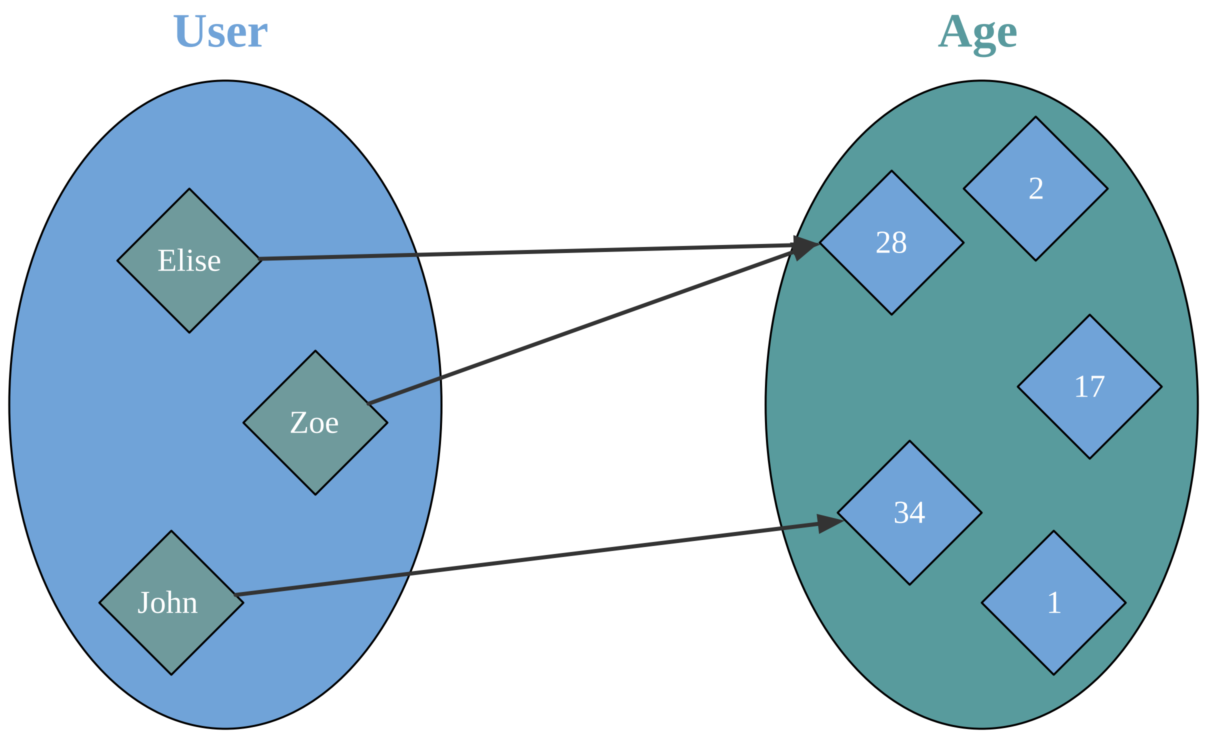
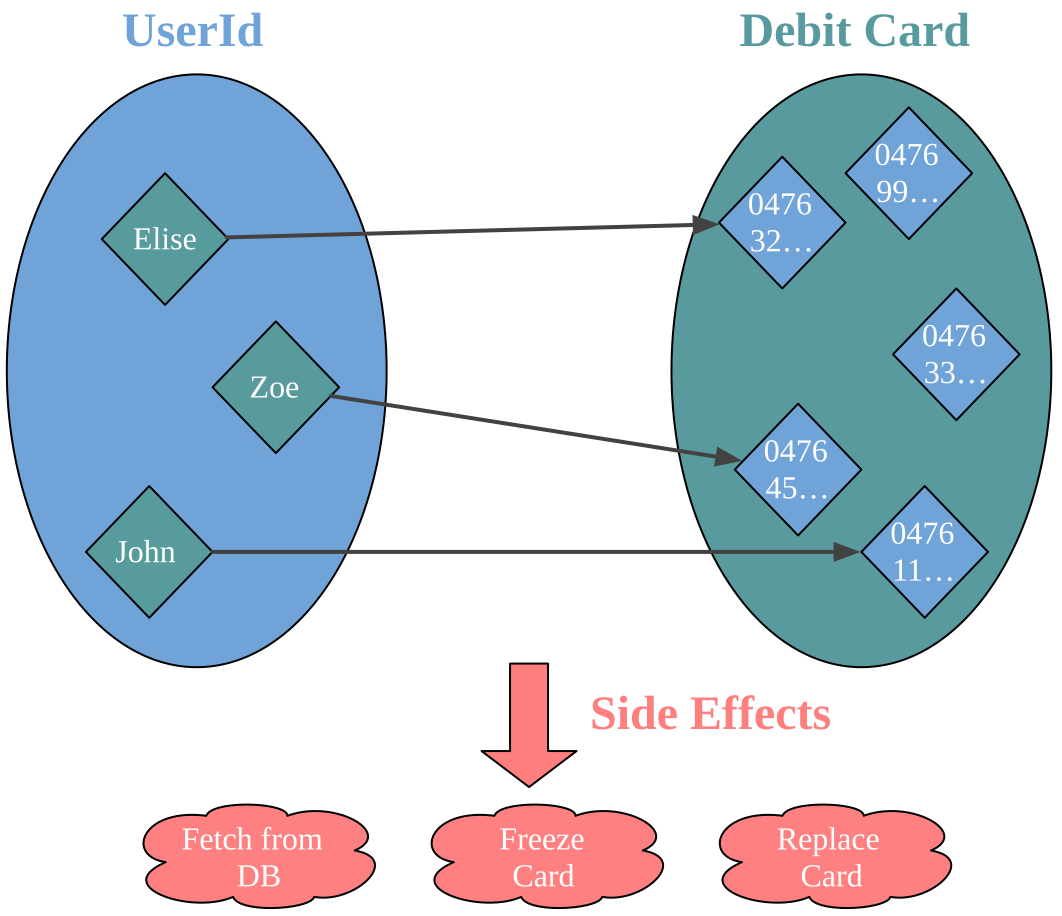
FOUNDATION

Side Effect

Pure function



Functions with side effects are not pure



Functional programming is useless *

Simon Peyton Jones co-author of haskell

A pure function cannot do anything

it can only produce a value

Create a value that describes actions

Create a value that describes actions

Interpret this value in Main

1. Encode description of actions

```scala
trait Description[A]
```

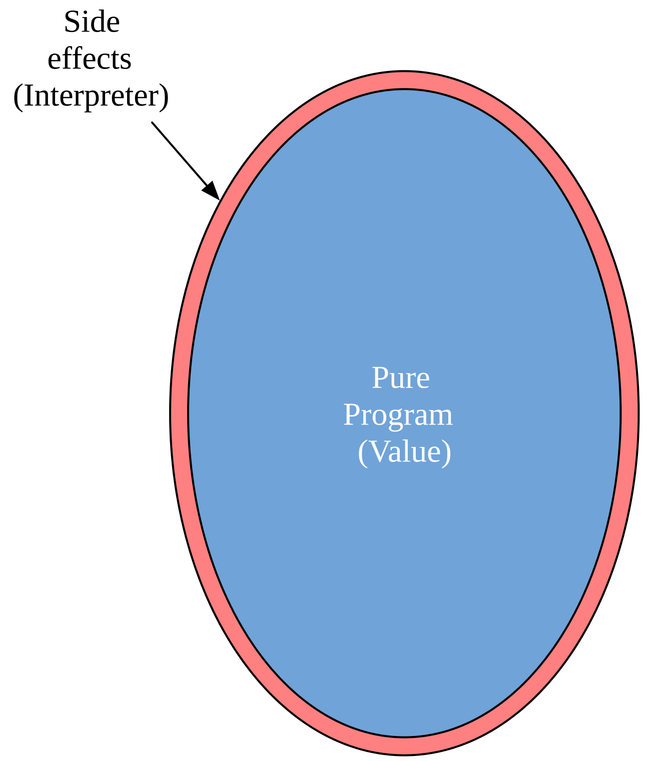
2. Define an interpreter of Description

```scala
def unsafeRun[A](fa: Description[A]): A = ???
```

3. Combine everything in Main

```scala
object Main extends App {
  val description: Description[Unit] = ???
  unsafeRun(description)
}
```

Run side effects at the edges



Examples of description / evaluation

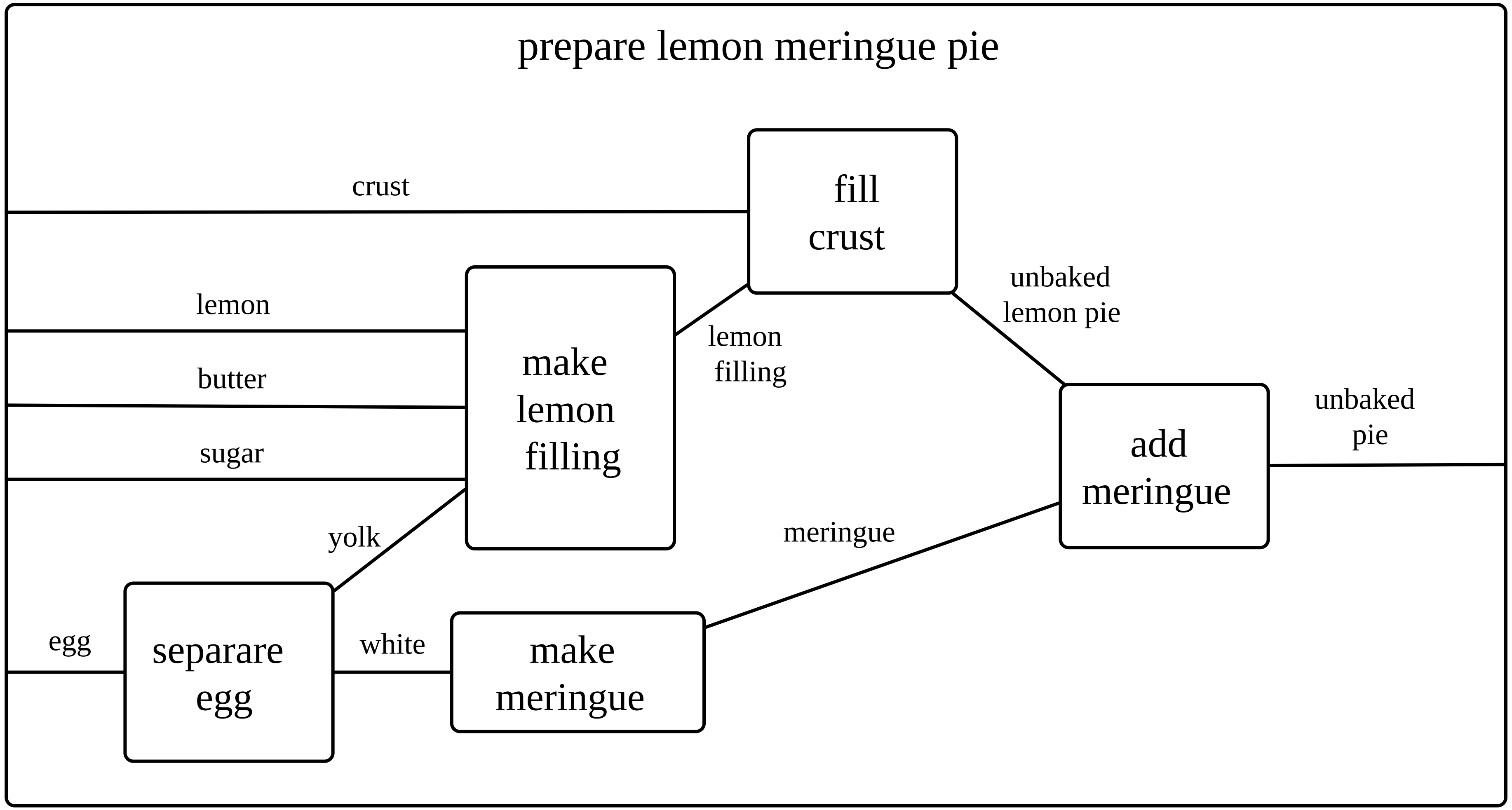
Cooking

1. Secret pasta recipe (Description)

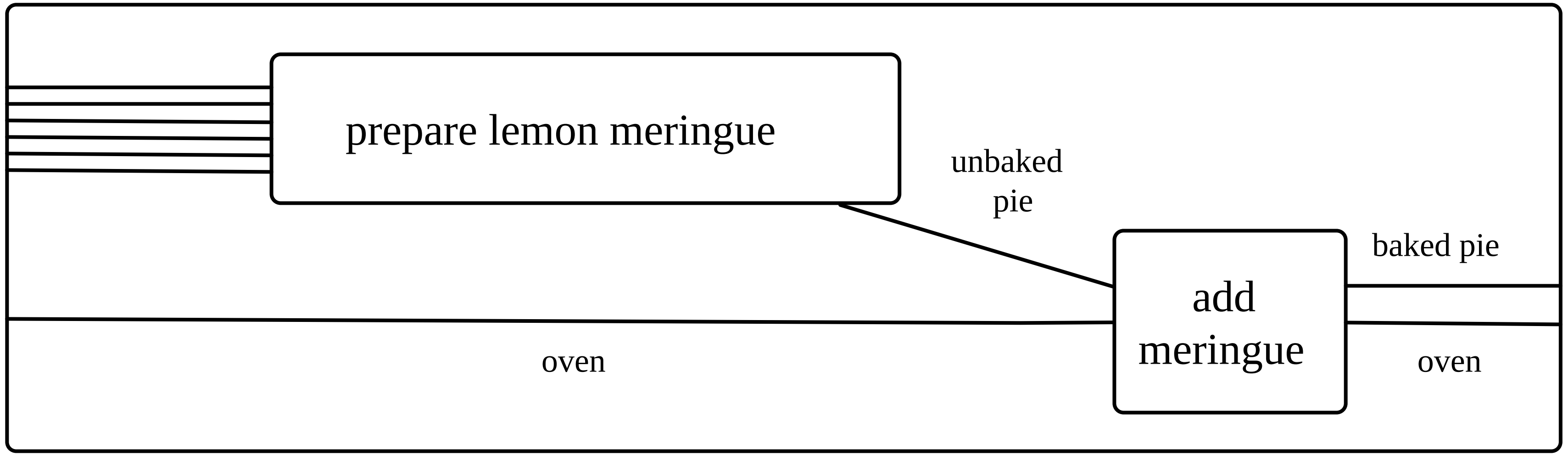
1. Boil 200 ml of water
2. Add 250 g of dry pasta
3. Wait 11 minutes
4. Drain the pasta

2. Cook (Unsafe evaluation)

Take the recipe and do it at home

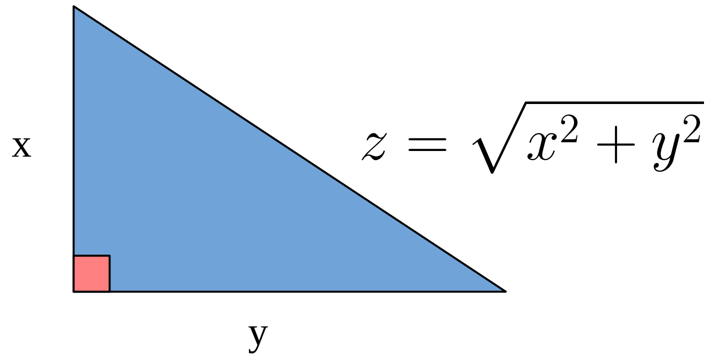

prepare lemon meringue pie
crust
fill crust
lemon
butter
sugar
make lemon filling
lemon filling
unbaked lemon pie
add meringue
unbaked pie
yolk
meringue
egg
separare egg
white
make meringue

String diagrams compose



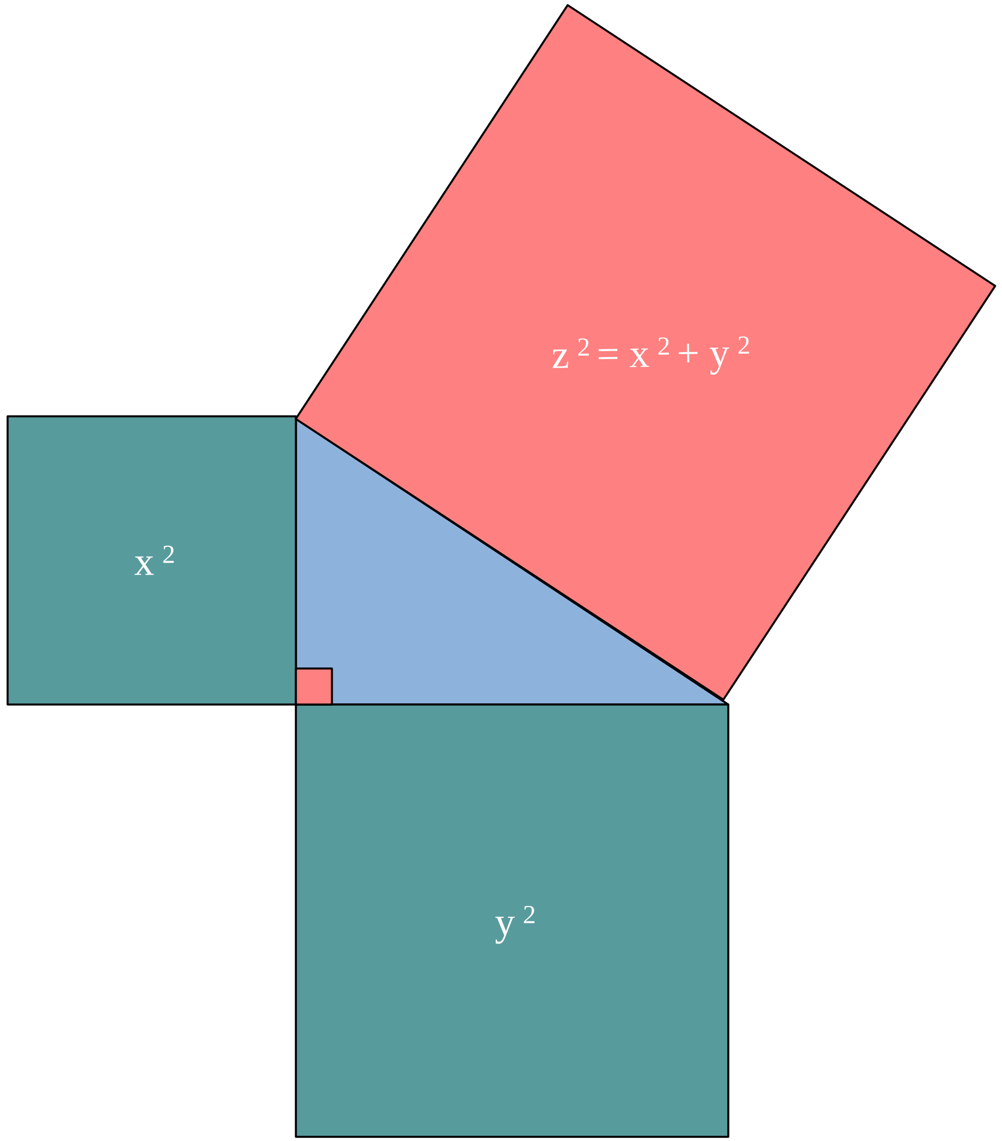
Mathematical formula

```
scala> val x = 2
x: Int = 2

scala> val y = 3
y: Int = 3

scala> val x2 = Math.pow(x, 2)
x2: Double = 4.0

scala> val y2 = Math.pow(y, 2)
y2: Double = 9.0

scala> val z  = Math.sqrt(x2 + y2)
z: Double = 3.605551275463989
```



Mathematical formula

```
scala> val x2 = Math.pow(x, 2)
x2: Double = 4.0

scala> val y2 = Math.pow(y, 2)
y2: Double = 9.0

scala> val z  = Math.sqrt(x2 + y2)
z: Double = 3.605551275463989

scala> Math.pow(z, 2)
res0: Double = 12.999999999999998

scala> x2 + y2
res1: Double = 13.0
```



Constraints Liberate, Liberties Constrain

How to encode description?

```scala
trait Description[A]

def unsafeRun[A](fa: Description[A]): A = ???
```

Thunk

```scala
type Thunk[A] = () => A  // Unit => A

def writeLine(message: String): Thunk[Unit] =
  () => println(message)

val today: Thunk[LocalDate] =
  () => LocalDate.now()

def fetch(url: String): Thunk[Iterator[String]] =
  () => scala.io.Source.fromURL(url)("ISO-8859-1").getLines
```

Thunk

```
type Thunk[A] = () => A  // Unit => A

def writeLine(message: String): Thunk[Unit] =
  () => println(message)

val today: Thunk[LocalDate] =
  () => LocalDate.now()

def fetch(url: String): Thunk[Iterator[String]] =
  () => scala.io.Source.fromURL(url)("ISO-8859-1").getLines
```

```
def unsafeRun[A](fa: Thunk[A]): A = fa()
```

Thunk

```scala
type Thunk[A] = () => A  // Unit => A

def writeLine(message: String): Thunk[Unit] =
  () => println(message)

val today: Thunk[LocalDate] =
  () => LocalDate.now()

def fetch(url: String): Thunk[Iterator[String]] =
  () => scala.io.Source.fromURL(url)("ISO-8859-1").getLines
```

```scala
def unsafeRun[A](fa: Thunk[A]): A = fa()
```

```
scala> val google = fetch("http://google.com")
google: Thunk[Iterator[String]] = $$Lambda$4450/0x0000000101817c40@eba0c4d

scala> unsafeRun(google).take(1).toList
res2: List[String] = List(<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en"><head><me
```

IO

```scala
trait IO[A] {

  def unsafeRun(): A // single abstract method

  def map[B](f: A => B): IO[B] = ???
  def flatMap[B](f: A => IO[B]): IO[B] = ???
  def retry: IO[A] = ???
}
```

IO

```
trait IO[A] {

  def unsafeRun(): A // single abstract method

  def map[B](f: A => B): IO[B] = ???
  def flatMap[B](f: A => IO[B]): IO[B] = ???
  def retry: IO[A] = ???
}
```

```
def writeLine(message: String): IO[Unit] =
  new IO[Unit] {
    def unsafeRun(): Unit = println(message)
  }
```

```
scala> val helloWorld = writeLine("Hello World")
helloWorld: IO[Unit] = $anon$1@70d979bd

scala> helloWorld.unsafeRun()
Hello World
```

Plan

- Implement our own IO
- Use IO to encode and test side effecting programs
- Discuss how to add asynchronicity
- Brief introduction to Free structures

IO Exercises

`exercises.sideeffect.IOExercises.scala`

Smart constructors

```scala
object IO {

  def succeed[A](constant: A): IO[A] = ???

  def effect[A](block: => A): IO[A] = ???

  def fail[A](error: Throwable): IO[A] = ???

}

trait IO[A] {
  def unsafeRun(): A
}
```

IO Summary

- An `IO` is a thunk of potentially impure code
- Composing `IO` is referentially transparent, nothing get executed
- It is easier to test `IO` if they are defined in a interface

IO Execution

IO execution

```
case class UserId (value: String)
case class OrderId(value: String)

case class User(userId: UserId, name: String, orderIds: List[OrderId])
```

```
def getUser(userId: UserId): IO[User] =
  IO.effect{
    val response = httpClient.get(s"http://foo.com/user/${userId.value}")
    if(response.status == 200) parseJson[User](response.body)
    else throw new Exception(s"Invalid status ${response.status}")
  }

def deleteOrder(orderId: OrderId): IO[Unit] =
  IO.effect{
    val response = httpClient.delete(s"http://foo.com/order/${orderId.value}")
    if(response.status == 200) () else throw new Exception(s"Invalid status ${response.status}")
  }
```

How is it executed?

```scala
def deleteAllUserOrders(userId: UserId): IO[Unit] =
  for {
    user <- getUser(userId)
    _    <- traverse(user.orderIds)(deleteOrder)
  } yield ()

object Main extends App {

  deleteAllUserOrders(UserId("1234")).unsafeRun()

}
```

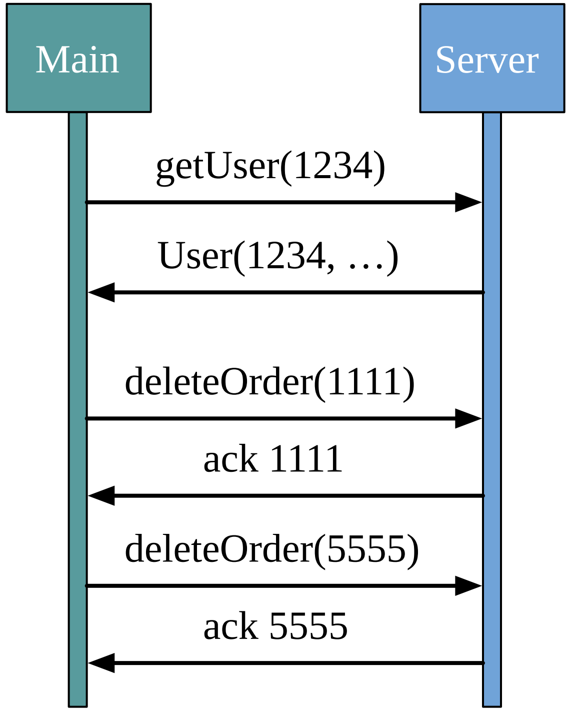
Discuss with your neighbour 3-4 min

How is it executed?

```scala
def deleteAllUserOrders(userId: UserId): IO[Unit] =
  for {
    user <- getUser(userId)
    // User("1234", "Rob", List("1111", "5555"))
    _    <- deleteOrder(user.orderIds(0)) // 1111
    _    <- deleteOrder(user.orderIds(1)) // 5555
  } yield ()

object Main extends App {

  deleteAllUserOrders(UserId("1234")).unsafeRun()

}
```

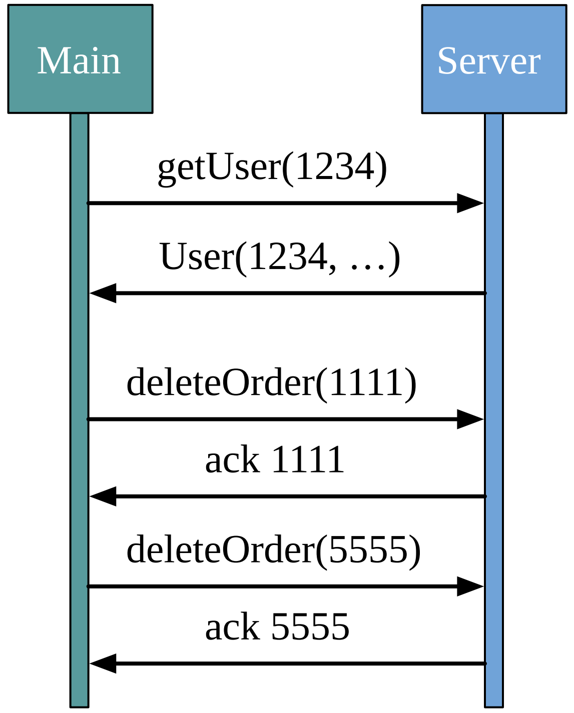
IO execution is sequential

```scala
def deleteAllUserOrders(userId: UserId): IO[Unit] =
  for {
    user <- getUser(userId)
    // User("1234", "Rob", List("1111", "5555"))
    _    <- deleteOrder(user.orderIds(0)) // 1111
    _    <- deleteOrder(user.orderIds(1)) // 5555
  } yield ()

object Main extends App {

  deleteAllUserOrders(UserId("1234")).unsafeRun()

}
```



Which IO could be evaluated concurrently?

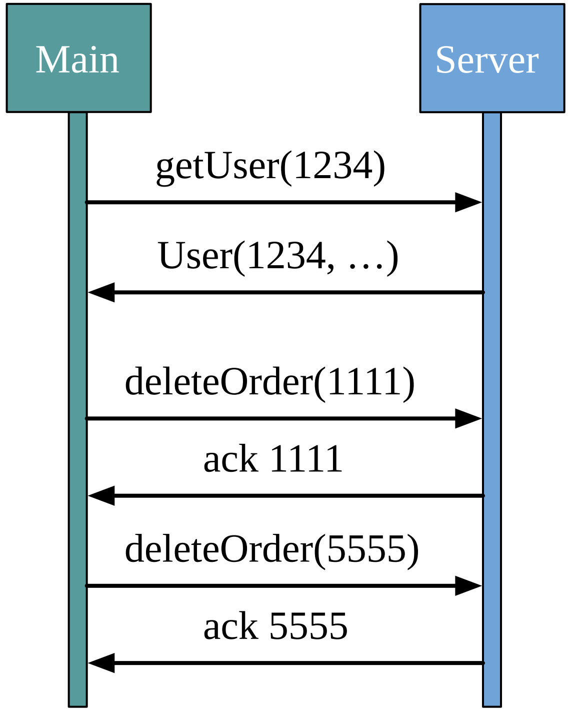
```scala
def deleteAllUserOrders(userId: UserId): IO[Unit] =
  for {
    user <- getUser(userId)
    // User("1234", "Rob", List("1111", "5555"))
    _    <- deleteOrder(user.orderIds(0)) // 1111
    _    <- deleteOrder(user.orderIds(1)) // 5555
  } yield ()

object Main extends App {

  deleteAllUserOrders(UserId("1234")).unsafeRun()

}
```



Discuss with your neighbour 3-4 min

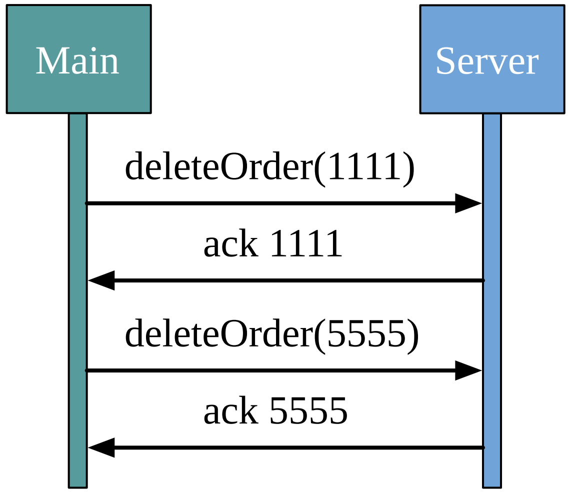
For comprehension cannot be done concurrently

```
def deleteAllUserOrders(userId: UserId): IO[Unit] =
  for {
    user <- getUser(userId)
    // User("1234", "Rob", List("1111", "5555"))
    _    <- deleteOrder(user.orderIds(0)) // 1111
    _    <- deleteOrder(user.orderIds(1)) // 5555
  } yield ()
```



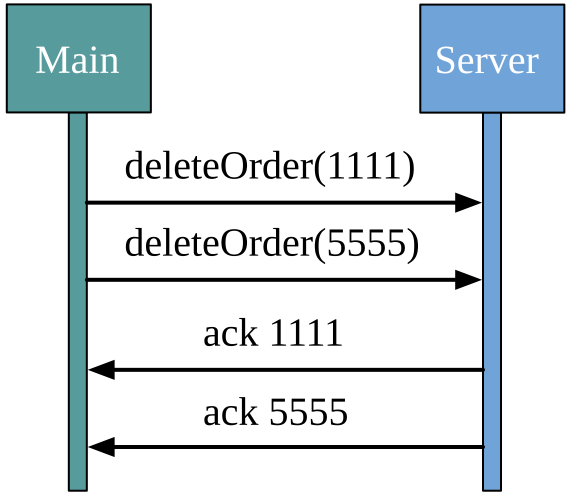
For comprehension cannot be done concurrently

```
def delete2Orders(orderId1: OrderId, orderId2: OrderId): IO[Unit] =
  for {
    ackOrder1 <- deleteOrder(orderId1)
    ackOrder2 <- deleteOrder(orderId2)
  } yield ()
```



Concurrent execution

```scala
def parExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] = ???


def delete2Orders(orderId1: OrderId, orderId2: OrderId): IO[Unit] =
  parExec(deleteOrder(orderId1), deleteOrder(orderId2))
```

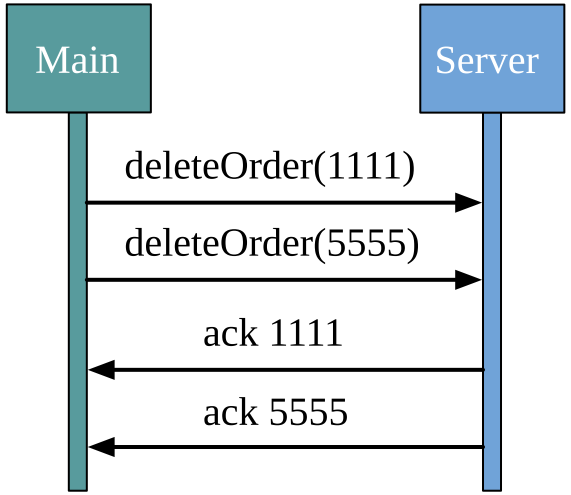


parExec is loosely defined

```
def parExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] =
  io1

def parExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] =
  io2

def parExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] =
  for {
    _ <- io1
    _ <- io2
  } yield ()

def parExec(io1: IO[Unit], io2: IO[Unit]): IO[Unit] =
  IO.succeed(())
```

Parametricity

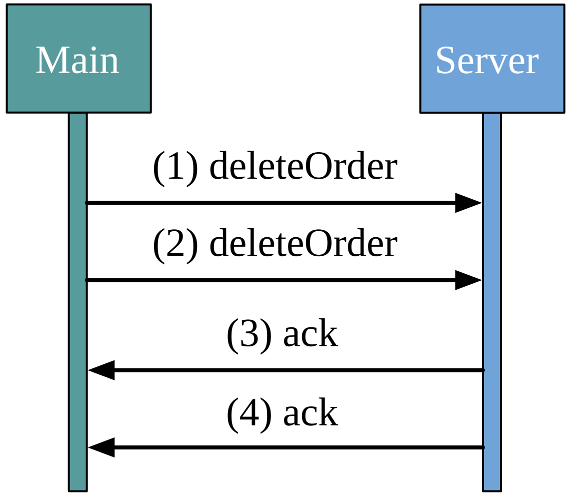
```
def parMap2[A, B, C](fa: IO[A], fb: IO[B])
                    (f: (A, B) => C): IO[C] = ???

def delete2Orders(orderId1: OrderId, orderId2: OrderId): IO[Unit] =
  parMap2(
    deleteOrder(orderId1),
    deleteOrder(orderId2)
  )((_,_) => ())
```



How concurrency is done with Future?

Future

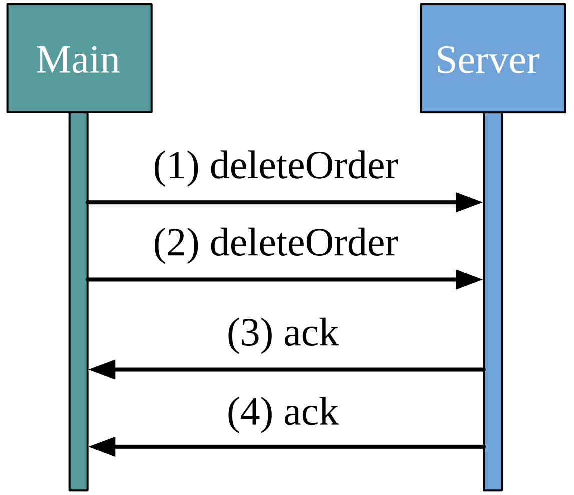
```scala
import scala.concurrent.{ExecutionContext, Future}

def deleteOrder(orderId: OrderId)
  (implicit ec: ExecutionContext): Future[Unit] =
  Future { ??? }

def delete2Orders(
  orderId1: OrderId,
  orderId2: OrderId
)(implicit ec: ExecutionContext): Future[Unit] = {

  val delete1: Future[Unit] = deleteOrder(orderId1) // (1) side effect
  val delete2: Future[Unit] = deleteOrder(orderId2) // (2) side effect

  for {
    _ /* (3) */ <- delete1
    _ /* (4) */ <- delete2
  } yield ()
}
```

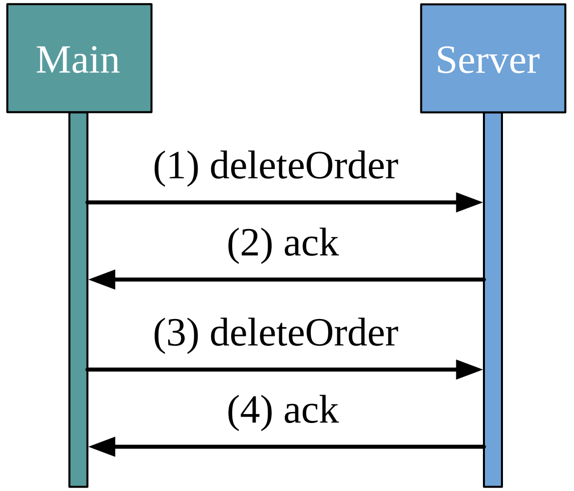


Creating a Future is not Pure

```scala
def deleteOrdersConcurrent(orderId1: OrderId,orderId2: OrderId)
  (implicit ec: ExecutionContext): Future[Unit] = {

  val delete1 = deleteOrder(orderId1) // (1)
  val delete2 = deleteOrder(orderId2) // (2)

  for {
    _ /* (3) */ <- delete1
    _ /* (4) */ <- delete2
  } yield ()
}
```

```scala
def deleteOrdersSequential(orderId1: OrderId,orderId2: OrderId)
  (implicit ec: ExecutionContext): Future[Unit] =
  for {
    _ /* (2) */ <- deleteOrder(orderId1) // (1)
    _ /* (4) */ <- deleteOrder(orderId2) // (3)
  } yield ()
```

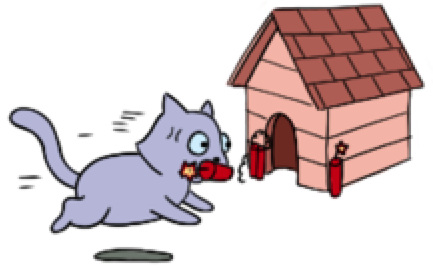




[@impurepics](https://twitter.com/impurepics)

FUTURE

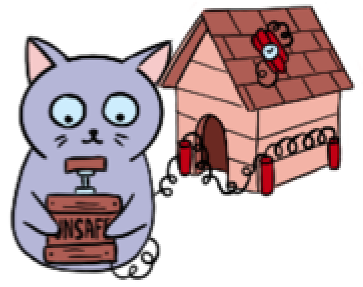
1. CREATE YOUR FUTURES


2. WIRE THEM TOGETHER

3. OOPS! SEEMS LIKE WE FORGOT SMTH

IO

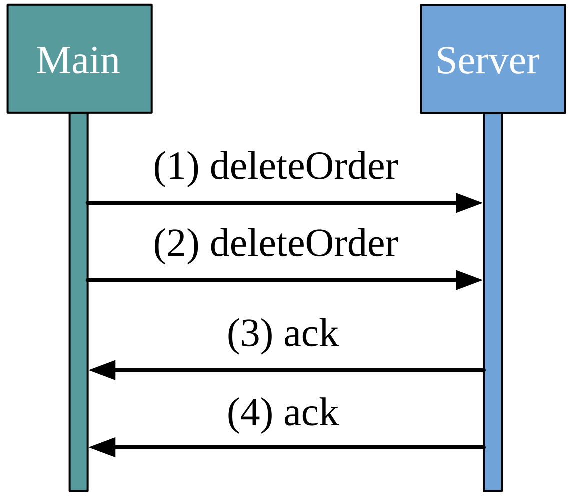
1. CREATE YOUR IOS


2. WIRE THEM TOGETHER

3. PROFIT!

Execution Context

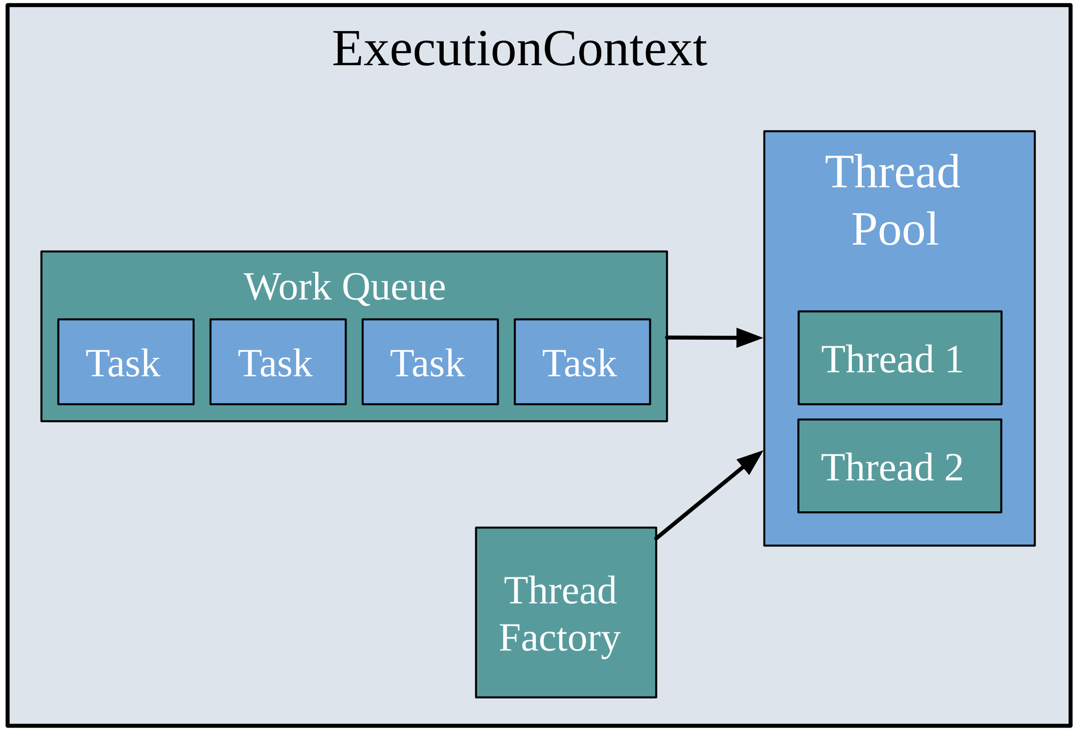
```scala
import scala.concurrent.{ExecutionContext, Future}

def deleteOrder(orderId: OrderId)(ec: ExecutionContext): Future[Unit] =
  Future { ??? }(ec)

def delete2Orders(
  orderId1: OrderId,
  orderId2: OrderId
)(ec: ExecutionContext): Future[Unit] = {

  val delete1 = deleteOrder(orderId1)(ec) // (1) side effect
  val delete2 = deleteOrder(orderId2)(ec) // (2) side effect

  delete1.flatMap(_ =>         // (3)
    delete2.map(_ => ())(ec) // (4)
  )(ec)
}
```



Execution Context

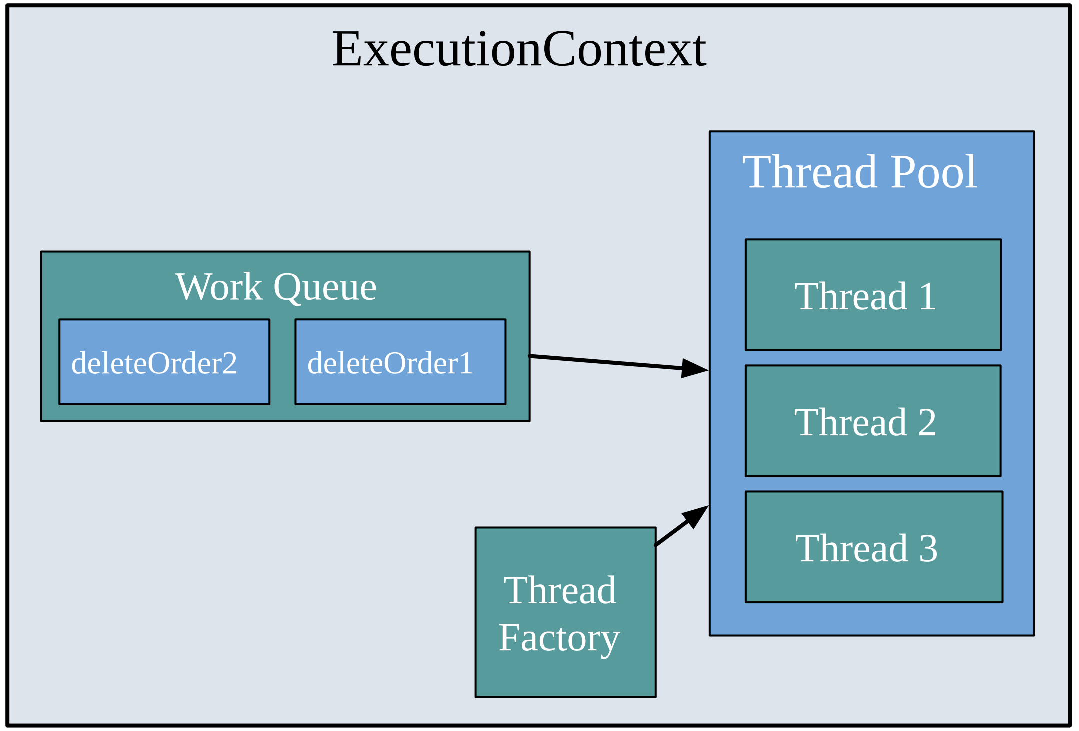
```scala
import java.util.concurrent.Executors
import scala.concurrent.ExecutionContext

val factory = threadFactory("test")
val pool = Executors.newFixedThreadPool(2, factory)
val ec = ExecutionContext.fromExecutorService(pool)

var x: Int = 0

val inc: Runnable = new Runnable {
  def run(): Unit = x += 1
}
```

```
scala> x
res4: Int = 0

scala> (1 to 10).foreach(_ => ec.execute(inc))

scala> x
res6: Int = 10
```



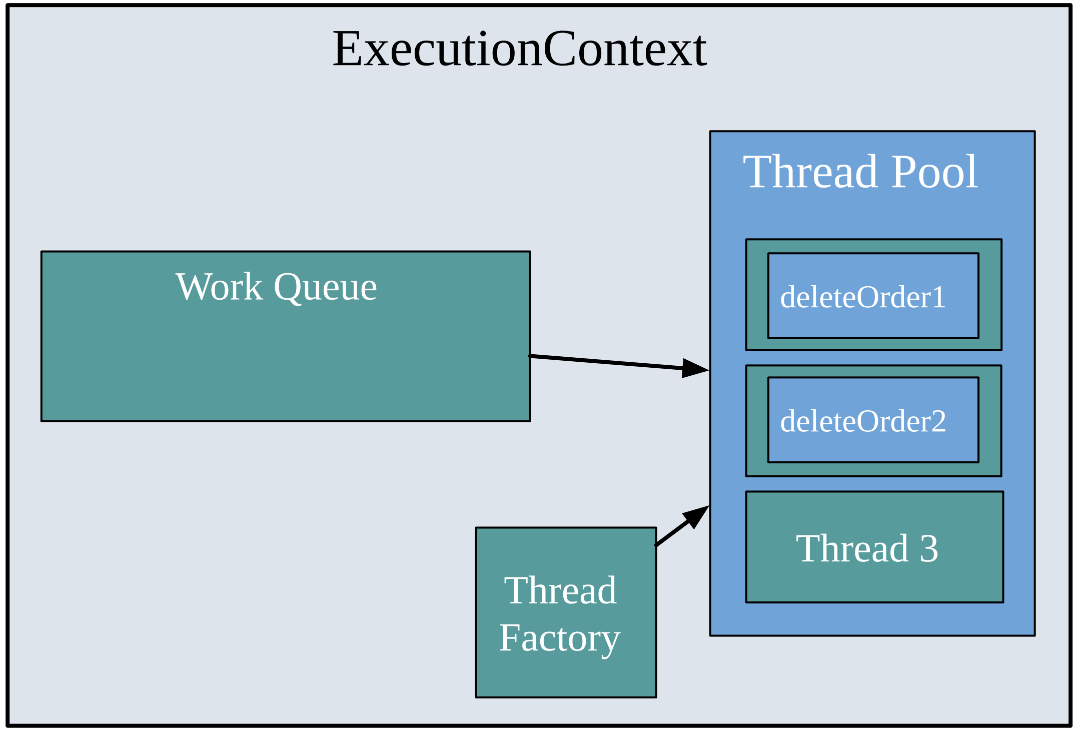
Execution Context

```
def delete2Orders(
  orderId1: OrderId,
  orderId2: OrderId
)(ec: ExecutionContext): Future[Unit] = {

  val delete1 = deleteOrder(orderId1)(ec) // (1)
  val delete2 = deleteOrder(orderId2)(ec) // (2)

  delete1.flatMap(_ =>       // (3)
    delete2.map(_ => ())(ec) // (4)
  )(ec)
}
```



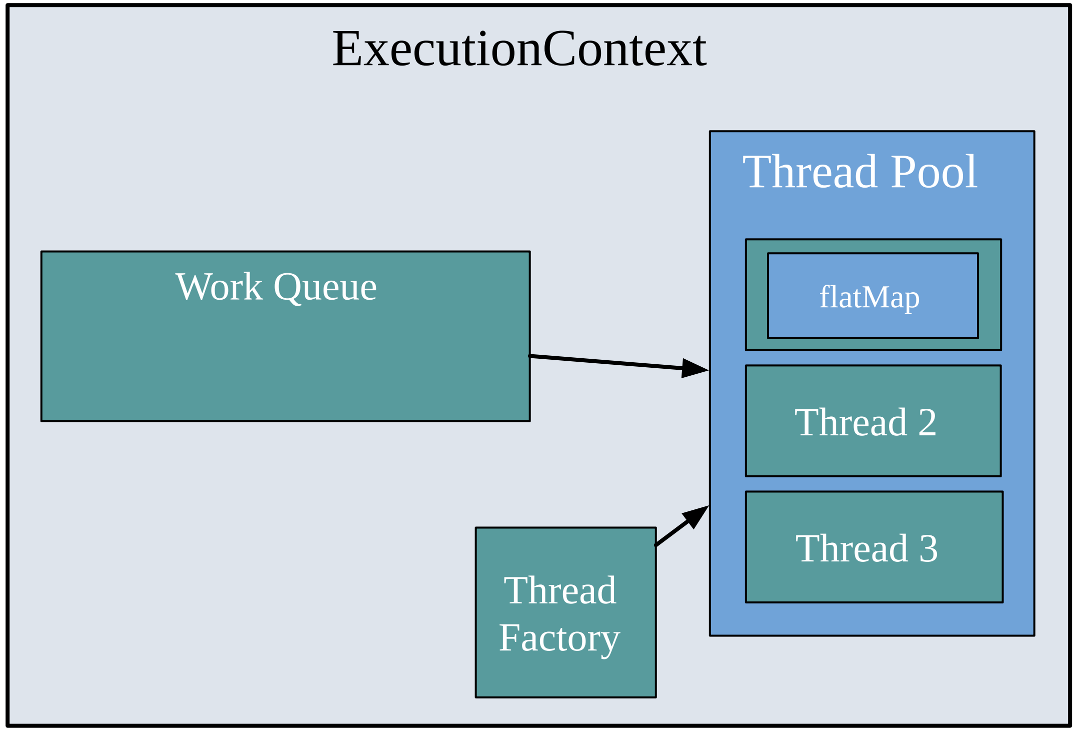
Execution Context

```
def delete2Orders(
  orderId1: OrderId,
  orderId2: OrderId
)(ec: ExecutionContext): Future[Unit] = {

  val delete1 = deleteOrder(orderId1)(ec) // (1)
  val delete2 = deleteOrder(orderId2)(ec) // (2)

  delete1.flatMap(_ =>       // (3)
    delete2.map(_ => ())(ec) // (4)
  )(ec)
}
```



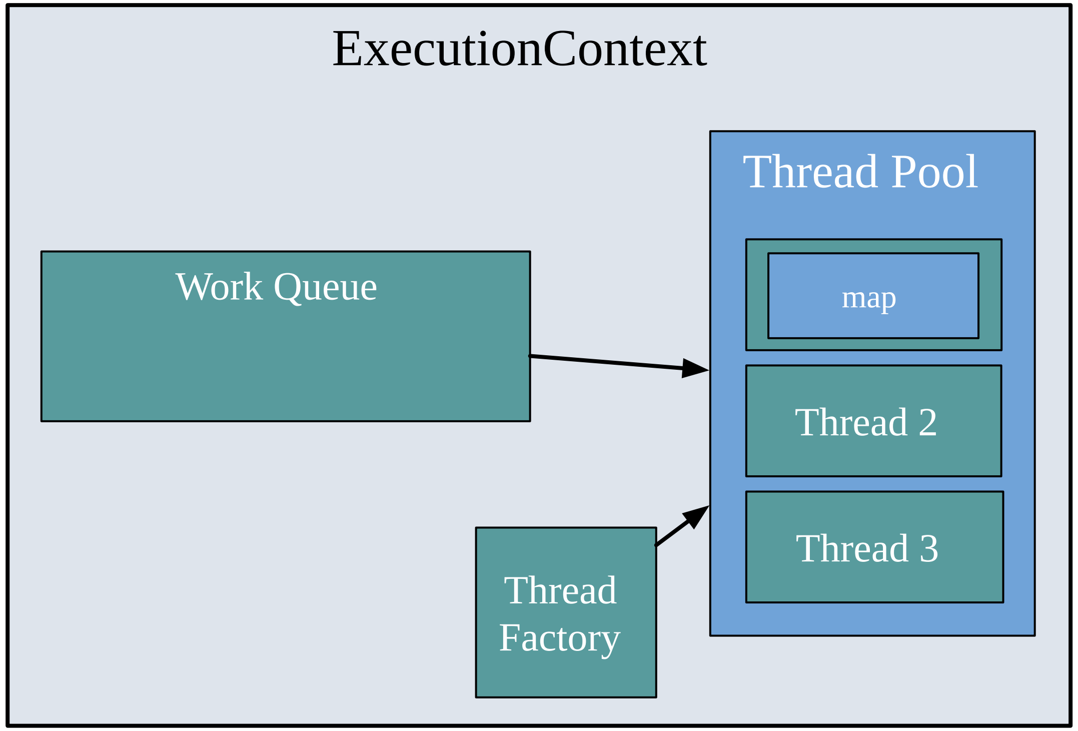
Execution Context

```
def delete2Orders(
  orderId1: OrderId,
  orderId2: OrderId
)(ec: ExecutionContext): Future[Unit] = {

  val delete1 = deleteOrder(orderId1)(ec) // (1)
  val delete2 = deleteOrder(orderId2)(ec) // (2)

  delete1.flatMap(_ =>         // (3)
    delete2.map(_ => ())(ec) // (4)
  )(ec)
}
```



Execution Context

```
def delete2Orders(
  orderId1: OrderId,
  orderId2: OrderId
)(ec: ExecutionContext): Future[Unit] = {

  val delete1 = deleteOrder(orderId1)(ec) // (1)
  val delete2 = deleteOrder(orderId2)(ec) // (2)

  delete1.flatMap(_ =>        // (3)
    delete2.map(_ => ())(ec) // (4)
  )(ec)
}
```



How can we adapt Future behaviour to pure IO?

Concurrent IO

```scala
trait IO[+A] {
  def start(ec: ExecutionContext): ???
}
```

Discuss with your neighbour 3-4 min

Concurrent IO

```
trait IO[+A] {
  def start(ec: ExecutionContext): IO[???]
}
```

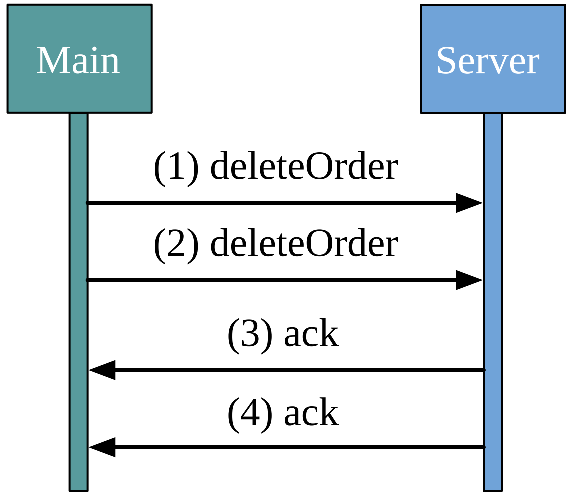
Concurrent IO

```
trait IO[+A] {
  def start(ec: ExecutionContext): IO[IO[A]]
}
```

Concurrent IO: parMap2

```
trait IO[+A] {
  def start(ec: ExecutionContext): IO[IO[A]]
}
```

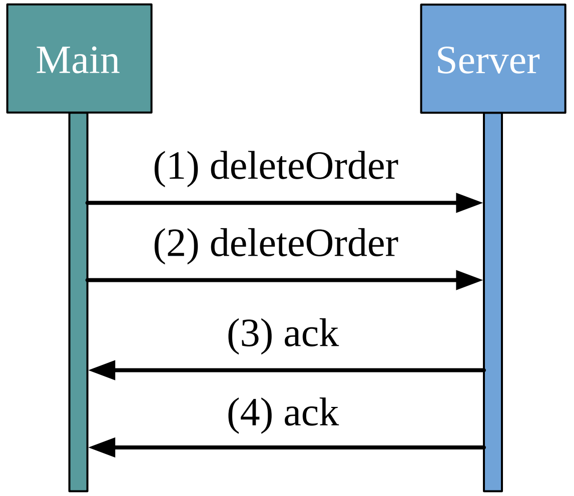
```
def parMap2[A, B, C](
  fa: IO[A],
  fb: IO[B]
)(f: (A, B) => C)(ec: ExecutionContext): IO[C] = ???
```



Concurrent IO: parMap2

```scala
trait IO[+A] {
  def start(ec: ExecutionContext): IO[IO[A]]
}
```

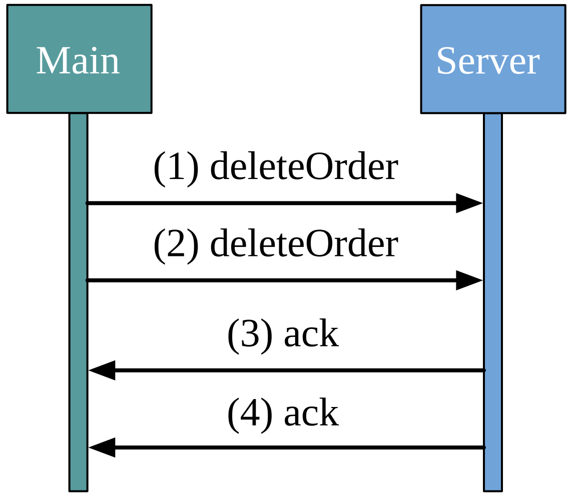
```scala
def parMap2[A, B, C](
  fa: IO[A],
  fb: IO[B]
)(f: (A, B) => C)(ec: ExecutionContext): IO[C] =
  for {
    awaitForA <- fa.start(ec) // (1)
    awaitForB <- fb.start(ec) // (2)
    a         <- awaitForA    // (3)
    b         <- awaitForB    // (4)
  } yield f(a, b)
```



Concurrent IO is referentially transparent

```
trait IO[+A] {
  def start(ec: ExecutionContext): IO[IO[A]]
}
```

```
def parMap2[A, B, C](
  fa: IO[A],
  fb: IO[B]
)(f: (A, B) => C)(ec: ExecutionContext): IO[C] = {

  val asyncIOA = fa.start(ec)
  val asyncIOB = fb.start(ec)

  for {
    awaitForA <- asyncIOA     // (1)
    awaitForB <- asyncIOB     // (2)
    a         <- awaitForA    // (3)
    b         <- awaitForB    // (4)
  } yield f(a, b)
}
```



Concurrent IO with Async

```scala
type Callback[-A] = Either[Throwable, A] => Unit

sealed trait IO[+A]

object IO {
  case class Thunk[+A](f: () => A) extends IO[A]

  case class Async[+A](f: Callback[A] => Unit, ec: ExecutionContext) extends IO[A]
}
```

Concurrent IO with Async

```scala
type Callback[-A] = Either[Throwable, A] => Unit

sealed trait IO[+A]

object IO {
  case class Thunk[+A](f: () => A) extends IO[A]

  case class Async[+A](f: Callback[A] => Unit, ec: ExecutionContext) extends IO[A]
}
```

An IO is either a Thunk or a Async computation with a CallBack

Concurrent IO with Async

```scala
type Callback[-A] = Either[Throwable, A] => Unit

sealed trait IO[+A]

object IO {
  case class Thunk[+A](f: () => A) extends IO[A]

  case class Async[+A](f: Callback[A] => Unit, ec: ExecutionContext) extends IO[A]
}
```

An IO is either a Thunk or a Async computation with a CallBack

More details in How do Fibers work from Fabio Labella

IO Async Exercises

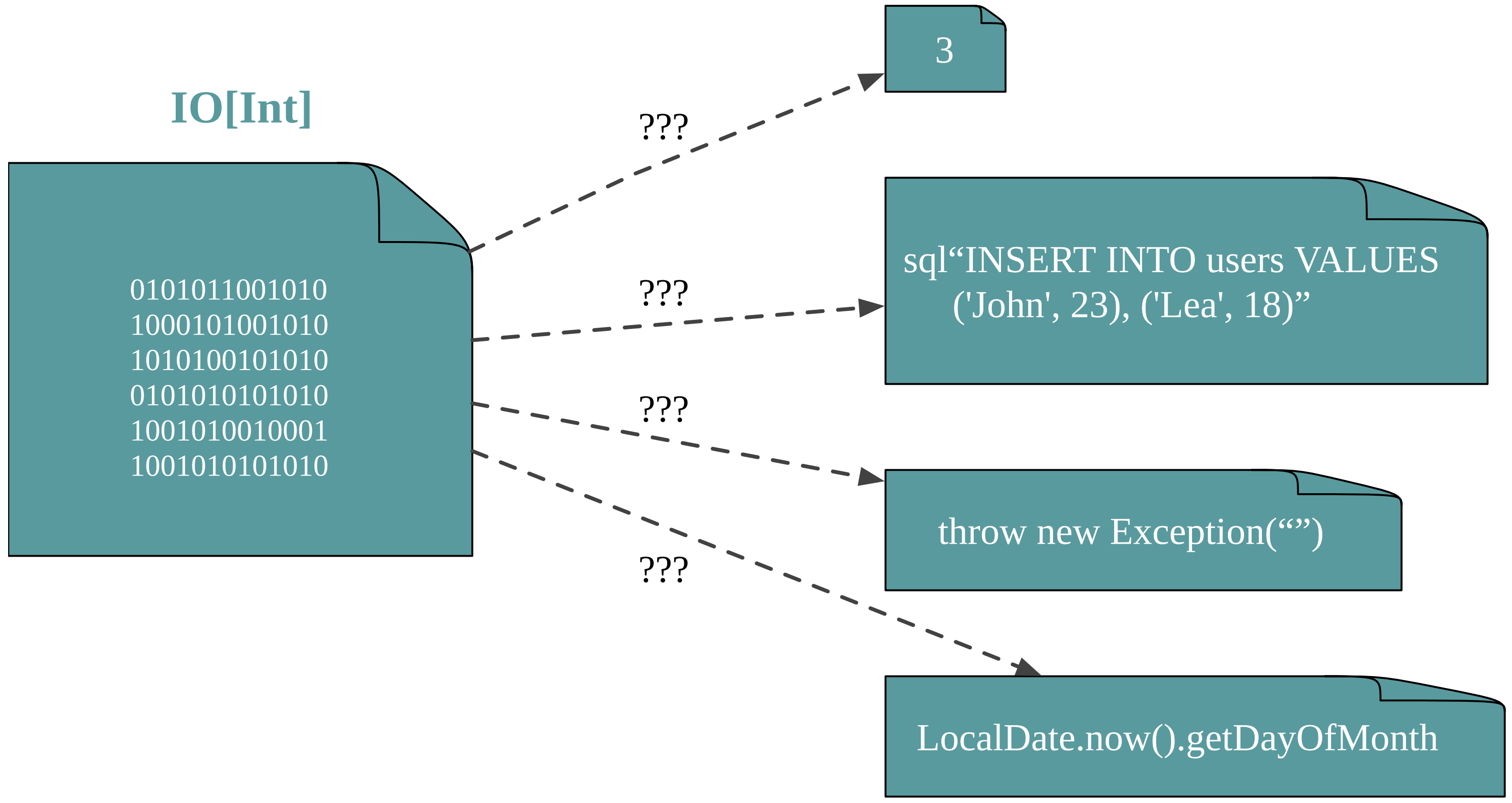
`exercises.sideeffect.IOAsyncExercises.scala`

Libraries do much more

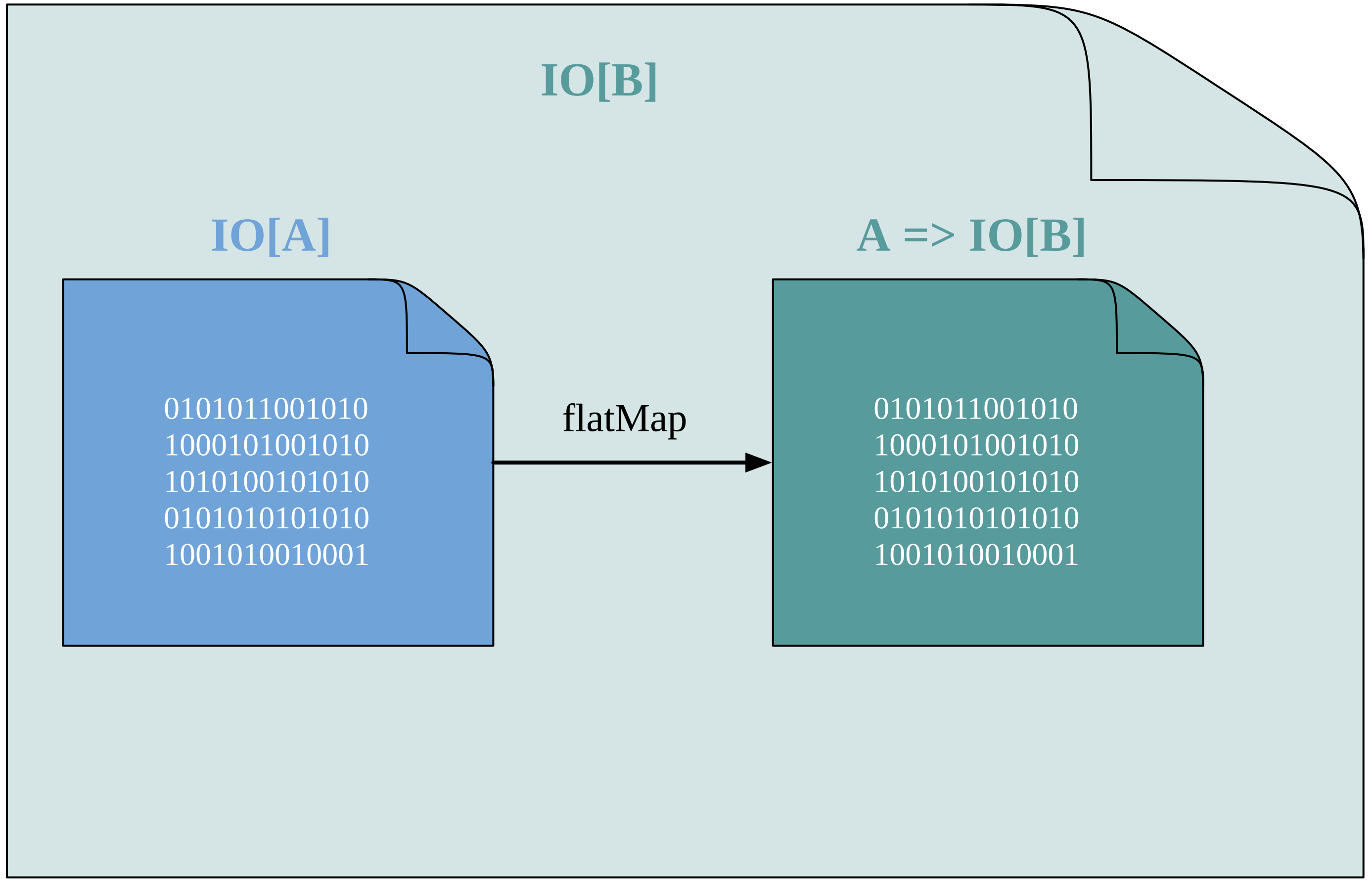
- Stack safety and JVM optimisation
- Cancellation, e.g. race two IO and cancel the loser
- Safe resource shutdown, e.g. close file, shutdown server
- Efficient Timer, retry utilities
- Help to chose right thread pool for different type of work: blocking, compute, dispatcher

What are the limitations of IO?

IO cannot be introspected



IO cannot be introspected



How can we encode side effects more precisely?

Warning: this is an advanced technique

Effect Algebra

```scala
sealed trait Description[A]

object Description {
  case object Today                          extends Description[LocalDate]
  case class  FetchString(url: String)       extends Description[String]
  case class  WriteLine(message: String) extends Description[Unit]
}
```

Effect Algebra

```scala
sealed trait Description[A]

object Description {
  case object Today                        extends Description[LocalDate]
  case class  FetchString(url: String)     extends Description[String]
  case class  WriteLine(message: String)   extends Description[Unit]
}
```

```scala
import Description._

def unsafeRun[A](fa: Description[A]): A =
  fa match {
    case Today            => LocalDate.now()
    case FetchString(url) => Source.fromURL(url).mkString("")
    case WriteLine(msg)   => println(msg)
  }
```

Effect Algebra

```scala
object Main extends App {

  val description: Description[Unit] = WriteLine("Hello World")

  unsafeRun(description)

}
```

```
scala> Main.main(Array.empty)
Hello World
```

Interpret algebra in different ways

```
def testInterpreter[A](fa: Description[A]): A =
  fa match {
    case Today            => LocalDate.of(2019, 8, 17)
    case FetchString(url) => "Hello World"
    case WriteLine(msg)   => ()
  }
```

```
def loggerInterpreter[A](fa: Description[A]): String =
  fa match {
    case Today            =>  "call Today"
    case FetchString(url) => s"call FetchString for $url"
    case WriteLine(msg)   => s"call WriteLine with $msg"
  }
```

How to add new descriptions?

How to combine description together?

How to add new descriptions?

```scala
sealed trait Description[A]
object Description {
  case object Today                          extends Description[LocalDate]
  case class  FetchString(url: String)       extends Description[String]
  case class  WriteLine(message: String) extends Description[Unit]
}
```

How to add new descriptions?

```scala
sealed trait Description[A]
object Description {
  case object Today                       extends Description[LocalDate]
  case class  FetchString(url: String)    extends Description[String]
  case class  WriteLine(message: String)  extends Description[Unit]
}
```

1. Add primitive (□ not really scalable)

```scala
case object FetchJson extends Description[Json]
```

How to add new descriptions?

```scala
sealed trait Description[A]
object Description {
  case object Today                        extends Description[LocalDate]
  case class  FetchString(url: String)     extends Description[String]
  case class  WriteLine(message: String) extends Description[Unit]
}
```

1. Add primitive (□ not really scalable)

```scala
case object FetchJson extends Description[Json]
```

2. Transform existing actions (□ composable)

```scala
FetchString.map(parseJson)
```

Problem

```scala
sealed trait Description[A]

object Description {
  case object Today                          extends Description[LocalDate]
  case class  FetchString(url: String)       extends Description[String]
  case class  WriteLine(message: String) extends Description[Unit]
}
```

```scala
import Description._

def map[A, B](fa: Description[A])(f: A => B): Description[B] =
  fa match {
    case Today            => ???
    case FetchString(url) => ???
    case WriteLine(msg)   => ???
  }
```

Free structures (brief introduction)

```
sealed trait FreeMap[A]

object FreeMap {
  case class Map[X, A](description: Description[X], update: X => A) extends FreeMap[A]
}
```

Free structures (brief introduction)

```
sealed trait FreeMap[A]

object FreeMap {
  case class Map[X, A](description: Description[X], update: X => A) extends FreeMap[A]
}
```

```
import io.circe.Json

def parseJson(x: String): Json =
  io.circe.parser.parse(x).getOrElse(Json.obj())

def fetchJson(url: String): FreeMap[Json] =
  Map(FetchString(url), parseJson)
```

Free structures

```scala
sealed trait FreeMap[A] {
  def map[C](f: A => C): FreeMap[C]
}

object FreeMap {
  def lift[A](description: Description[A]): FreeMap[A] =
    Map(description, identity[A])

  case class Map[X, A](description: Description[X], update: X => A) extends FreeMap[A] {
    def map[C](f: A => C): FreeMap[C] = Map(description, update andThen f)
  }
}
```

```scala
def fetchString(url: String): FreeMap[String] = FreeMap.lift(FetchString(url))

def fetchJson(url: String)   : FreeMap[Json]   = fetchString(url).map(parseJson)
```

Free structures

1. Primitives

```scala
val today                     : FreeMap[LocalDate] = FreeMap.lift(Today)
def fetchString(url: String): FreeMap[String]    = FreeMap.lift(FetchString(url))
def writeLine(msg: String)  : FreeMap[Unit]      = FreeMap.lift(WriteLine(msg))
```

2. Derived description

```scala
def fetchJson(url: String): FreeMap[Json] = fetchString(url).map(parseJson)
```

Free structures

3. Interpreters

```scala
def unsafeRun[A](fa: Description[A]): A =
  fa match {
    case Today            => LocalDate.now()
    case FetchString(url) => Source.fromURL(url).mkString("")
    case WriteLine(msg)   => println(msg)
  }

def unsafeRunFree[A](fa: FreeMap[A]): A =
  fa match {
    case Map(fa, f) => f(unsafeRun(fa))
  }
```

Tadam!

```
object Main extends App {

  val description: FreeMap[Json] = fetchJson("https://api.github.com/users/julien-truffaut/orgs")

  println(unsafeRunFree(description))

}
```

```
scala> Main.main(Array.empty)
[
  {
    "login" : "fp-tower",
    "id" : 50878186,
    "node_id" : "MDEyOk9yZ2FuaXphdGlvbjUwODc4MTg2",
    "url" : "https://api.github.com/orgs/fp-tower",
    "repos_url" : "https://api.github.com/orgs/fp-tower/repos",
    "events_url" : "https://api.github.com/orgs/fp-tower/events",
    "hooks_url" : "https://api.github.com/orgs/fp-tower/hooks",
    "issues_url" : "https://api.github.com/orgs/fp-tower/issues",
    "members_url" : "https://api.github.com/orgs/fp-tower/members{/member}",
    "public_members_url" : "https://api.github.com/orgs/fp-tower/public_members{/member}",
    "avatar_url" : "https://avatars1.githubusercontent.com/u/50878186?v=4",
    "description" : ""
  },
  {
    "login" : "typelevel",
    "id" : 3731824
```

Free translates functions to data structures (GADT)

```
def         readLine :                  String
case object ReadLine extends Description[String]

def         writeLine(message: String) :                   Unit
case object WriteLine(message: String) extends Description[Unit]

def        map[X, A](action: Description[X], update: X => A) :       Description[A]
case class Map[X, A](action: Description[X], update: X => A) extends     FreeMap[A]
```

Algebra Exercises

`exercises.sideeffect.AlgebraExercises.scala`

Free Summary

- Free translates code into data
- Easy to interpret an algebra in many ways (log, test, real, metrics)
- Complex (GADT, natural transformation, Coproduct, ...)
- Can miss some features from target effect like parallel execution, resource handling

All problems in computer science can be solved by another level of indirection

David Wheeler

Free is several orders of magnitude more complex than IO

Resources and further study

- Seven Sketches in Compositionality: An Invitation to Applied Category Theory
- Constraints Liberate, Liberties Constrain
- How do Fibers work

Module 3: Error Handling