Data, and when not to use it

Who I am

Edmund Noble

Maintainer of typelevel/cats, typelevel/cats-mtl, atnos-org/eff

Contributor to Scalaz

Developer at SlamData

Come by Gitter and say hi!

What this talk is about

The uses of typical data structures in the purely functional and imperative traditions

Misconceptions about data structures in pure FP which can follow you while transitioning between paradigms

Most importantly, when to use data structures at all

Types, and types

Object-functional code in Scala tends to have two kinds of types treated separately

Open types (interfaces) and closed types (data types)

Openness refers to subtyping which isn’t directly relevant to the talk, what I’m trying to capture here is the distinction between data and interfaces

What is a data type?

Data types in Scala are sums of products (we're excluding functions)

One sealed trait, multiple case classes (sums) each built out of several values (products)

Sometimes recursive, one case class mentions the sealed trait

Commonly-used data types

```
sealed trait Either[+L, +R]

case class Right[+R](r: R) extends Either[Nothing, R]

case class Left[+L](l: L) extends Either[L, Nothing]
```

Either, popular data type for describing failing computations

Problem description: divide two numbers then take the logarithm of the result, base 10

```scala
case class DividedByZero(dividend: Int)
def divide(dividend: Int, divisor: Int): DividedByZero Either Int =
 if (divisor == 0) Left(DividedByZero(dividend))
 else Right(dividend / divisor)

case class NoLogarithm(number: Double)
def log10(number: Double): NoLogarithm Either Double =
 if (number <= 0) Left(NoLogarithm(number))
 else Right(Math.log10(number))
```

Typical usage of Either, two error types for two error meanings

```scala
sealed trait DividedByZeroOrNoLogarithm
case class LDividedByZero(dividend: Int) extends DividedByZeroOrNoLogarithm
case class LNoLogarithm(number: Double) extends DividedByZeroOrNoLogarithm
def divideAndLog(dividend: Int, divisor: Int): DividedByZeroOrNoLogarithm Either Double =
  for {
    divideResult <- divide(dividend, divisor).leftMap {
      case DividedByZero(dividend) =>
        LDividedByZero(dividend)
    }
    logResult <- log10(divideResult).leftMap {
      case NoLogarithm(number) =>
        LNoLogarithm(number)
    }
  } yield logResult
```

```scala
  type DividedByZeroOrNoLogarithm = DividedByZero Either NoLogarithm
  def divideAndLog(dividend: Int, divisor: Int): DividedByZeroOrNoLogarithm Either Double = for {
    divideResult <- divide(dividend, divisor).leftMap(Left(_))
    logResult <- log10(divideResult).leftMap(Right(_))
  } yield logResult
//    L1 Either L2 => (L1 Either L2) Either L3 ===> easy!
//    L1 Either L3 => (L1 Either L2) Either L3 ===> not so easy!
```

```scala
type DividedByZeroOrNoLogarithm[A] = DividedByZero Either (NoLogarithm Either A)

def divideAndLog(dividend: Int, divisor: Int): DividedByZeroOrNoLogarithm[Double] = {
  divide(dividend, divisor).map(log10(_))
}
```

An aside on monads

First and second options call flatMap

Last option has a type so precise it describes the order of operations

flatMap destroys type information

```
sealed trait SVG
case class DrawText(text: String, x: Double, y: Double) extends SVG
case class DrawEllipse(x: Double, y: Double, rx: Double, ry: Double) extends SVG
case class DrawCircle(x: Double, y: Double, r: Double) extends SVG
case class DrawRect(x: Double, y: Double, w: Double, h: Double) extends SVG
```

```
sealed trait SVG
case class DrawText(text: String, x: Double, y: Double, next: SVG) extends SVG
case class DrawEllipse(x: Double, y: Double, rx: Double, ry: Double, next: SVG) extends SVG
case class DrawCircle(x: Double, y: Double, r: Double, next: SVG) extends SVG
case class DrawRect(x: Double, y: Double, w: Double, h: Double, next: SVG) extends SVG
case object DrawNothing extends SVG
```

```scala
def svgProgram(xZero: Double, yZero: Double): List[SVG] = {
 val radius = 10
 DrawText("Scala World 2017", xZero, yZero) ::
 DrawCircle(xZero - radius, yZero - radius, radius) ::
 Nil
}
```

```scala
def printSVG(svg: List[SVG]): String = svg match {
 case DrawText(t, x, y) :: ss => s"Text($t, $x, $y)\n" + printSVG(ss)
 case DrawEllipse(x, y, rx, ry) :: ss => s"Ellipse($x, $y, $rx, $ry)\n" + printSVG(ss)
 case DrawCircle(x, y, r) :: ss => s"Circle($x, $y, $r)\n" + printSVG(ss)
 case DrawRect(x, y, w, h) :: ss => s"Rect($x, $y, $w, $h)\n" + printSVG(ss)
 case Nil => ""
}
```

```scala
sealed trait RandomAccess[A]
case class Write(index: Int, data: String) extends RandomAccess[Unit]
case class Read(index: Int, length: Int) extends RandomAccess[String]
```

```scala
sealed trait RandomAccess
case class Write(index: Int, data: String) extends RandomAccess
case class Read[A](index: Int, length: Int, fromData: String => A) extends RandomAccess
```

```scala
sealed trait RandomAccess[A]
case class Write[A](index: Int, data: String) extends RandomAccess[A]
case class Read[A](index: Int, length: Int, fromData: String => A) extends RandomAccess[A]
```

```scala
case class Write[A](index: Int, data: String, nextInstruction: RandomAccess[A]) extends RandomAccess[A]
```

```scala
case class Read[A](index: Int, length: Int, doWithData: String => RandomAccess[A]) extends RandomAccess[A]
```

```scala
case class End[A](value: A) extends RandomAccess[A]
```

```scala
sealed trait RandomAccess[A]
case class Write[A](index: Int, data: String, nextInstruction: RandomAccess[A]) extends RandomAccess[A]
case class Read[A](index: Int, length: Int, doWithData: String => RandomAccess[A]) extends RandomAccess[A]
case class End[A](value: A) extends RandomAccess[A]
```

```
def dupe(index: Int, length: Int): Free[RandomAccess, Unit] = for {
 data <- Free.liftF(Read(index, length))
 _ <- Free.liftF(Write(index + length, length, data))
} yield ()

def interpret: RandomAccess ~> StateT[Either[IndexOutOfBoundsException, ?], String, ?] =
  Lambda[RandomAccess ~> StateT[Either[IndexOutOfBoundsException, ?], String, ?]].apply {
    case Write(i, l, d) => StateT[Either[IndexOutOfBoundsException, ?], String, Unit](s =>
      if (s.length < i + l) (new IndexOutOfBoundsException).left
      else s.substring(0, i) + d + s.substring(i + l))
    case Read(i, l) => StateT[Either[IndexOutOfBoundsException, ?], String, String](s =>
      if (s.length < i + l) (new IndexOutOfBoundsException).left
      else s.substring(i, i + l - 1))
}
```

```scala
case class EitherK[F[_], G[_], A](value: F[A] Either G[A])
```

Commonalities

Solutions

```
type DividedByZeroOrNoLogarithm = DividedByZero Either NoLogarithm
trait Inject[E, S] {
  def apply(s: S): E
}

def divideAndLog[E](dividend: Int, divisor: Int)
                   (implicit div: Inject[E, DividedByZero], log: Inject[E, NoLogarithm]): E Either Double =
  for {
    divideResult <- divide(dividend, divisor).leftMap(div(_))
    logResult <- log10(divideResult).leftMap(log(_))
  } yield logResult
```

```scala
trait InjectK[E[_], S[_]] { def apply[A](s: S[A]): E[A] }
```

Data types in Scala

Duality of types

```scala
def produceInt: Int = 1
def consumeIntConsumer[A](consumer: Int => A): A = consumer(1)
```

Church Encoding

```
def divide[A](dividend: Int, divisor: Int, dividedByZero: Int => A, result: Int => A): A =
 if (divisor == 0) dividedByZero(dividend)
 else result(dividend / divisor)
```

```
def log10[A](number: Double, noLogarithm: Double => A, result: Double => A): A =
 if (number <= 0) noLogarithm(number)
 else result(Math.log10(number))
```

```
def divideAndLog[A](dividend: Int, divisor: Int,
                    dividedByZero: Int => A, noLogarithm: Double => A,
                    result: Double => A): A = {
 divide(dividend, divisor, dividedByZero, log10(_, noLogarithm, result))
```

Monads and continuations

```
def divide[F[_]: Monad](dividend: Int, divisor: Int, dividedByZero: Int => F[Int]): F[Int] =
 if (divisor == 0) dividedByZero(dividend)
 else (dividend / divisor).pure[F]

def log10[F[_]: Monad](number: Double, noLogarithm: Double => F[Double]): F[Double] =
 if (number <= 0) noLogarithm(number)
 else Math.log10(number).pure[F]

def divideAndLog[F[_]: Monad](dividend: Int, divisor: Int,
                              dividedByZero: Int => F[Int], noLogarithm: Double => F[Double]): A = {
 divide(dividend, divisor, dividedByZero).flatMap(log10(_, noLogarithm))
}
```

$$(a \mathbin{++} b) \mathbin{++} c == a \mathbin{++} (b \mathbin{++} c)$$

$$a \mathbin{++} Nil == a$$

$$Nil \mathbin{++} a == a$$

```scala
trait SVG[A] {
 def drawText(text: String, x: Double, y: Double): A
 def drawEllipse(x: Double, y: Double, rx: Double, ry: Double): A
 def drawCircle(x: Double, y: Double, r: Double): A
 def drawRect(x: Double, y: Double, w: Double, h: Double): A
}
```

```
def svgProgram[A: Monoid](svg: SVG[A], xZero: Double, yZero: Double): A = {
 val radius = 10
 svg.drawText(“Scala World 2017”, xZero, yZero) |+|
 svg.drawCircle(xZero - radius, yZero - radius, radius)
}
```

```scala
val svgPrint: SVG[String] = new SVG[String] {
 def drawText(t: String, x: Double, y: Double) => s“Text($t, $x, $y)\n”
 def drawEllipse(x: Double, y: Double, rx: Double, ry: Double) => s”Ellipse($x, $y, $rx, $ry)\n”
 def drawCircle(x: Double, y: Double, r: Double) => s”Circle($x, $y, $r)\n”
 def drawRect(x: Double, y: Double, w: Double, h: Double) => s”Rect($x, $y, $w, $h)\n”
}
```

```scala
trait RandomAccess[F[_]] {
 def write(index: Int, data: String): F[Unit]
 def read(index: Int, length: Int): F[String]
}
def dupe[F[_]: Monad](ra: RandomAccess[F], index: Int, length: Int): F[Unit] =
  for {
    data <- ra.read(index, length)
    _ <- ra.write(index + length, length, data)
  } yield ()
```

```scala
val interpreter = new RandomAccess[StateT[Either[IndexOutOfBoundsException, ?], String, ?]] {
  def write(i: Int, l: Int, data: String) =
    StateT[Either[IndexOutOfBoundsException, ?], String, Unit](s =>
      if (s.length < i + l) (new IndexOutOfBoundsException).left
      else s.substring(0, i) + d + s.substring(i + l))

  def read(index: Int, length: Int) =
    StateT[Either[IndexOutOfBoundsException, ?], String, String](s =>
      if (s.length < i + l) (new IndexOutOfBoundsException).left
      else s.substring(i, i + l - 1))
}
```

Summing up

Free $\vdash$ Forgetful