Those 10000 classes I never wrote.

The power of recursion schemes applied to data engineering.
GREETINGS

• The name’s Valentin (@ValentinKasas)
• Organizer of the Paris Scala User Group
• Member of the ScalaIO conference crew
• Cook
• Wannabe marathon runner
DISCLAIMER

• Too much to show in a short time
  ⇒ no time for cat pictures or funny gifs

• A lot of code that is
  • potentially scary
  • probably doesn’t compile
THE STORY OF A DATA ENGINEERING PROBLEM
Building a « meta-Datalake »

~100 data sources w/ ~100 tables each

From None to production in 6 months

5 developers
• Data coming in batches and streams
• Privacy by design
• Data sources enrolled on a rolling basis
• Pipeline must be configured at runtime
• Needs a specific schema (← privacy)
• Must work with different input & output formats
  • JSON, CSV, Parquet, Avro, …
SOLUTION
The only way to have a generic Pipeline is to build it around a schema.

That schema must contain enough information to allow the required «privacy-by-design».
Schemas are cool

With a schema we can generically

• validate data
• generate random test data
• translate data between formats
Schemas are recursive

• a schema is composed of smaller schemas
  • that are composed of smaller schemas
    • that are composed of smaller schemas
      • ... etc
  • ... etc
• ... and so is the data they represent
Explicit recursion is bad

• The compiler doesn’t help much
• StackOverflowError is around the corner
• Mixed concerns
• Not reusable code
Recursion Schemes FTW!

- Originate in « Functional programming with Bananas, Lenses, Envelopes and Barbed Wire »
- Decouple the how and the what
- Scala implementation: [github.com/slamdata/matryoshka](https://github.com/slamdata/matryoshka)
**Families of Schemes**

- **folds**: destroy a structure bottom-up
  - ex: cata

- **unfolds**: build-up a structure top-down
  - ex: ana

- **refolds**: combine an unfold w/ a fold
  - ex: hylo
To use any recursion scheme, you need:

• a **Functor** (called « pattern-functor »)
• a **Fix-point type** (most of the time)
• an **Algebra** and/or a **Coalgebra**
sealed trait Tree
final case class Node(left: Tree, right: Tree) extends Tree
final case class Leaf(label: Int) extends Tree

• idea : replace the recursive « slot » by a type parameter
sealed trait TreeF[A]
final case class Node[A](left: A, right: A) extends TreeF[A]
final case class Leaf[A](label: Int) extends TreeF[A]

object TreeF {

  implicit val treeFunctor = new Functor[TreeF] {
      case Node(l, r) => Node(f(l), f(r))
      case Leaf(i)    => Leaf[B](i)
    }
  }
}
Problem: different shapes of trees have different types

val tree1 = Node(Node(Leaf(1), Leaf(2)), Node(Leaf(3), Leaf(4)))
// tree1: Node[Node[Leaf[Nothing]]] = ...

val tree2 = Node(Leaf(0), Leaf(-1))
// tree2: Node[Leaf[Nothing]] = ...

val tree3 = Node(Leaf(42), Node(Leaf(12), Leaf(84)))
// tree3: Node[TreeF[_ <: Leaf[Nothing]]] = ...
• Solution: use a fix-point type to «swallow» recursion

```scala
final case class Fix[F[_]](unFix: F[Fix[F]])
```
**Fix-point types**

- Solution: use a fix-point type to « swallow » recursion

```scala
final case class Fix[F[_]](unFix: F[Fix[F]])

val tree2 = Fix[TreeF](Node(
  Fix[TreeF](Leaf(0)),
  Fix[TreeF](Leaf(-1))))
// tree2: Fix[TreeF] = ...

val tree3 = Fix[TreeF](Node(
  Fix[TreeF](Leaf(42)),
  Fix[TreeF](Node(
    Fix[TreeF](Leaf(12)),
    Fix[TreeF](Leaf(84))))))
// tree3: Fix[TreeF] = ...
```
(Co)Algebras

• Define **what** to do with a **single layer** of your recursive structure

• **Algebras**: collapse, 1 layer at a time
  \[ F[A] \Rightarrow A \]

• **Coalgebras**: build-up, 1 layer at a time
  \[ A \Rightarrow F[A] \]

• **A** is referred to as the (co)algebra’s **carrier**
**HYLOMORPHISMS**

- Pattern-functor
- Algebra
- Coalgebra

```python
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
```
def hylo[F[_], A, B](a: A)  
  (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =  

  alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_], A, B](a: A)
  (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
  alg(coalg(a) map (hylo(_)(alg, coalg))))
def hylo[F[_], A, B](a: A)
  (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B = 
  alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))


**HYLOMORPHISMS**

- □ Pattern-functor
- ▲ → ▲ Algebra
- • → • Coalgebra

```python
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
```
def hylo[F[_], A, B](a: A) (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B = alg(coalg(a) map (hylo(_)(alg, coalg)))
**HYLOMORPHISMS**

def hylo[F[_, A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_], A, B](a: A) 
  (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B = 
  
  alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_], A, B](a: A) (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B = 
alg(coalg(a) map (hylo(_)(alg, coalg)))
**HYLOMORPHISMS**

```
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
```
**Hylomorphisms**

```
def hylo[F[_], A, B](a: A)
  (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
  alg(coalg(a) map (hylo(_)(alg, coalg))))
```
def hylo[F[()], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_:)(alg, coalg)))
def hylo[F[_], A, B](a: A) 
  (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B = 
  alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[()], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
        alg(coalg(a) map (hylo(})(alg, coalg)))
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_], A, B](a: A)
(alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =

alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylomorphisms\[F[\_], A, B\](a: A)\n(alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =

alg(coalg(a) map (hylomorphisms(\_)(alg, coalg)))
def hylo[F[_], A, B](a: A)
  (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
  alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[[_]], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
**Hylomorphisms**

\[
def hylo[F[_], A, B](a: A) 
  (alg: Algebra[F, B], co alg: Coalgebra[F, A]): B = 
  alg(co alg(a) \text{ map } (hylo(_)(alg, co alg)))
\]
def hylo[F[_], A, B](a: A) (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B = alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[()], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo[()])(alg, coalg)))
def hylo[F[_], A, B](a: A)  
(alg: Algebra[F, B], coalg: Coalgebra[F, A]): B = 
  alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_, A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))

Pattern-functor

Algebra

Coalgebra
def hylo[F[_], A, B](a: A) (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B = alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_, A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
**Hylomorphisms**

```
def hylo[F[_, A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[__], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(__)(alg, coalg)))
def hylo[F[_], A, B](a: A) 
(alg: Algebra[F, B], coalg: Coalgebra[F, A]): B = 
alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
**HYLOMORPHISMS**

Pattern-functor

Algebra

Coalgebra

```python
def hylo[F[()], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
```
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
**HYLOMORPHISMS**

- Pattern-functor
- Algebra
- Coalgebra

```python
def hylo[F[_], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(_)(alg, coalg)))
```
def hylo[F[_]], A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
        alg(coalg(a) map (hylo(_)(alg, coalg)))
def hylo[F[\_] _, A, B](a: A)
    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =
    alg(coalg(a) map (hylo(\_)(alg, coalg)))
One schema to rule them all…

Input Schema

Parquet

Avro
ONE SCHEMA TO RULE ONE SCHEMA TO RULE THEM ALL...

Input Schema

SchemaF[A]

Avro

Parquet
ONE SCHEMA TO RULE ONE SCHEMA TO RULE THEM ALL...
ONE SCHEMA TO RULE ONE SCHEMA TO RULE THEM ALL...

Input Schema

SchemaF[A]

Avro

Parquet
sealed trait SchemaF[A]

final case class StructF[A](fields: Map[String, A]) extends SchemaF[A]
final case class ArrayF[A](elementType: A) extends SchemaF[A]
final case class IntF[A]() extends SchemaF[A]
final case class StringF[A]() extends SchemaF[A]

// etc ...

object SchemaF {

  implicit val schemaFunctor = new Functor[SchemaF] {

      case StructF(fields) => StructF(fields.mapValues(f))
      case ArrayF(elem) => ArrayF(f(elem))
      case IntF() => IntF[B]()
      case StringF() => StringF[B]()
    }
  }
}
STEP 2: ALGEBRAS

val schemaToParquet: Algebra[SchemaF, DataType] = {
  case StructF(fields) =>
    StructType(fields.map((StructField.apply _).tupled))
  case ArrayF(elem) => ArrayType(elem)
  case IntF() => IntegerType
  case StringF() => StringType
}

fixSchema.cata(schemaToParquet)
val avroToSchema: Coalgebra[SchemaF, Schema] = { avro =>
    avro.getType match {
      case RECORD =>
        StructF(avro.getFields.map(f => f.name -> f.schema))
      case ARRAY => ArrayF(avro.getElementType())
      case INT => IntF()
      case STRING => StringF()
    }
  }

avroSchema.ana[Fix](avroToSchema)

avroSchema.hylo(schemaToParquet, avroToSchema)
Validating Data

- we used github.com/jto/validation
- defines the Rule[I, O] type
- we need a single ADT to represent data
- otherwise we couldn’t write algebras

```scala
sealed trait DataF[A]
final case class GStruct[A](fields: Map[String, A]) extends DataF[A]
final case class GArray[A](elements: List[A]) extends DataF[A]
final case class GInt[A](value: Int) extends DataF[A]
final case class GString[A](value: String) extends DataF[A]
```
PRODUCING VALIDATORS

val validator: Algebra[SchemaF, Rule[JsValue, Fix[DataF]]] = {
  case StructF(fields) =>
    fields.toList.traverse{ case (name, validation) =>
      name -> (JsPath \ name).read(validation)
    }.map(fs => Fix(GStruct(fs.toMap)))

  case ArrayF(element) =>
    JsPath.pickList(element).map(es => Fix(GArray(es)))

  case IntF() =>
    JsPath.read[Int].map(v => Fix(GInt(v)))

  case StringF() =>
    JsPath.read[String].map(v => Fix(GString(v)))
}
LESS EASY STUFF
• **Impossible** to write directly an Algebra[$\text{SchemaF, Schema}$]

• In an avro schema, each type must have an **unique** name

• **Solution:**
  
  • use the path of each node to name types
  
  • store previously built schemas in a registry
**Solution 1: Keep Track of the Path**

```scala
definal case class EnvT[E, F[_], A](run: (E, F[A]))

type WithPath[A] = EnvT[String, SchemaF, A]

val schemaWithPath: Coalgebra[WithPath, (String, Fix[SchemaF])] = {
  case (path, Fix(StructF(fields))) =>
    EnvT((
      path,
      StructF(fields.map{case (k, v) => (s"$path.$k", Fix(v))})
    ))
  case (path, Fix(ArrayF(e))) =>
    EnvT((
      path,
      ArrayF((path, Fix(e)))
    ))
  // ...
}
```
def withPathToAvro(namespace: String):
  Algebra[EnvT[String, SchemaF, ?], Schema] = {
    case EnvT((path, StructF(fields))) =>
      SchemaBuilder
        .namespace(namespace)
        .record(path)
        ....
  }

('', fixSchema).hylo(withPathToAvro, schemaWithPath)
SOLUTION 2: USE A REGISTRY

- Schemes come in different flavours: «classic», monadic, generalized

- Here we need the monadic flavour:


  \begin{align*}
  \text{type } & \text{FingerPrinted } = (\text{Long, Schema}) \\
  \text{type } & \text{Registry}[A] = \text{State}[\text{Map}[\text{Long, Schema}], A]
  \end{align*}
Solution 2: Using a Registry

```scala
val reuseTypes: AlgebraM[Registry, SchemaF, FingerPrinted] = {
  avro =>
  case StructF(fields) =>
    val fp = fingerPrint(fields)
    State.get.flatMap{
      knownTypes =>
        if(knownTypes.contains(fp))
          State.state(fp -> knownTypes(fp))
        else {
          val schema = SchemaBuilder.
            newRecord(fp)
          ...
          State.put( knownType + (fp -> schema))
            .map(State.state(fp -> schema))
        }
    }
}

fixSchema.cataM(reuseTypes)
```
• Once we get a `Fix[DataF]`, we need to serialize it
  • to Parquet, using spark’s `Row`
  • to Avro using `GenericRecord`
We can nest Row, so we can use an algebra with Row as the carrier

But, we don’t want our « simple » columns to be wrapped

If we have a 2 columns table we want to output Row(col1, col2), not Row(Row(col1), Row(col2))
• We need to know when we’ve reached the « bottom » of our tree

• \textit{para} is a scheme that gives us the previous « tree » it has consumed

• It uses a $\text{GAlgebra}[(T, ?), F, A]$
val toRow: GAlgebra[(Fix[DataF], ?), DataF, Row] = {
  case GStruct(fields) =>
    val values = fields.map{ case (_, (fix, value)) =>
      if (isSimpleType(fix)) value.get(0) // unwrap the row
      else
        value
    }
    Row(values: _*)

  case GInt(value) =>
    Row(value)

  // etc...
}

fixData.para[Row](toRow)
TO BE CONTINUED ...
WELL, ACTUALLY ...
INSPIRED FROM REAL FACTS

... but inspired only
IT MIGHT SEEM HARD AT FIRST ...
It might seem hard at first...
... BUT IT GETS BETTER
... But it gets better