Abstraction
in
F[_]

Marcus Henry, Jr.

@dreadedsoftware

Software Developer @integrichain

Materials & Shoutouts

Deck:

dreadedsoftware.com/s/tls2017_deck.pdf
dreadedsoftware.com/s/tls2017_deck.pptx

Code:

https://github.com/dreadedsoftware/talks/tree/master/tls2017

Shoutouts to:

Typelevel & NE Scala for the conference
Mr. Rob for his talk on Fix Point Types
Mr. Runar for his talk on Contraints and Liberties

```
trait Pipeline[F[_], A, B]{
  final def apply(uri: URI): F[Unit] = {
    val in = read(uri)
    val computed = convert(in, computation)
    convert(computed, write)
  }
  def read(uri: URI): F[A]
  def computation(in: A): B
  def write(in: B): Unit
  def convert[U, V](in: F[U], f: U => V): F[V]
}
```

Pipeline

```
def convert[U, V](in: F[U], f: U => V): F[V]
```

cats.Functor

```
def map    [A, B](fa: F[A])(f: A => B): F[B]
```

```
trait Pipeline[F[_], A, B]{
  final def apply(uri: URI)(implicit
    F: Functor[F]): F[Unit] = {
    val in = read(uri)
    val computed = F.map(in)(computation)
    F.map(computed)(write)
  }
  def read(uri: URI): F[A]
  def computation(in: A): B
  def write(in: B): Unit
}
```

```
trait Read[F[_], A] extends Function1[URI, F[A]]
trait Computation[A, B] extends Function1[A, B]
trait Write[B] extends Function1[B, Unit]

trait Pipeline[F[_], A, B]{
  final def apply(uri: URI)(implicit
    F: Functor[F],
    read: Read[F, A],
    computation: Computation[A, B],
    write: Write[B]): F[Unit] = {
    val in = read(uri)
    val computed = F.map(in)(computation)
    F.map(computed)(write)
  }
}
```

```
sealed trait Pipeline[F[_], A, B]{
  def apply(uri: URI): F[Unit]
}
object Pipeline{
  final def apply[F[_]: Functor, A, B](implicit
    read: Read[F, A],
    computation: Computation[A, B],
    write: Write[B]) = {
    val F: Functor[F] = implicitly
    new Pipeline[F, A, B]{
      override def apply(uri: URI): F[Unit] = {
        val in = read(uri)
        val computed = F.map(in)(computation)
        F.map(computed)(write)
      }
    }
  }
}
```

```
val pipeline1 = Pipeline[Stream, …]
val pipeline2 = Pipeline[Stream, …]
val pipeline3 = Pipeline[Stream, …]
val pipeline4 = ???//combine 1 2 and 3
pipeline4(uri)
```

```scala
sealed trait Pipeline[T, A, B]{
  type Out
  def apply(uri: URI): Out
}
object Pipeline{
  final def apply[T: Guard, F[_]: Functor, A, B](implicit
    read: Read[F, A],
    computation: Computation[A, B],
    write: Write[B]) = {
    val G: Guard[T] = implicitly
    val F: Functor[F] = implicitly
    new Pipeline[T, A, B]{
      type Out = Either[Unit, F[Unit]]
      override def apply(uri: URI): Out = if(uri.toString.contains(G.name)){
        val in = read(uri)
        val computed = F.map(in)(computation)
        Right(F.map(computed)(write))
      } else Left(())
    }
  }
}
```

For each Pipeline there is a Guard with a Constant

```
trait One
trait Two
trait Three
implicit val guard1 = new Guard[One]{
  override def name: String = “one”
}
implicit val guard2 = new Guard[Two]{
  override def name: String = “two”
}
implicit val guard3 = new Guard[Three]{
  override def name: String = “three”
}
```

```
val pipeline1 = Pipeline[One, …]
val pipeline2 = Pipeline[Two, …]
val pipeline3 = Pipeline[Three, …]
def perform(uri: URI) = {
  pipelineFib(uri).fold(
    _ => Left(pipelineTri(uri).fold(
      _ => Left(pipelineFac(uri).fold(
        _ => Left(()),
        a => Right(a)
      )),
      a => Right(a)
    )),
    a => Right(a)
  )
}
perform(uri)
```

HList

Heterogenous

Items can be of different Types

List

A recursive sequence

List

```
scala> 1 :: '1' :: 1.0 :: Nil
res2: List[AnyVal] = List(1, 1, 1.0)
```

Hlist

```
scala> 1 :: '1' :: 1.0 :: HNil
res3:
shapeless.::[Int,shapeless.::[Char,shapeless.::[Double,shapeless.HNil]]]
= 1 :: 1 :: 1.0 :: HNil
```

Coproduct

Categorical Dual

Arrows are Reversed

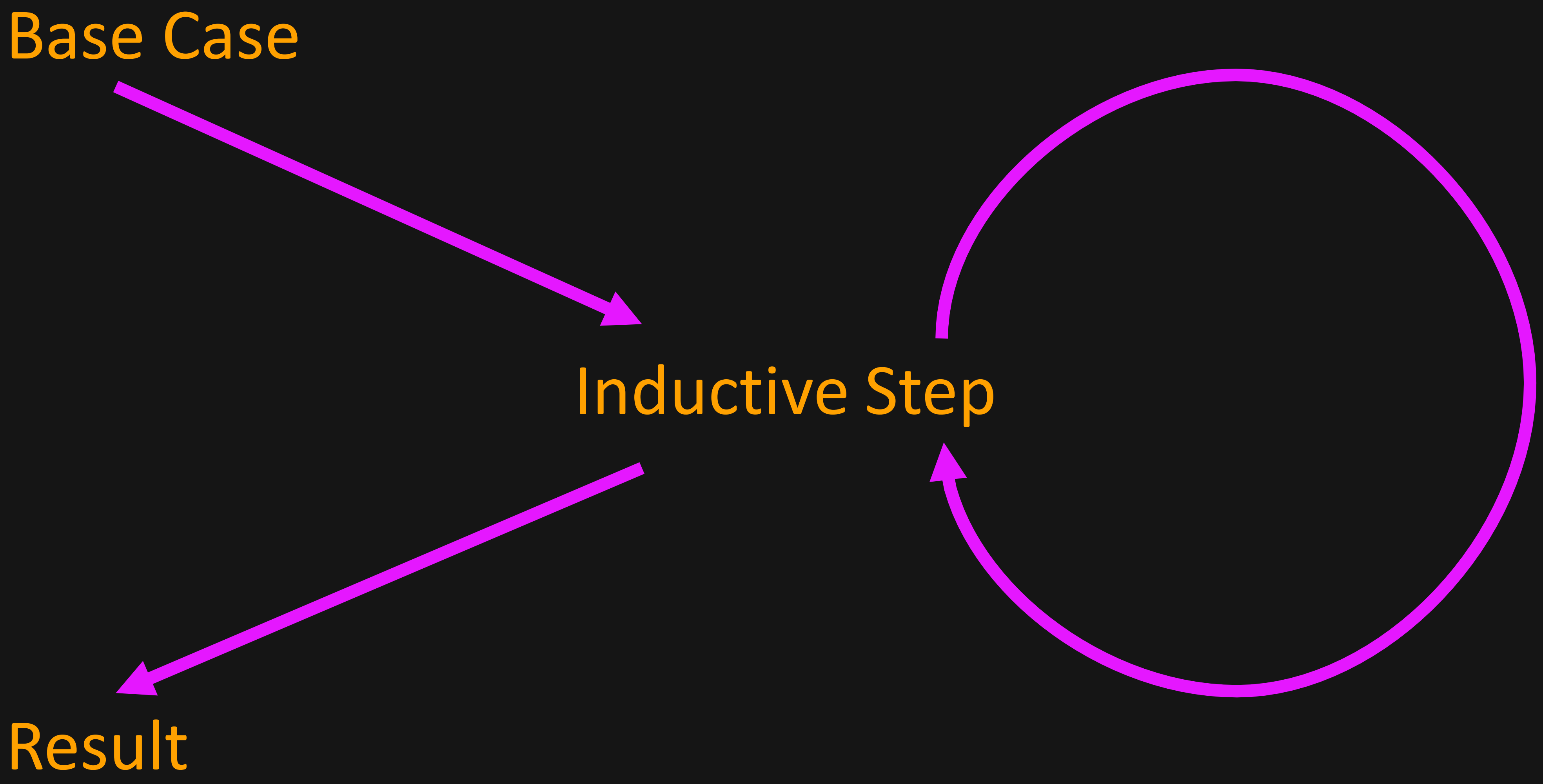
Product

All of these things

Any of these things

```
scala> type T = Int :+: Char :+: Double :+: CNil
defined type alias T
scala> val t: T = Inl(1)
t: T = Inl(1)
scala> val t: T = Inr(Inl('1'))
t: T = Inr(Inl(1))
scala> val t: T = Inr(Inr(Inl(1.0)))
t: T = Inr(Inr(Inl(1.0)))
```

Induction



Base Case

Inductive Step

Result

```
object Pipeline{
  type Aux[T, F[_], A, B, O] = Pipeline[T, F, A, B, O]{type Out = O}
  def apply[T: Guard, F[_]: Functor, A, B](implicit
    …): Aux[T, F, A, B, Either[Unit, F[Unit]]] = …
  implicit def PNil: Pipeline.Aux[CNil, CNil, CNil, Unit:+:CNil] = {
    new Pipeline[CNil, CNil, CNil]{
      type Out = Unit:+:Cnil
      override def apply(uri: URI): Out = Inl(())
    }
  }
  implicit def inductivePipeline[TH, F[_], AH, BH,
    TT <: Coproduct, AT <: Coproduct, BT <: Coproduct, OT <: Coproduct
  ](implicit
    head: Pipeline.Aux[TH, AH, BH, Either[Unit, F[Unit]]], /*H for Head*/
    tail: Pipeline.Aux[TT, AT, BT, OT] /*T for Tail*/): Pipeline.Aux[
      TH:+:TT, AH:+:AT, BH:+:BT, F[Unit]:+:OT] =
    new Pipeline[TH:+:TT, AH:+:AT, BH:+:BT]{
      override type Out = F[Unit]:+:OT
      override def apply(uri: URI): Out = {
        head(uri).fold(
          _ => Inr(tail(uri)),
          s => Inl(s)
        )
      }
    }
}
```

```
implicit val pipeline1 = Pipeline[One, …]
implicit val pipeline2 = Pipeline[Two, …]
implicit val pipeline3 = Pipeline[Three, …]
val pipeline4 = pipeline1 op pipeline2 op pipeline3 op PNil
pipeline4(uri)
```

Implicit Induction

HList

View Bounds

IsHCons

A Spike in Compilation Time

Implicit AnyVal

Implicit Conversion

An Operator

```
object Pipeline{
  type Aux[T, F[_], A, B, O] = Pipeline[T, F, A, B, O]{type Out = O}
  def apply[…](implicit …): Aux[…] = …
  implicit def PNil: Pipeline.Aux[CNil, CNil, CNil, Unit:+:CNil] = …
  implicit def inductivePipeline[…](implicit …): Aux[…] = …
  implicit class Ops[
    TT <: Coproduct, AT <: Coproduct, BT <: Coproduct, OT <: Coproduct
  ](val tail: Pipeline.Aux[TT, AT, BT, OT]) extends AnyVal{
    def +:[TH, F[_], AH, BH](
      head: Pipeline.Aux[TH, AH, BH, Pipeline.Out[F]]
    ): Pipeline.Aux[TH:+:TT,AH:+:AT, BH:+:BT,F[Unit]:+:OT] =
      inductivePipeline[ TH, F, AH, BH, TT, AT, BT, OT](head, tail)
  }
}

val pipeline1 = Pipeline[One, …]
val pipeline2 = Pipeline[Two, …]
val pipeline3 = Pipeline[Three, …]
val pipeline4 = pipeline1 +: pipeline2 +: pipeline3 +:
Pipeline.PNil
pipeline4(uri)
```

Two Considerations

`implicit def Pnil: …`

This is not a val. The implicit conversion does not work when a val is used. The inferencer has trouble unifying.

`Pipeline[T, Dataset, …]`

Does not work. Spark Dataset requires a Context bound on its types. cats.Functor does not have Context bounds.

```
object Pipeline{
  trait BoundedFunctor[F[_], B[_]]{
    def map[U: B, V: B](fu: F[U])(f: U => V): F[V]
  }
  trait Functor[F[_]] extends BoundedFunctor[F, Id]
  def apply[T: Guard, F[_]: BoundedFunctor, A, B]…
  …
}
implicit val sparkFunctor = new BoundedFunctor[Dataset, Encoder]{
  override def map[U: Encoder, V: Encoder](
    fu: Dataset[U])(f: U => V): Dataset[V] = fu.map(f)
}
```

In Sum

Abstract Types First

Abstract Functions Second

Find library implementation of your discoveries

Lift independent implementation details Outside the Class

Implicits can help model inductive processes

Do not be afraid of rolling your own classes for special cases