospikeClient): IO[Unit] = IO.async { cb =>
        val listener = new WriteListener {
                override def onFailure(exception: AerospikeException): Unit = cb(Left(exception))
                override def onSuccess(key: Key): Unit = cb(Right(()))
        }
        client.put(key, listener, bins)
}
val program: IO[Unit] = put(key, value)(client)
program.runUnsafeSync //execute in real world
```

AerospikeIO[T]

- Réduire le boilerplate
- Description de l’interaction
- Pas de code technique
- Monade IO spécifique

Algèbre et interpréteur

Algèbre (monadique)

```scala
sealed trait AerospikeIO[A]

case class Get[A](key: Key) extends AerospikeIO[A]
case class Put[A](key: Key, value: A) extends AerospikeIO[Unit]

case class Pure[A](x: A) extends AerospikeIO[A]
case class Join[A, B](x: AerospikeIO[A], y: AerospikeIO[B]) extends
AerospikeIO[(A, B)]
case class FlatMap[A, B](x: AerospikeIO[A], f: A => AerospikeIO[B]) extends
AerospikeIO[B]
```

Algèbre (monadique)

```
sealed trait AerospikeIO[A]

case class Get[A](key: Key) extends AerospikeIO[A]
case class Put[A](key: Key, value: A) extends AerospikeIO[Unit]

case class Pure[A](x: A) extends AerospikeIO[A]
case class Join[A, B](x: AerospikeIO[A], y: AerospikeIO[B]) extends
AerospikeIO[(A, B)]
case class FlatMap[A, B](x: AerospikeIO[A], f: A => AerospikeIO[B]) extends
AerospikeIO[B]
```

Algèbre (monadique)

```scala
sealed trait AerospikeIO[A]

case class Get[A](key: Key) extends AerospikeIO[A]
case class Put[A](key: Key, value: A) extends AerospikeIO[Unit]

case class Pure[A](x: A) extends AerospikeIO[A]
case class Join[A, B](x: AerospikeIO[A], y: AerospikeIO[B]) extends
AerospikeIO[(A, B)]
case class FlatMap[A, B](x: AerospikeIO[A], f: A => AerospikeIO[B]) extends
AerospikeIO[B]
```

Operations

```
case class Mark(name: String, value: Int)


val program: AerospikeIO[Mark] = for {
    _ <- Put(keyBob, Mark("Bob", 17))
     bobMark <- Get[Mark](keyBob)
} yield bobMark
```

Operations (Représentation)

```
FlatMap[Mark](
    Put(keyBob, Mark(“Bob”, 17)),
    _ => Get[Mark](keyBob)
)
```

Interpréteur : Transformation Naturelle

```
val nt = new (List ~> Option) {
    def apply[A](list: List[A]): Option[A] = list match {
        case Nil => None
        case x :: xs => Some(x)
    }
}
//nt(1 :: Nil) => Some(1)
```

Interpréteur : Kleisli

```scala
val keepPositiveValues = (i: Int) => if (i > 0) Some(i) else None

val lowerThanTen = (i: Int) => if (i < 10) Some(i) else None

keepPositiveValues.andThen(lowerThanTen)
```

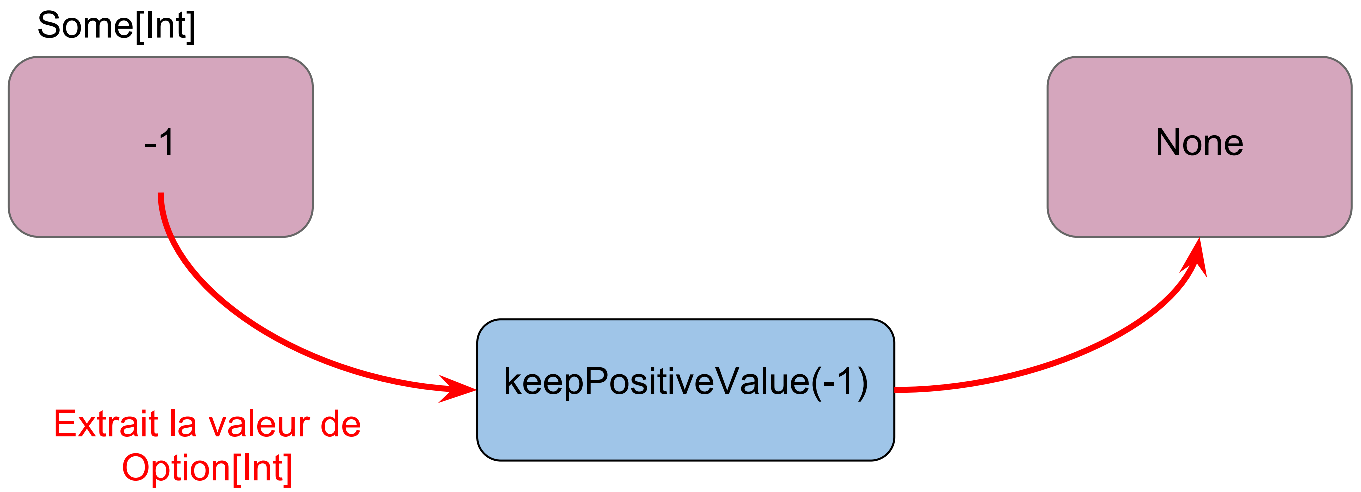
Interpréteur : Kleisli

```
val keepPositiveValues = (i: Int) => if (i > 0) Some(i) else None

val lowerThanTen = (i: Int) => if (i < 10) Some(i) else None

keepPositiveValues.andThen(lowerThanTen) // doesn't compile!
```

Interpréteur : Kleisli



Interpréteur : Kleisli

```
val keepPositiveValues = Kleisli[Option, Int, Int] { i =>
      if (i > 0) Some(i) else None
}
val lowerThanTen = Kleisli[Option, Int, Int] { i =>
      if (i < 10) Some(i) else None
}
keepPositiveValues.andThen(lowerThanTen) // compile
```

Interpréteur

```
type Stack[A] = Kleisli[Future, AerospikeClient, A]

~

AerospikeClient => Future[A]
```

Interpréteur

AerospikeIO ~> Stack

Interpréteur

```
case Put(key, bins) => Kleisli[Future, AerospikeClient, Unit] { client =>
    val promise = Promise[Unit]
    val listener = new WriteListener {
        def onFailure(exception) = promise.failure(exception)
        def onSuccess(key: Key)= promise.success(())
    }
    client.put(key, listener, bins)
    promise.future
}
```

Interpréteur

```
case Put(key, bins) => Kleisli[Future, AerospikeClient, Unit] { client =>
    val promise = Promise[Unit]
    val listener = new WriteListener {
        def onFailure(exception) = promise.failure(exception)
        def onSuccess(key: Key)= promise.success(())
    }
    client.put(key, listener, bins)
    promise.future
}
```

Interpréteur

```
case Put(key, bins) => Kleisli[Future, AerospikeClient, Unit] { client =>
    val promise = Promise[Unit]
    val listener = new WriteListener {
        def onFailure(exception) = promise.failure(exception)
        def onSuccess(key: Key)= promise.success(())
    }
    client.put(key, listener, bins)
    promise.future
}
```

Interpréteur

```
case Put(key, bins) => Kleisli[Future, AerospikeClient, Unit] { client =>
    val promise = Promise[Unit]
    val listener = new WriteListener {
        def onFailure(exception) = promise.failure(exception)
        def onSuccess(key: Key)= promise.success(())
    }
    client.put(key, listener, bins)
    promise.future
}
```

Run program

```
val interpreter: AerospikeIO ~> Kleisli[Future, AerospikeClient, ?] = ???

case class Mark(name: String, value: Int)
val program: Aerospike[Mark] = for {
    _       <- Put(keyBob, Mark("Bob", 17))
    bobMark <- Get[Mark](keyBob)
} yield bobMark

val kleisli: Kleisli[Future, AerospikeClient, Mark] = interpreter(program)
val result: Future[Mark] = kleisli(client)
```

Run program

```
val interpreter: AerospikeIO ~> Kleisli[Future, AerospikeClient, ?] = ???

case class Mark(name: String, value: Int)
val program: Aerospike[Mark] = for {
    _       <- Put(keyBob, Mark("Bob", 17))
    bobMark <- Get[Mark](keyBob)
} yield bobMark

val kleisli: Kleisli[Future, AerospikeClient, Mark] = interpreter(program)
val result: Future[Mark] = kleisli(client)
```

Run program

```
val interpreter: AerospikeIO ~> Kleisli[Future, AerospikeClient, ?] = ???

case class Mark(name: String, value: Int)
val program: Aerospike[Mark] = for {
    _       <- Put(keyBob, Mark("Bob", 17))
    bobMark <- Get[Mark](keyBob)
} yield bobMark

val kleisli: Kleisli[Future, AerospikeClient, Mark] = interpreter(program)
val result: Future[Mark] = kleisli(client)
```

Run program

```
val interpreter: AerospikeIO ~> Kleisli[Future, AerospikeClient, ?] = ???

case class Mark(name: String, value: Int)
val program: Aerospike[Mark] = for {
    _         <- Put(keyBob, Mark("Bob", 17))
    bobMark <- Get[Mark](keyBob)
} yield bobMark

val kleisli: Kleisli[Future, AerospikeClient, Mark] = interpreter(program)
val result: Future[Mark] = kleisli(client)
```

Sources

https://github.com/travisbrown/circe-algebra

https://github.com/jdegoes/scalaworld-2015

https://github.com/tpolecat/doobie

Merci!

https://github.com/tabmo/aerospike4s
https://github.com/rlecomte/presentation-fug-mtp

@lebalifant
@TabMoLabs