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

Problem Statement

- Building a « meta-Datalake »
- ~100 data sources w/ ~100 tables each
- From `None` to production in 6 months
- 5 developers

MORE FUN

- Data coming in batches and streams
- Privacy by design
- Data sources enrolled on a rolling basis

REQUIREMENTS

- Pipeline must be configured at runtime
- Needs a specific schema (← privacy)
- Must work with different input & output formats
  - JSON, CSV, Parquet, Avro, …

SOLUTION

One Schema to Rule Them All

- 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
- … 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

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`

RECURSION SCHEME RECIPE

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

Pattern Functors

```scala
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

Pattern Functors

```scala
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] {
    def map[A, B](fa: TreeF[A])(f: A => B): TreeF[B] = fa match {
      case Node(l, r) => Node(f(l), f(r))
      case Leaf(i)    => Leaf[B](i)
    }
  }
}
```

Pattern Functors

- Problem : different shapes of trees have different types

```scala
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]]] = ...
```

Fix-point types

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

```
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]])
```

```scala
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

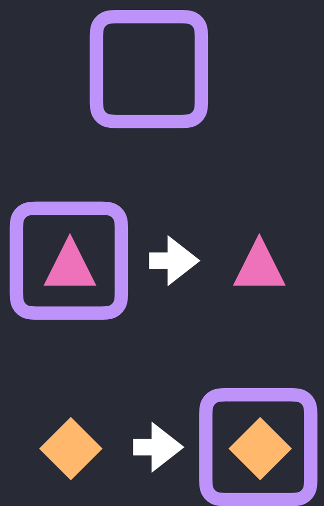
    `F[A] => A`

  - Coalgebras: build-up, 1 layer at a time

    `A => F[A]`

- `A` is referred to as the (co)algebra's carrier

Hylomorphisms



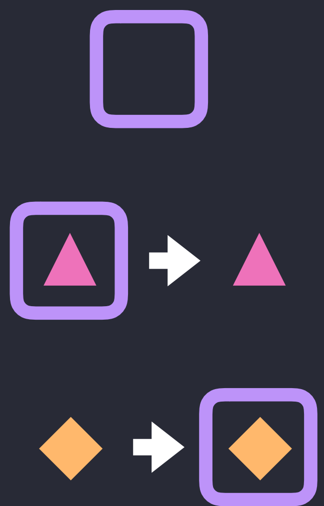
Pattern-functor

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)))
```

HYLOMORPHISMS



Pattern-functor

Algebra

Coalgebra

```scala
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

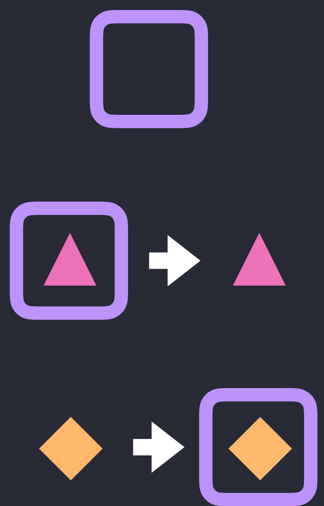
Algebra

Coalgebra

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



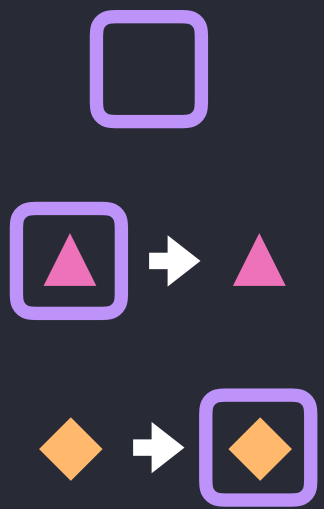
Pattern-functor

Algebra

Coalgebra

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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

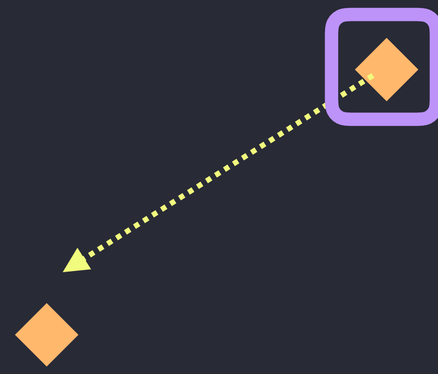
Hylomorphisms



Pattern-functor

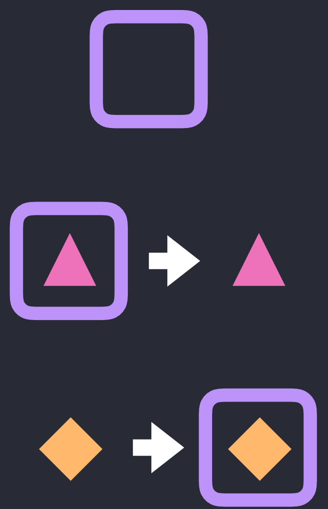
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

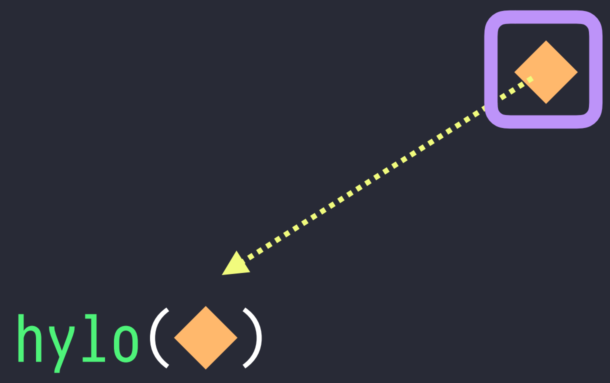
HYLOMORPHISMS



Pattern-functor

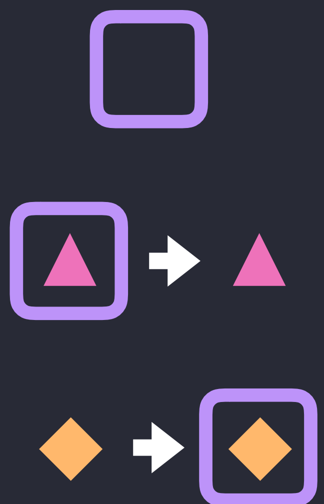
Algebra

Coalgebra




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

  alg(coalg(a) map (hylo(_)(alg, coalg)))
```

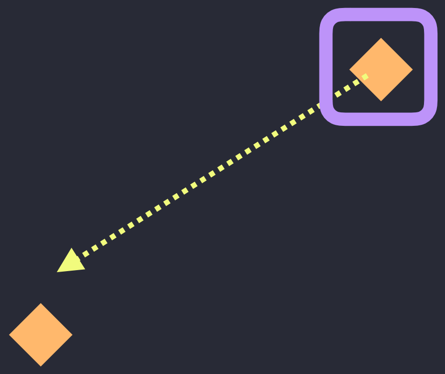
HYLOMORPHISMS



Pattern-functor

Algebra

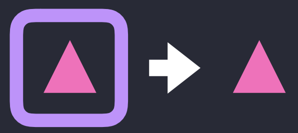
Coalgebra

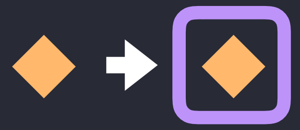


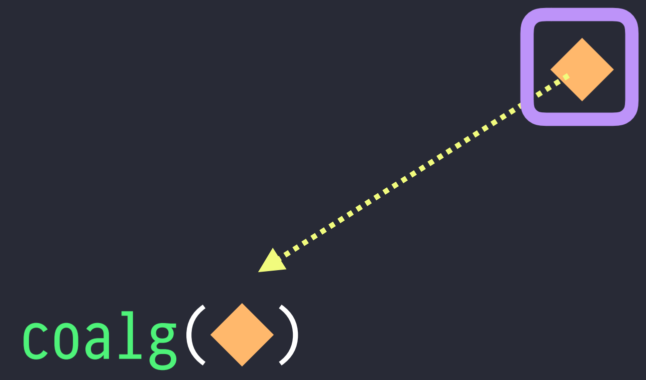
```
def hylo[F[_], A, B](a: A)
                   (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

HYLOMORPHISMS

 Pattern-functor

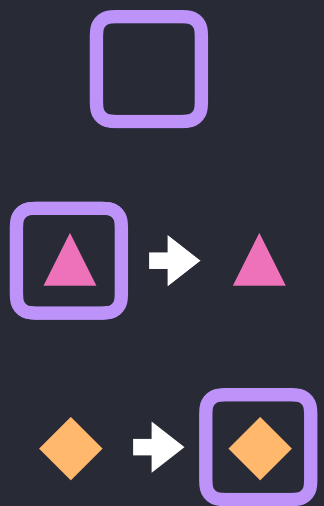
 Algebra

 Coalgebra




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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

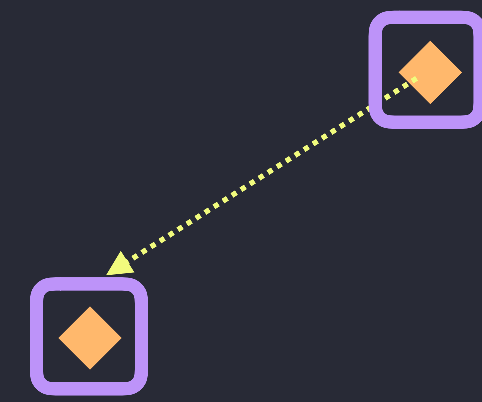
Hylomorphisms



Pattern-functor

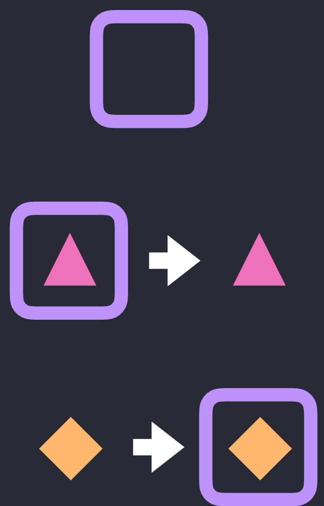
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

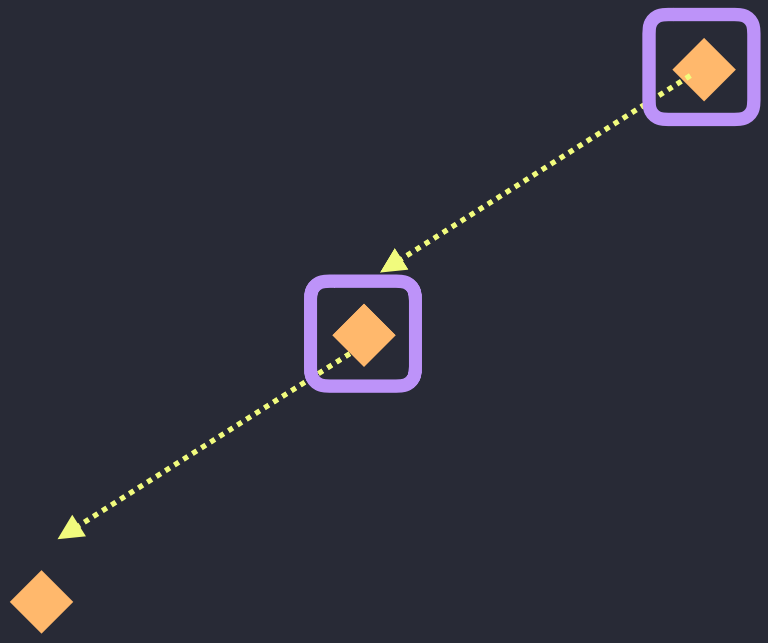
HYLOMORPHISMS



Pattern-functor

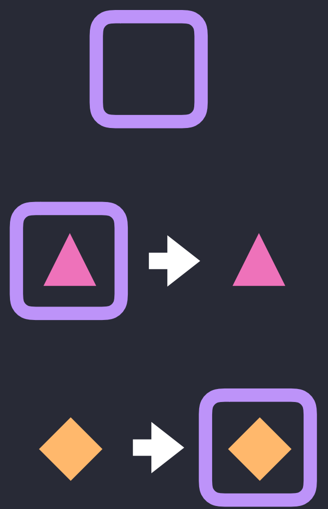
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

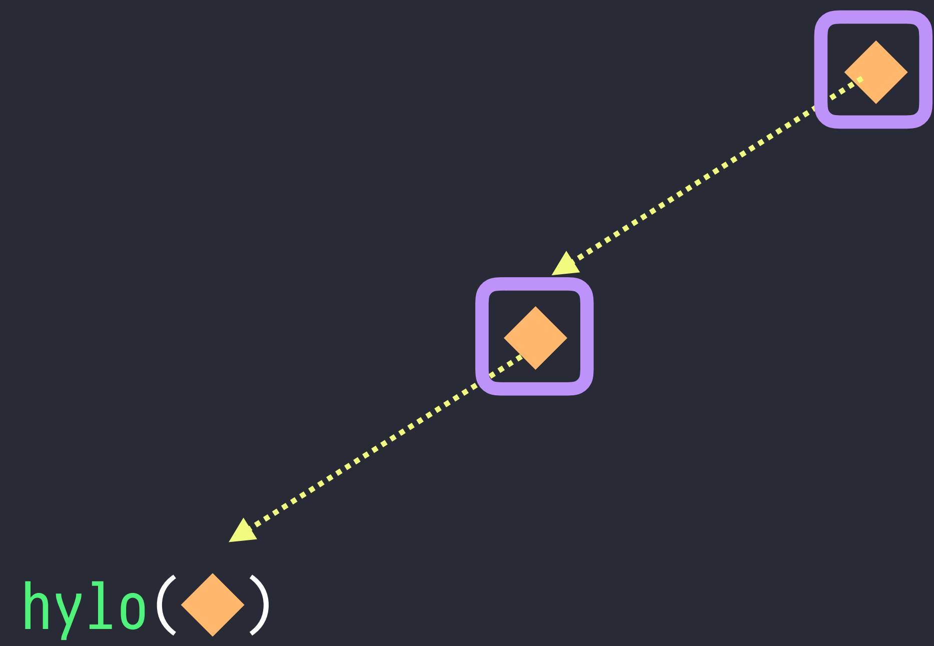
HYLOMORPHISMS



Pattern-functor

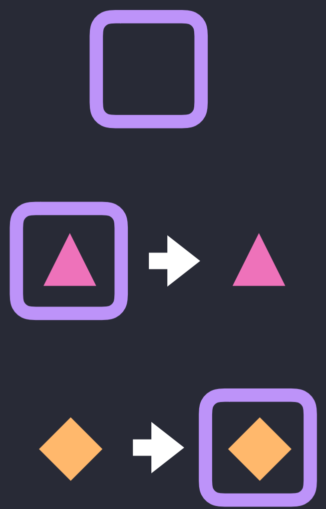
Algebra

Coalgebra




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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

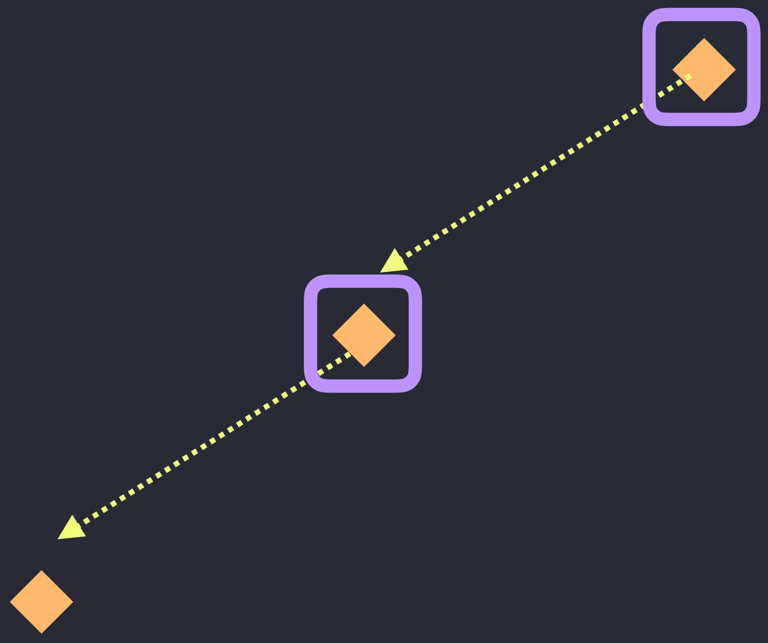
HYLOMORPHISMS



Pattern-functor

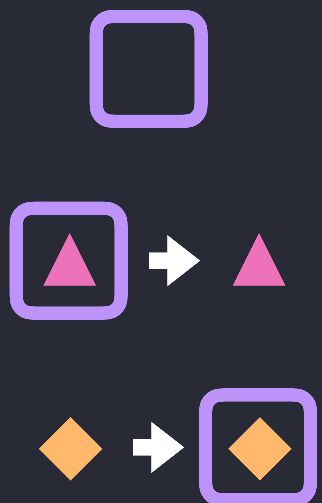
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

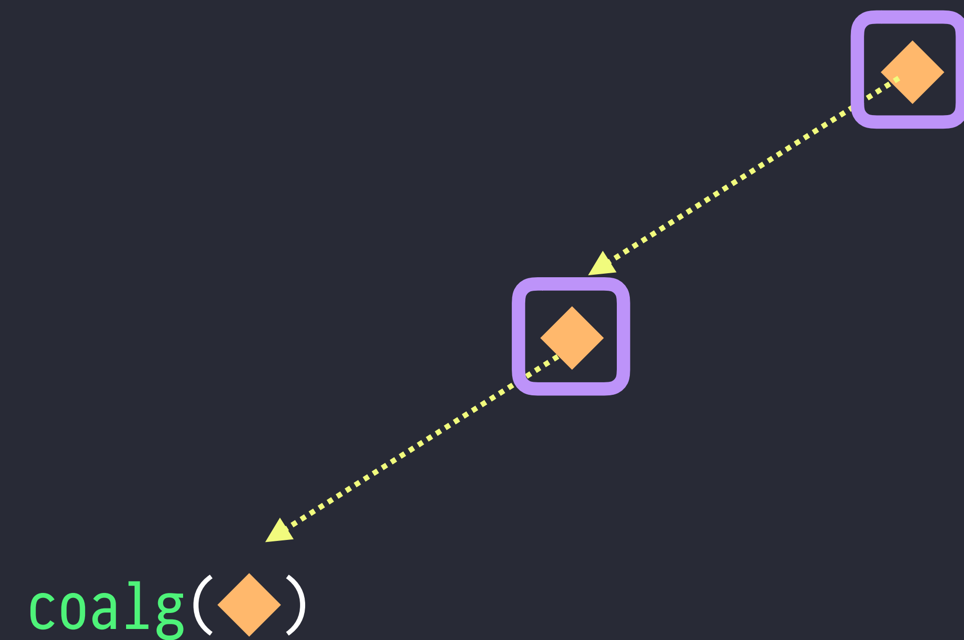
HYLOMORPHISMS



Pattern-functor

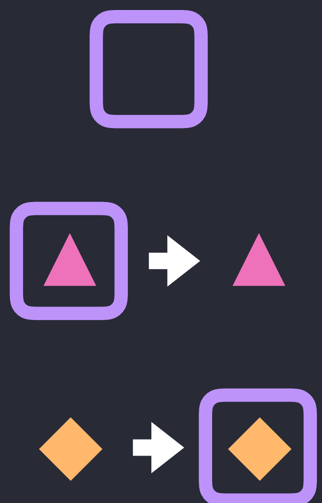
Algebra

Coalgebra




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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

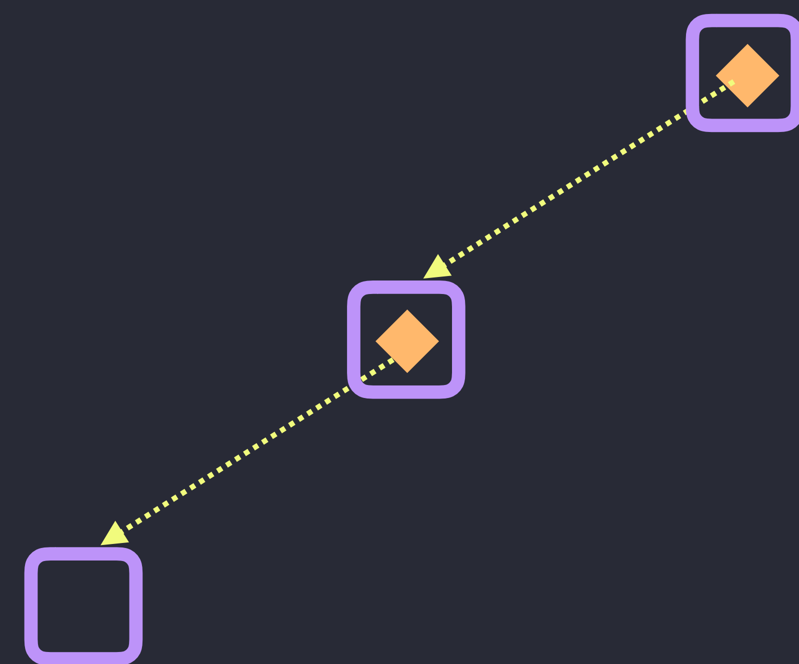
HYLOMORPHISMS



Pattern-functor

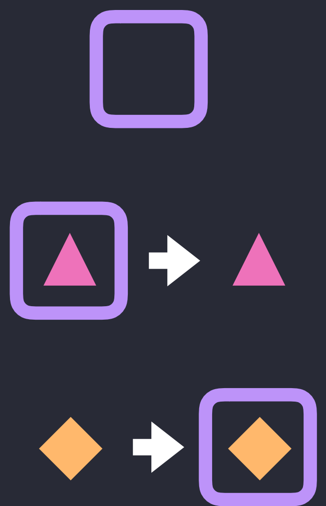
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

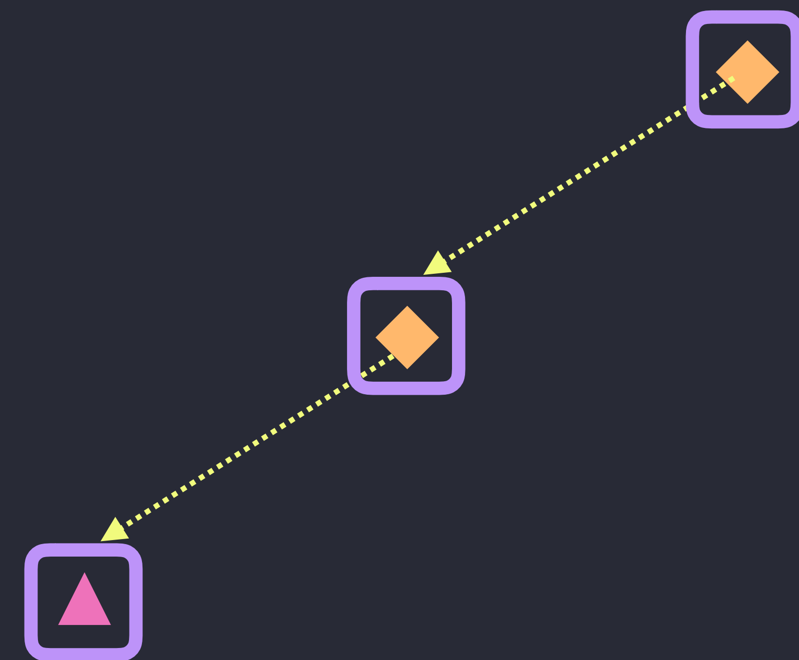
HYLOMORPHISMS



Pattern-functor

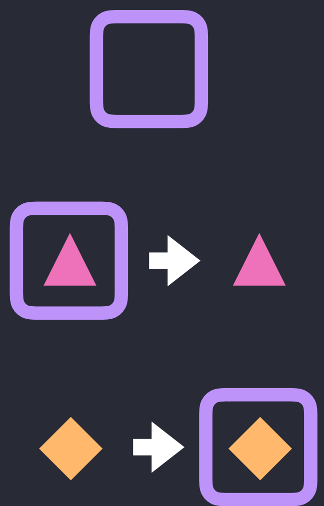
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

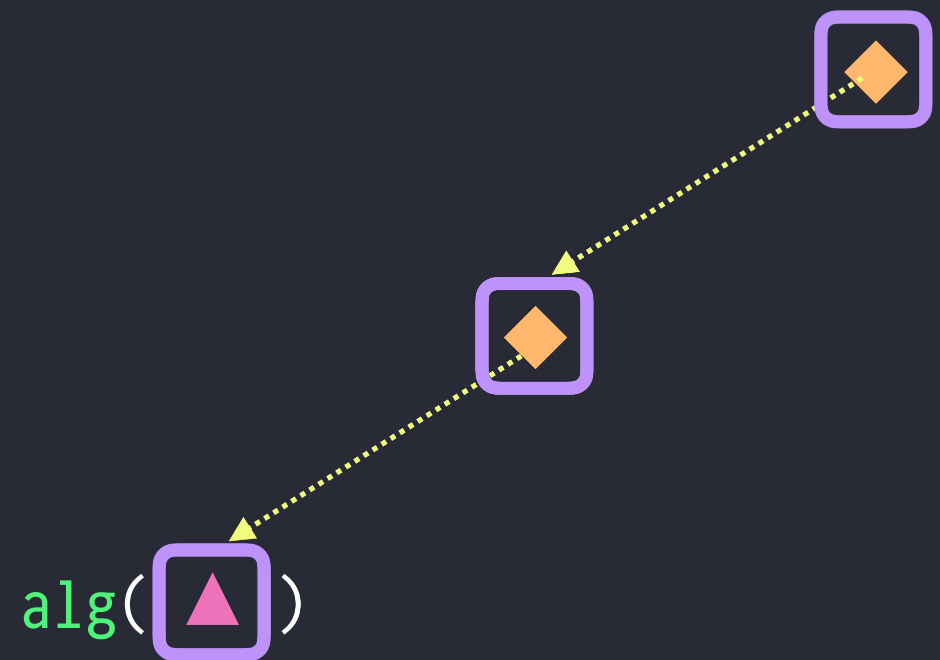
HYLOMORPHISMS



Pattern-functor

Algebra

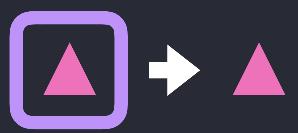
Coalgebra

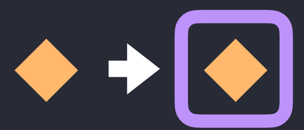


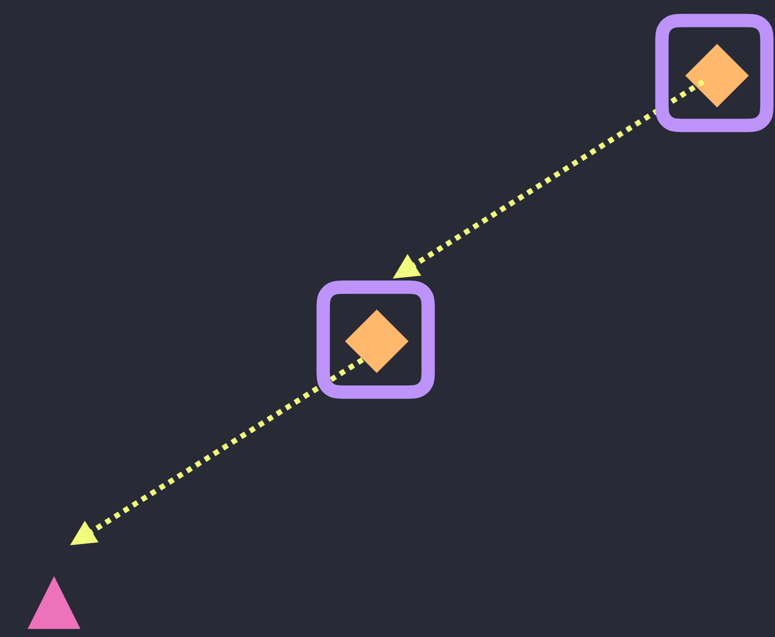

```
def hylo[F[_], A, B](a: A)
                   (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

HYLOMORPHISMS

 Pattern-functor

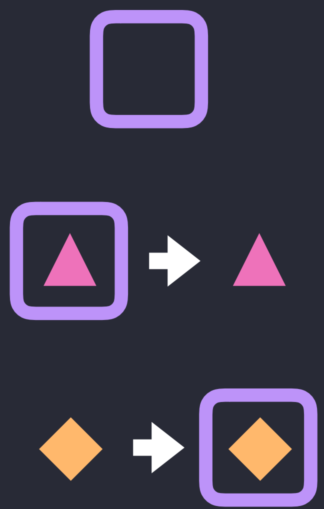
 Algebra

 Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

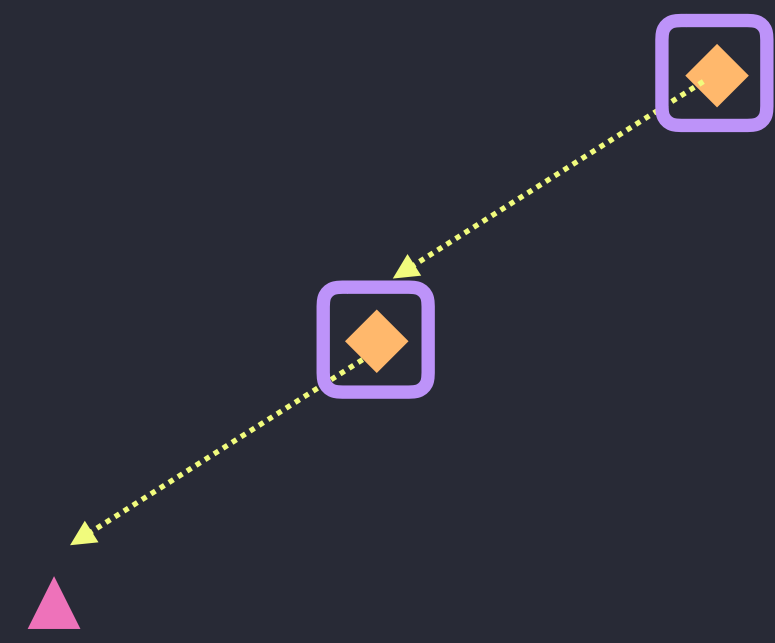
HYLOMORPHISMS



Pattern-functor

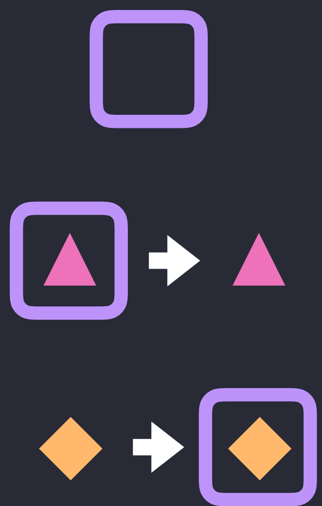
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

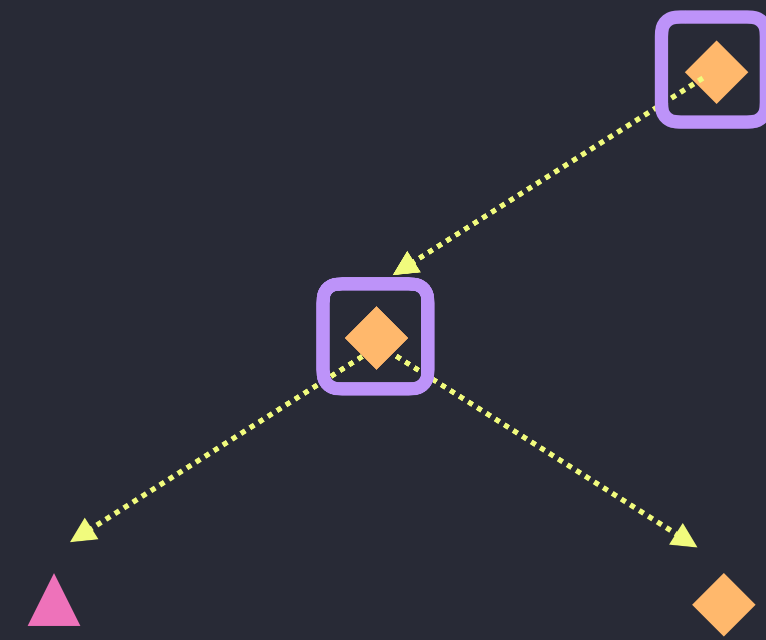
HYLOMORPHISMS



Pattern-functor

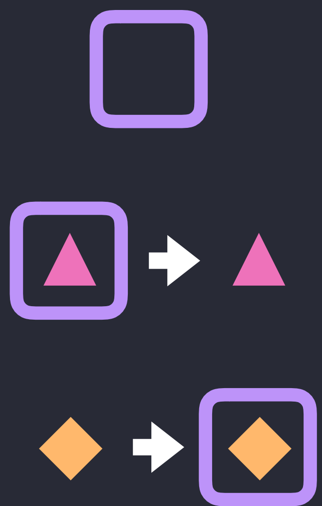
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

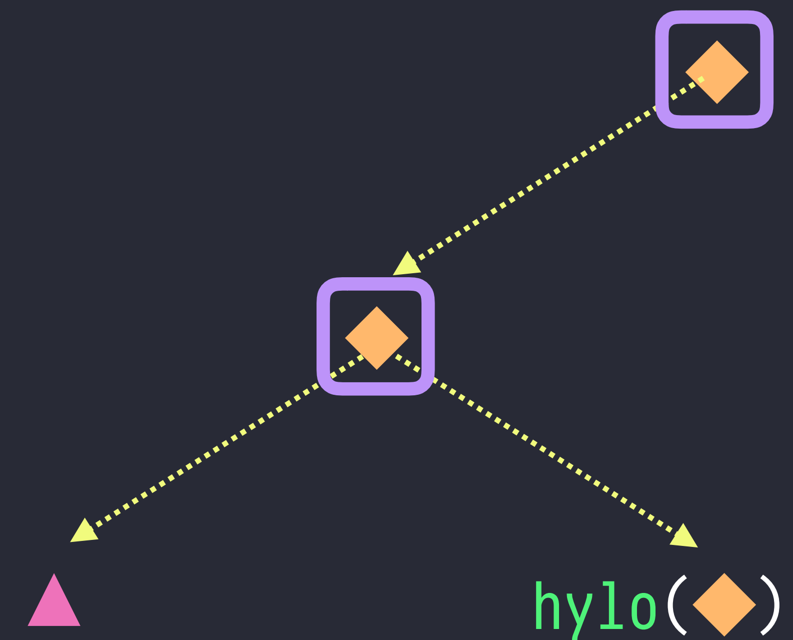
Hylomorphisms



Pattern-functor

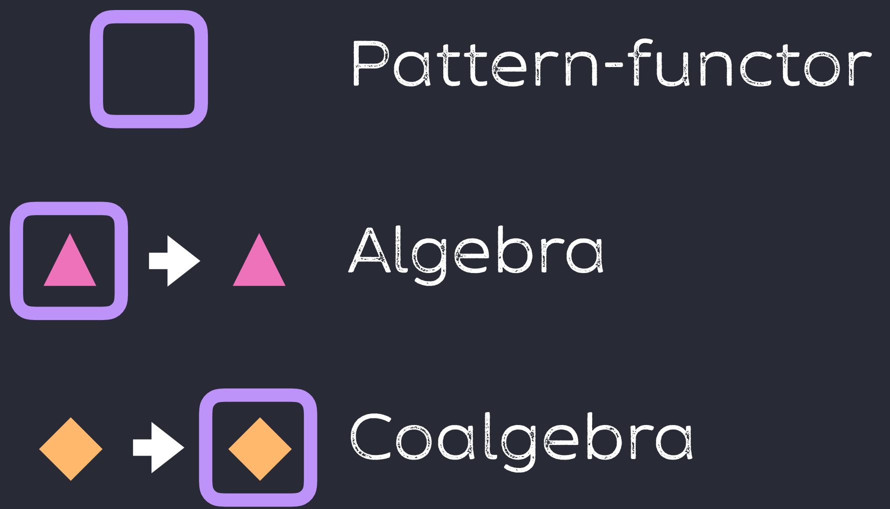
Algebra

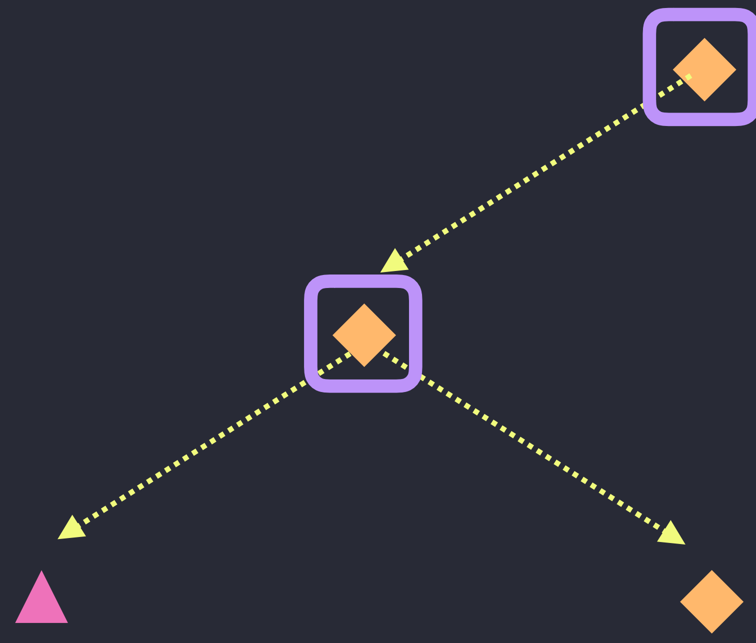
Coalgebra




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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

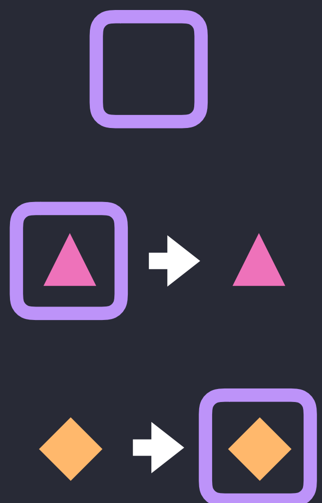
Hylomorphisms





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

  alg(coalg(a) map (hylo(_)(alg, coalg)))
```

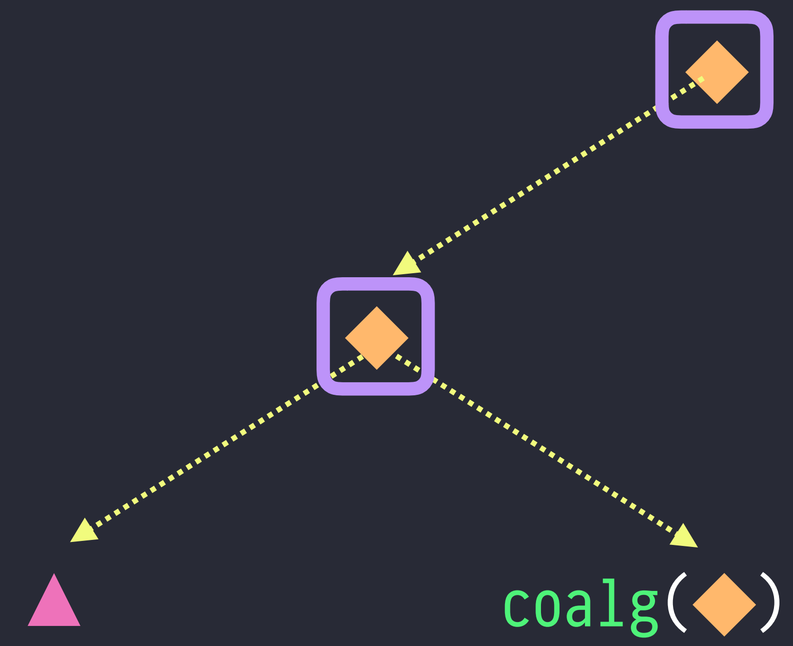
Hylomorphisms



Pattern-functor

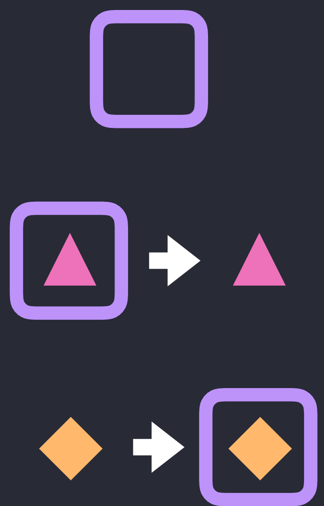
Algebra

Coalgebra




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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

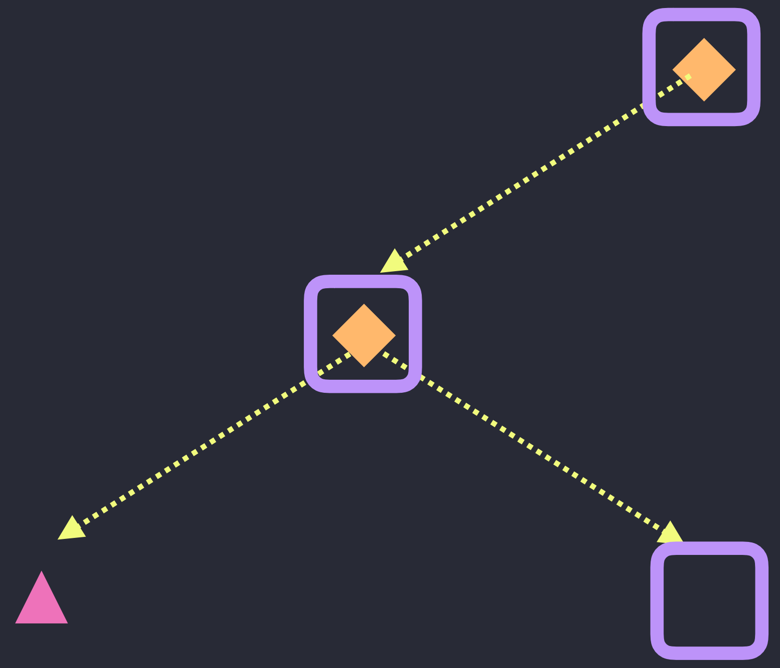
Hylomorphisms



Pattern-functor

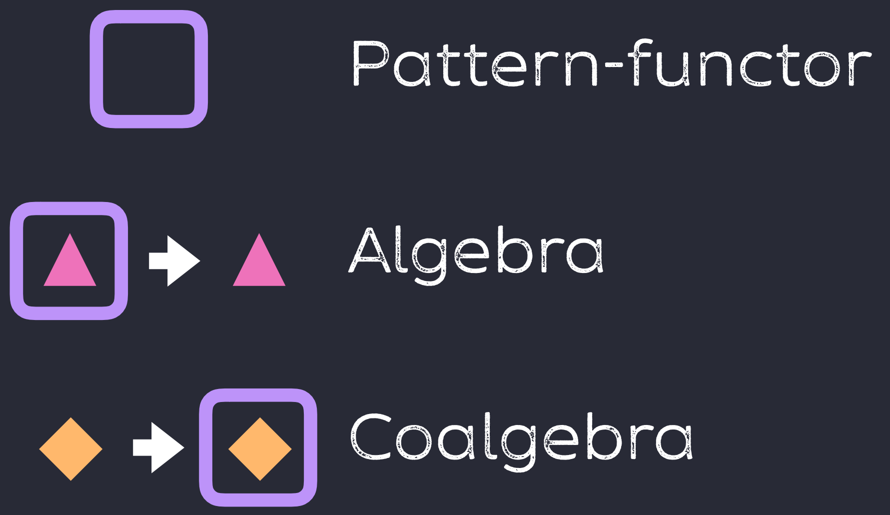
Algebra

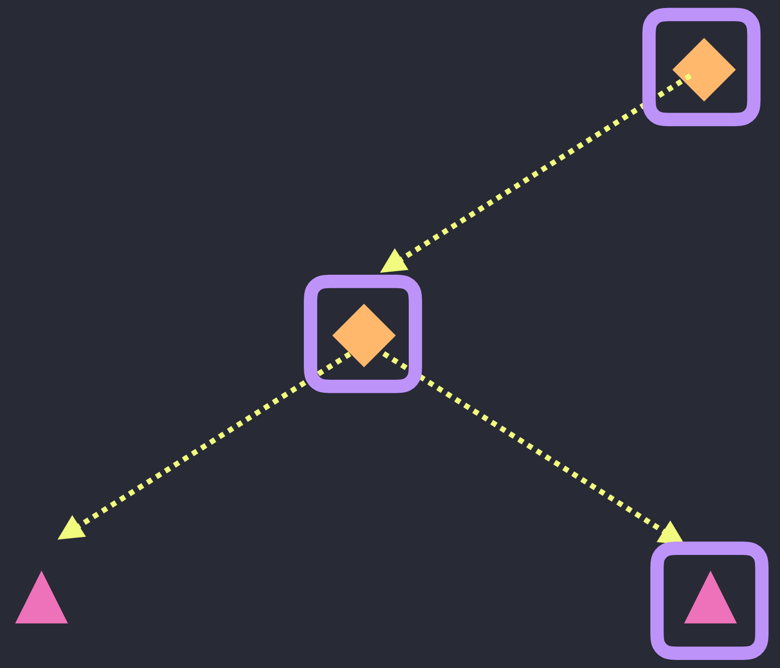
Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

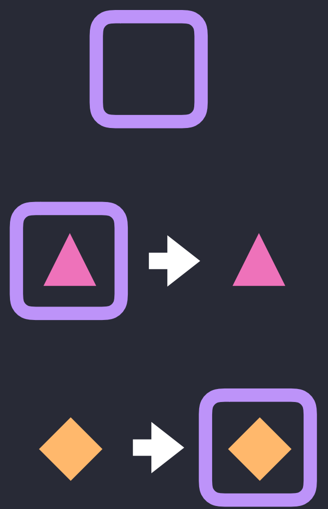
Hylomorphisms





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

  alg(coalg(a) map (hylo(_)(alg, coalg)))
```

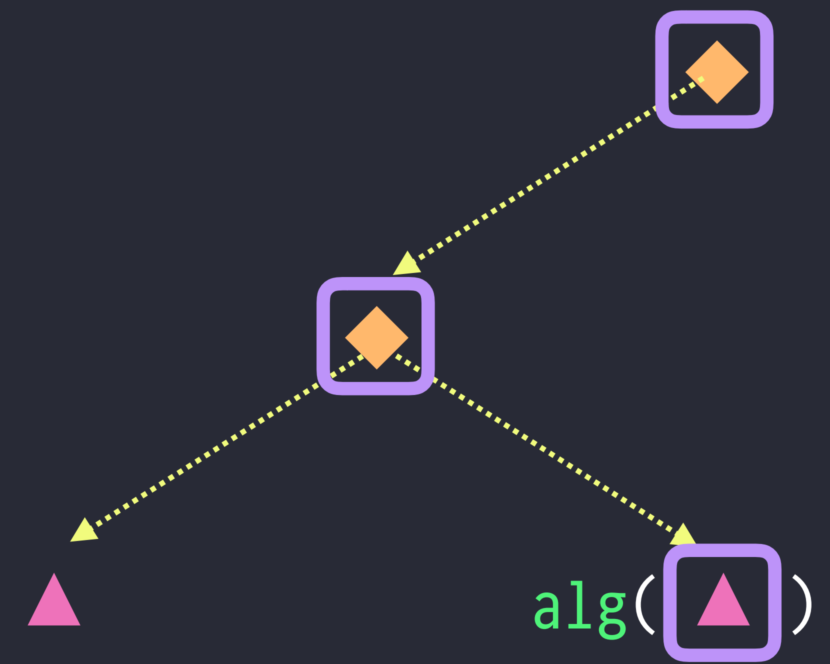
Hylomorphisms



Pattern-functor

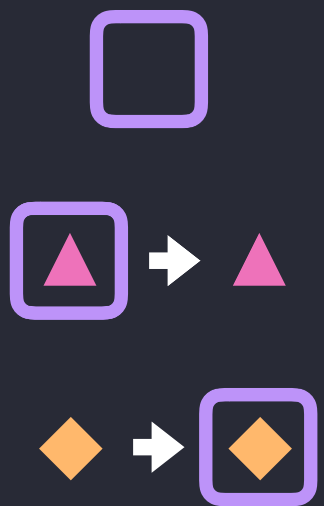
Algebra

Coalgebra




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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

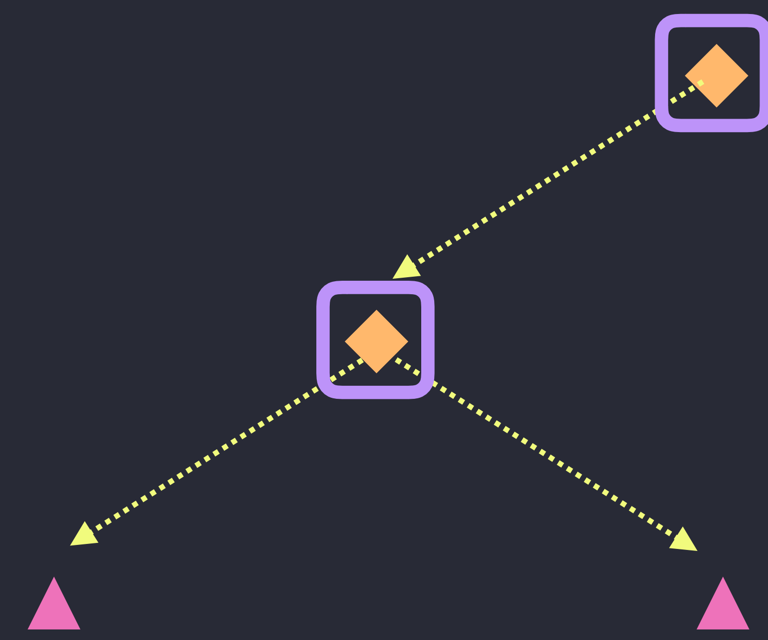
HYLOMORPHISMS



Pattern-functor

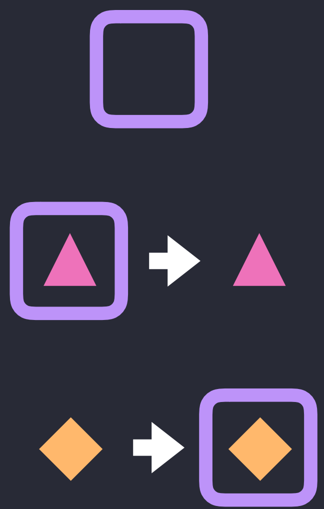
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

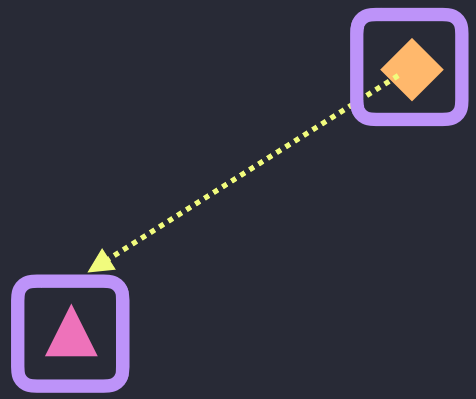
HYLOMORPHISMS



Pattern-functor

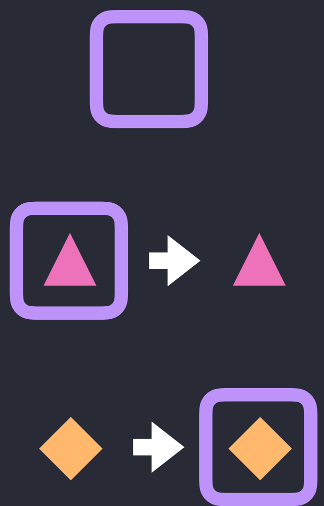
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

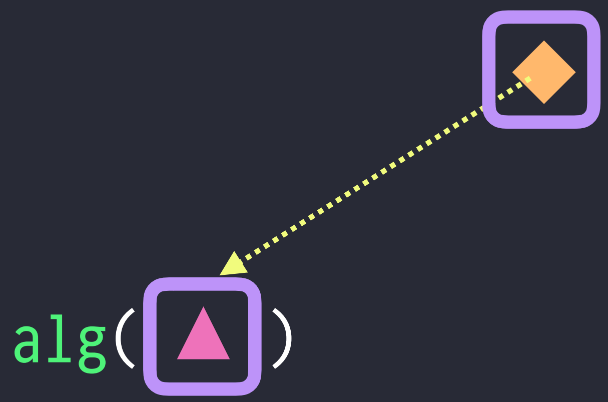
Hylomorphisms



Pattern-functor

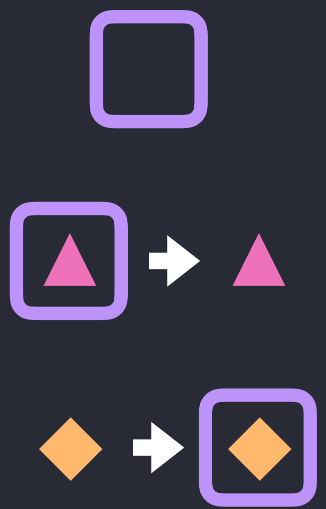
Algebra

Coalgebra




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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

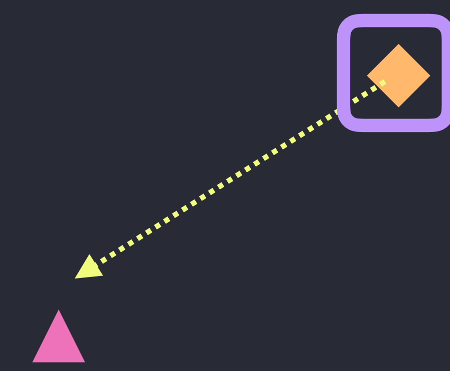
HYLOMORPHISMS



Pattern-functor

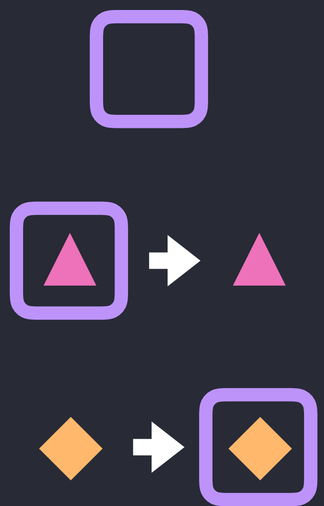
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

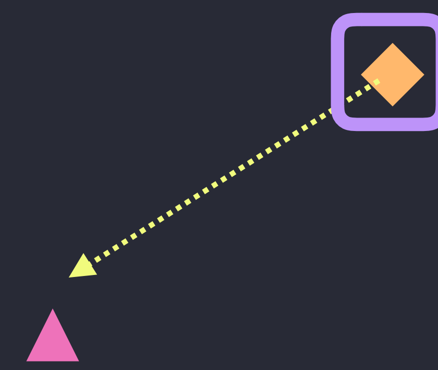
Hylomorphisms



Pattern-functor

Algebra

Coalgebra



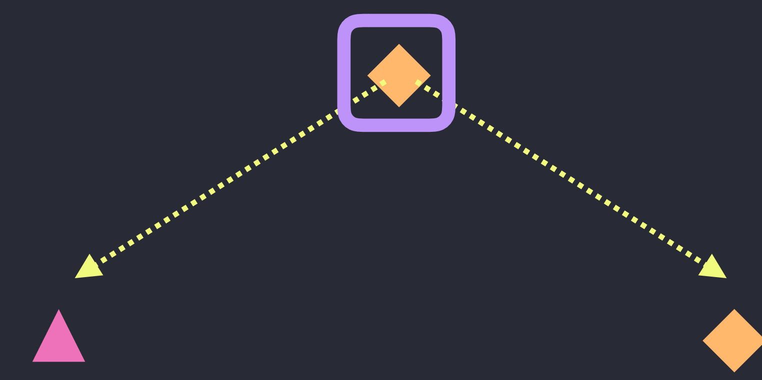
```
def hylo[F[_], A, B](a: A)
                    (alg: Algebra[F, B], coalg: Coalgebra[F, A]): B =

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

HYLOMORPHISMS

Pattern-functor

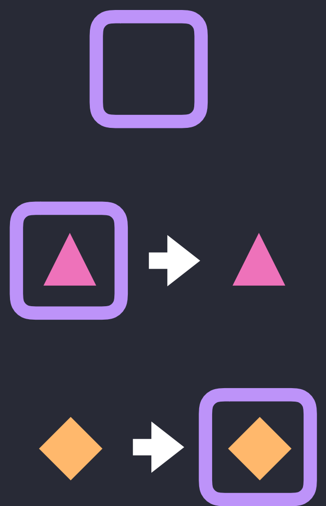
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

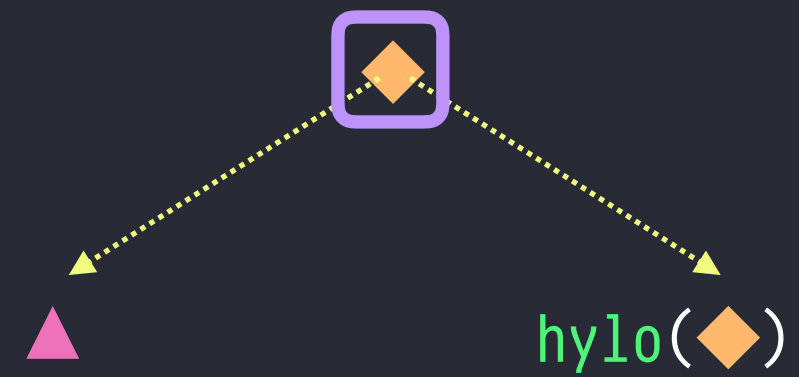
HYLOMORPHISMS



Pattern-functor

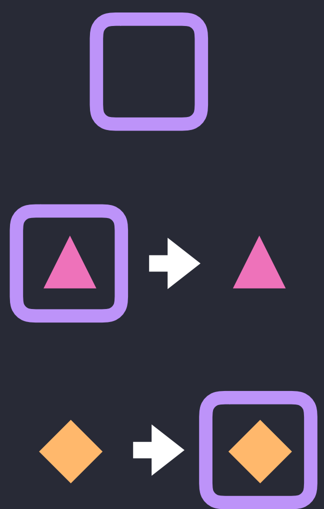
Algebra

Coalgebra




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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

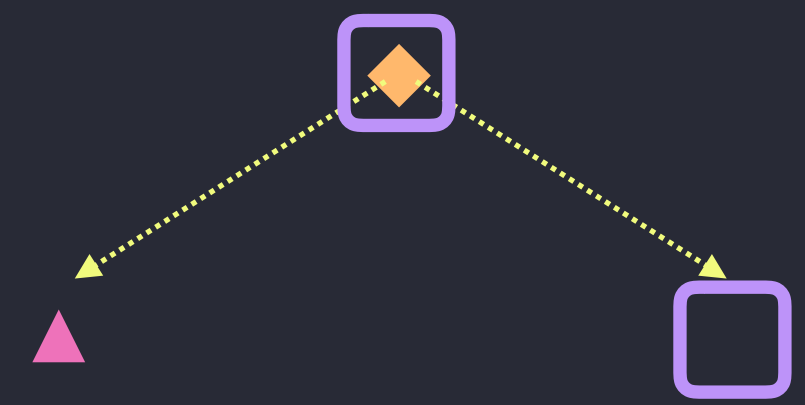
Hylomorphisms



Pattern-functor

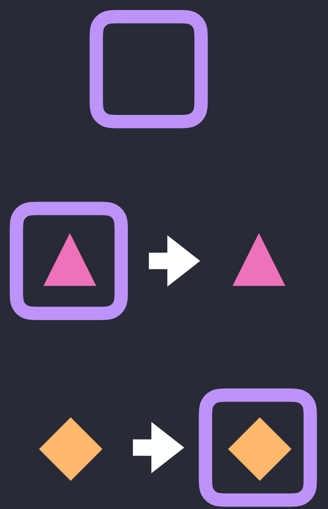
Algebra

Coalgebra



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

  alg(coalg(a) map (hylo(_)(alg, coalg)))
```

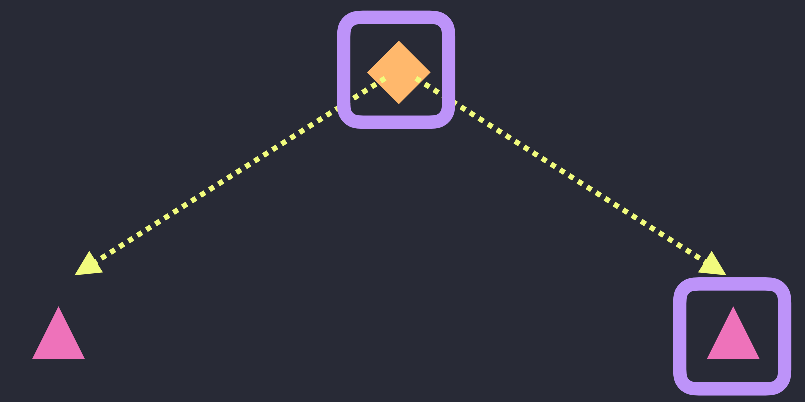
Hylomorphisms



Pattern-functor

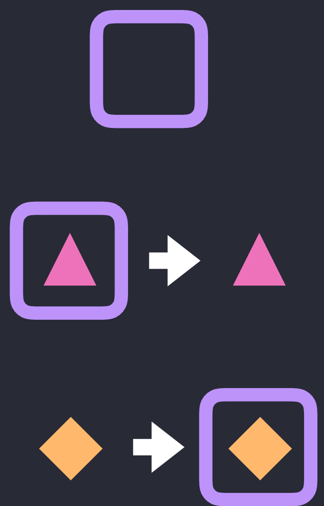
Algebra

Coalgebra



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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

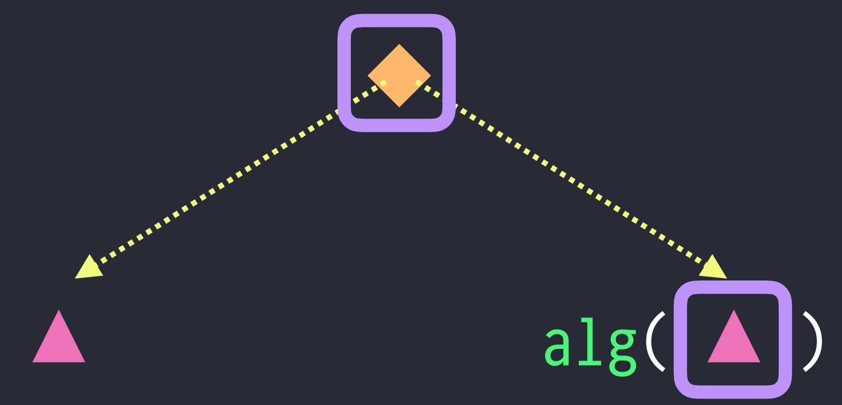
Hylomorphisms



Pattern-functor

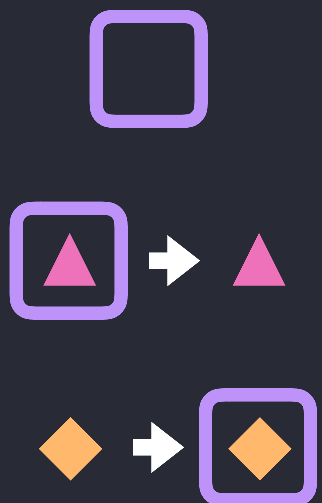
Algebra

Coalgebra




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

  alg(coalg(a) map (hylo(_)(alg, coalg)))
```

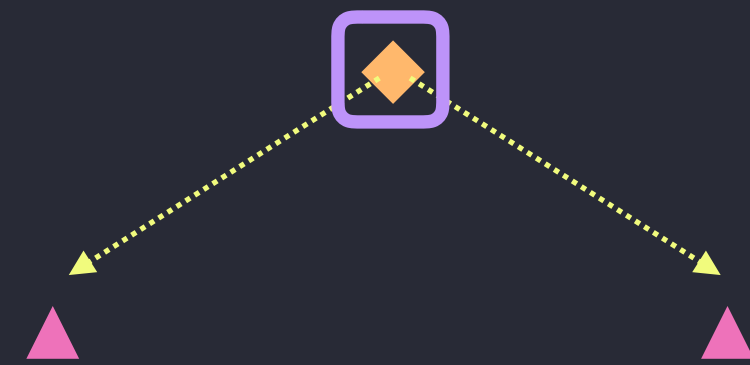
Hylomorphisms



Pattern-functor

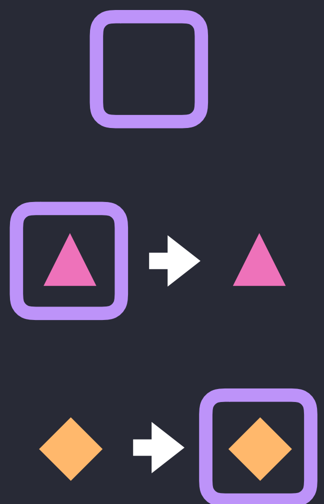
Algebra

Coalgebra



```
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

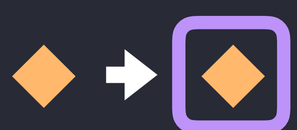
Coalgebra

```
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

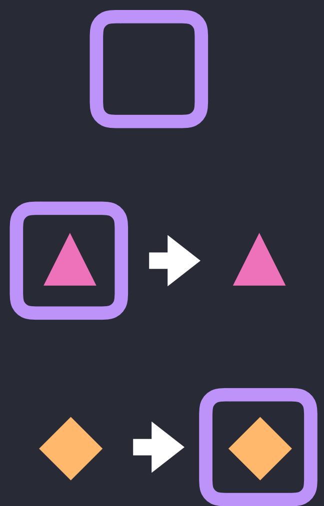
 Algebra

 Coalgebra

alg()

```
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

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

    alg(coalg(a) map (hylo(_)(alg, coalg)))
```

GETTING REAL

One schema to rule one schema to rule them all...

Input Schema

Avro

Parquet

One Schema to rule one schema to rule them all...

Input Schema

Avro

`SchemaF[A]`

Parquet

One Schema to Rule One Schema to Rule Them All...

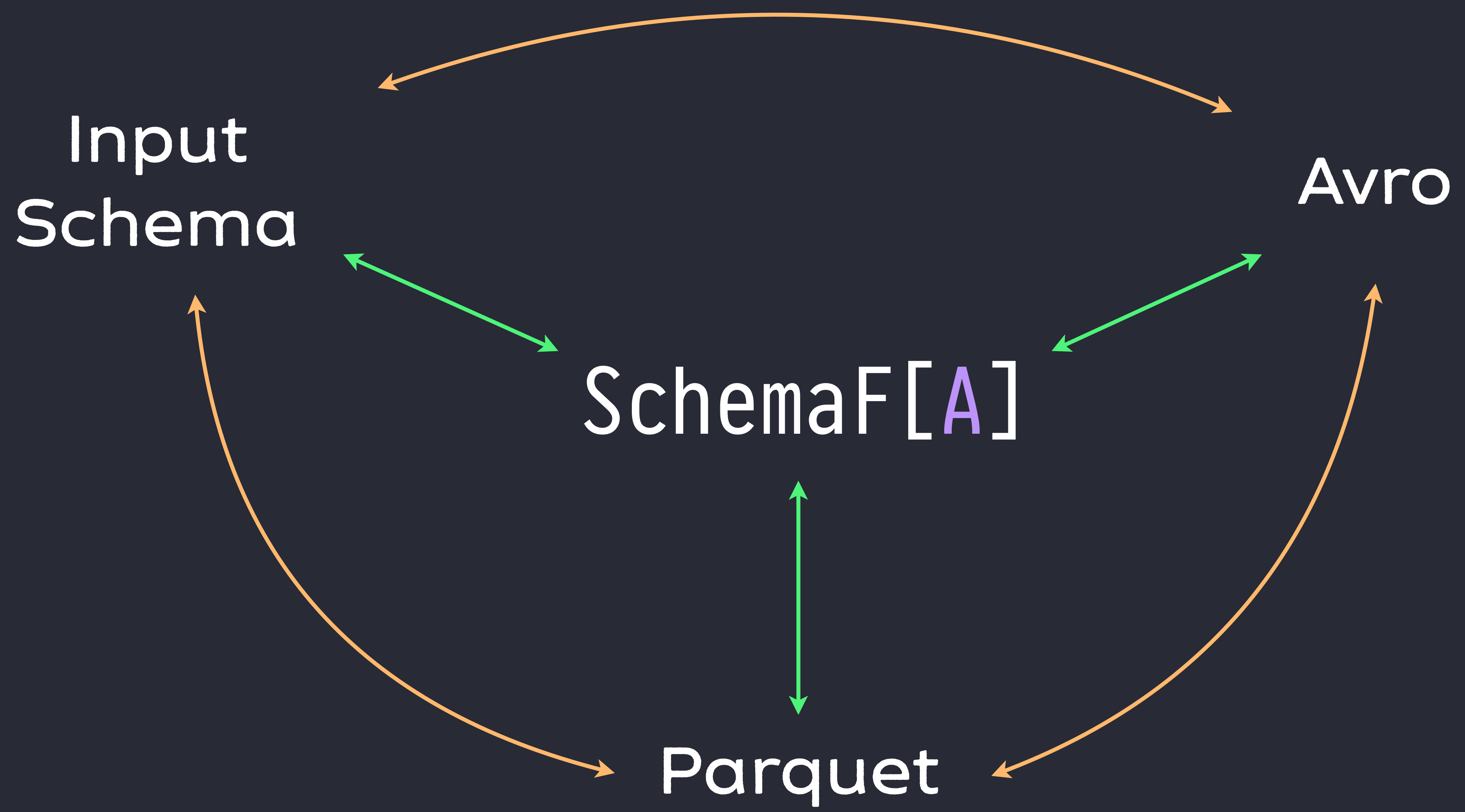
Input Schema

Avro

SchemaF[A]

Parquet

One schema to rule one schema to rule them all...



STEP 1 : PATTERN FUNCTOR

```scala
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] {

    def map[A, B](fa: SchemaF[A])(f: A => B): SchemaF[B] = fa match {
      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)
```

STEP 2.1: COALGEBRAS

```
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

```
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

```scala
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

Schema to Avro

- Impossible to write directly an `Algebra[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
final 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)))
    ))
  // ...
}
```

KEEPING TRACK OF THE PATH (2)

```scala
def withPathToAvro(namespace: String)
: Algebra[EnvT[String, SchemaF, ?], Schema] = {
  case EnvT((path, StructF(fields))) =>
    SchemaBuilder
      .namespace(namespace)
      .record(path)
      ....
}
```

```scala
("", fixSchema).hylo(withPathToAvro, schemaWithPath)
```

SOLUTION 2 : USE A REGISTRY

- Schemes come in different flavours : « classic », monadic, generalized
- Here we need the monadic flavour :

```
AlgebraM[M[_], F[_], A] = F[A] => M[A]
```

```
type FingerPrinted = (Long, Schema)

type Registry[A] = State[Map[Long, Schema], A]
```

Solution 2 : Using a registry

```
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)
```

Writing Transformed Data

- Once we get a `Fix[DataF]`, we need to serialize it
  - to Parquet, using spark's `Row`
  - to Avro using `GenericRecord`

Serializing to Parquet

- We can nest `Row`s, 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))`

Paramorphism

- We need to know when we've reached the « bottom » of our tree
- `para` is a scheme that gives us the previous « tree » it has consumed
- It uses a `GAlgebra[(T, ?), F, A]`

```
GAlgebra[W[_], F[_], A] = F[W[A]] => A
```

BUILDING ROWS

```scala
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