Recursion schemes

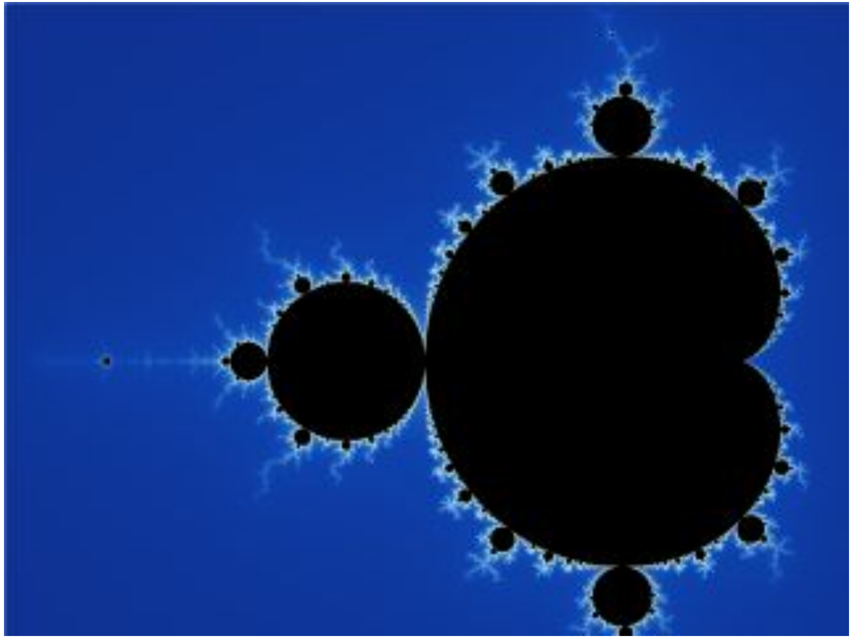
avec Scala

Recursion

Élément qui se contient lui même

Scheme

Un pattern, une structure



Recursion schemes

Un pattern qui se contient lui même.

Scala: FP + OOP

- Fonction
- Objet

Factorial

$n! = n * (n-1) * (n-2) * \ldots * 2 * 1$

$0! = 1$

$4! = 4 * 3 * 2 * 1 = 24$

```
def factorial(n: Int): Int = {
 if(n==0) 1
 else {
   val r = factorial(n-1)
   n * r
 }
}
```

Sum

$\sum n = n + (n-1) + (n-2) + \ldots + 1 + 0$

$\sum 0 = 0$

$\sum 4 = 4 + 3 + 2 + 1 + 0 = 10$

```scala
def sum(n: Int): Int = {
 if(n==0) 0
 else {
   val r = sum(n-1)
   n + r
 }
}
```

isPair

%n = !(%(n-1))

%0 = true

%3 = !(!(!(true)))

```scala
def isPair(n: Int): Boolean = {
 if(n==0) true
 else {
   val r = isPair(n-1)
   !r
 }
}
```

Un schéma de récursion apparaît

```scala
def isPair(n: Int): Boolean = {
 if(n==0) true
 else {
   val r = isPair(n-1)
   !r
 }
}
```

```scala
def sum(n: Int): Int = {
 if(n==0) 0
 else {
   val r = sum(n-1)
   n + r
 }
}
```

```scala
def factorial(n: Int): Int = {
 if(n==0) 1
 else {
   val r = factorial(n-1)
   n * r
 }
}
```

```scala
def scheme[A](baseCase: A, rec: (Int, A) ⇒ A)(n: Int): A = {
 if (n == 0) baseCase
 else {
   val r = scheme(baseCase, rec)(n - 1)
   rec(n, r)
 }
}
```

```scala
def scheme[A](baseCase: A, rec: (Int, A) => A)(n: Int): A

def factorial(n: Int): Int = scheme[Int](1, _*_)(n)

def sum(n: Int): Int       = scheme[Int](0, _+_)(n)

def isPair(n: Int)         = scheme[Boolean](true, (_, b: Boolean) => !b)(n)
```

Optimisons la fonction

```
def scheme[A](baseCase: A, rec: (Int, A) ⇒ A)(n: Int): A = {
 var res = baseCase
 var i = 1
 while (i <= n) {
   res = rec(i, res)
   i += 1
 }
 res
}
```

```scala
def scheme[A](baseCase: A, rec: (Int, A) ⇒ A)(n: Int): A = {
 var res = baseCase
 var i = 1
 while (i <= n) {
   res = rec(i, res)
   i += 1
 }
 res
}
schema(Nil, (i: Int, xs: List[Int]) ⇒ i +: xs)(10)          // List(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)

schema("", (i: Int, xs: String) ⇒ s"$i/$xs")(10)          // 10/9/8/7/6/5/4/3/2/1/
```

Matryoshka

Les poupées russes

“Generalized folds, unfolds, and traversals for fixed point data structures in Scala.”

Recursion Schemes

folds (tear down a structure)
algebra f a → Fix f → a

↔

unfolds (build up a structure)
coalgebra f a → a → Fix f

	folds	unfolds	
generalized (f w → w f) → (f (w α) → β)	**cata**morphism f a → a	**ana**morphism a → f a	**g**eneralized (m f → f m) → (α → f (m β))
	prepromorphism* … after applying a NatTrans (f a → a) → (f → f)	**postpro**morphism* … before applying a NatTrans (a → f a) → (f → f)	
	paramorphism* … with primitive recursion f (Fix f × a) → a	**apo**morphism* … returning a branch or single level a → f (Fix f ∨ a)	
	zygomorphism* … with a helper function (f b → b) → (f (b × a) → a)	**g apo**morphism (b → f b) → (a → f (b ∨ a))	
g histomorphism (f h → h f) → (f (w a) → a)	**histo**morphism … with prev. answers it has given f (w a) → a	**futu**morphism … multiple levels at a time a → f (m a)	**g futu**morphism (h f → f h) → (a → f (m a))

refolds (build up then tear down a structure)
algebra g b → (f → g) → *coalgebra f a* → a → b

hylomorphism cata; ana		**g**eneralized apply the generalizations for both the relevant fold and unfold
dynamorphism histo; ana	**codyna**morphism cata; futu	
chronomorphism histo; futu		
Elgot algebra … may short-circuit while building cata; a → b ∨ f a	**coElgot** algebra … may short-circuit while tearing a × g b → b; ana	

reunfolds (tear down then build up a structure)
coalgebra g b → (a → b) → *algebra f a* → Fix f → Fix g

metamorphism ana; cata	**g**eneralized apply … both … [un]fold

combinations (combine two structures)
algebra f a → Fix f → Fix f → a

zippamorphism f a → a
mergamorphism … which may fail to combine (f (Fix f) × f (Fix f)) ∨ f a → a

others

synchromorphism ???
exomorphism ???
mutumorphism ???

Stolen from Edward Kmett's http://comonad.com/reader/2009/recursion-schemes/
* This gives rise to a family of related recursion schemes, modeled in recursion-schemes with distributive law combinators

These can be combined in various ways. For example, a "zygohistomorphic prepromorphism" combines the zygo, histo, and prepro aspects into a signature like *(f b → b) → (f → f) → (f (w (b × a)) → a) → Fix f → a*

```scala
sealed trait IntList[T]
case class Empty[T]() extends IntList[T]
case class Cons[T](head: Int, tail: T) extends IntList[T]
```

But

- Construire des structures récursives
- Pour faire du calcul récursif
- Manipuler facilement des types complexes

Remarque

```
val intList: ??? = Cons(1, Cons(2, Cons(3, Empty())))
```

Remarque

```
val intList:  IntList[Cons[Cons[Empty[Nothing]]]]  =
Cons(1, Cons(2, Cons(3, Empty())))
```

Fixed Points

- Il faut un point fixe de IntList
- Idéalement un type T tel que T == IntList[T]
- Fix[_] : Fix[IntList] == IntList[Fix[IntList]] == IntList[IntList[Fix[IntList]]]
- On a bien Fix[IntList] point fixe de IntList

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

val fixIntList: Fix[IntList] =
Fix(Cons(1, Fix(Cons(2, Fix(Cons(1, Fix(Cons(2, Fix(Empty())))))))))

fixIntList.unFix
// Cons(1,Fix(Cons(2,Fix(Cons(1,Fix(Cons(2,Fix(Empty()))))))))
```

Etape 1: Functor

```
implicit val listFunc = new Functor[IntList] {

 override def map[A, B](fa: IntList[A])(f: A ⇒ B): IntList[B] = fa match {
   case Empty()                          ⇒ Empty[B]()
   case Cons(head: Int, tail: A)  ⇒ Cons(head, f(tail))
 }

}
```

Anamorphism

Du haut vers le bas

Généralisation du unfold

```
type Coalgebra[F[_], A] = A => F[A]
```

Etape 2: Coalgebra

```scala
val coalgebraInt: Coalgebra[IntList, Int] = {
 case 0 ⇒ Empty()
 case n ⇒ Cons(n, n - 1)
}
```

```scala
type Coalgebra[F[_], A] = A => F[A]
```

Etape 2: Coalgebra

```
def apply[F[_]: Functor] (f: Coalgebra[F, A])(implicit T: Corecursive.Aux[T, F]) : T

type Coalgebra[F[_], A] = A => F[A]

val coalgebraInt: Coalgebra[IntList, Int] = {
 case 0 ⇒ Empty()
 case n ⇒ Cons(n, n - 1)
}
```

```
4.ana[Fix[IntList]](coalgebraInt)
// Fix(Cons(4,Fix(Cons(3,Fix(Cons(2,Fix(Cons(1,Fix(Empty())))))))))
```

Catamorphism

Du bas vers le haut

Généralisation du fold sur les ADT

```
type Algebra[F[_], A] = F[A] => A
```

Etape 2: Algebra

```
val algebraProduct: Algebra[IntList, Int] = {
 case Empty()         ⇒ 1
 case Cons(head, tail) ⇒ head * tail
}

val algebraSum: Algebra[IntList, Int] = {
 case Empty()         ⇒ 0
 case Cons(head, tail) ⇒ head + tail
}

val algebraBool: Algebra[IntList, Boolean] = {
 case Empty()         ⇒ true
 case Cons(_, tail)   ⇒ !tail
}
```

```
type Algebra[F[_], A] = F[A] => A
```

Etape 2: Algebra

```
val fixIntList: Fix[IntList] =
Fix(Cons(1, Fix(Cons(2, Fix(Cons(1, Fix(Cons(2, Fix(Empty())))))))))

def cata[A](f: Algebra[F, A])(implicit BF: Functor[F]): A

fixIntList.cata(algebraProduct)   // 4
fixIntList.cata(algebraSum)       // 6
fixIntList.cata(algebraBool)      // true
```

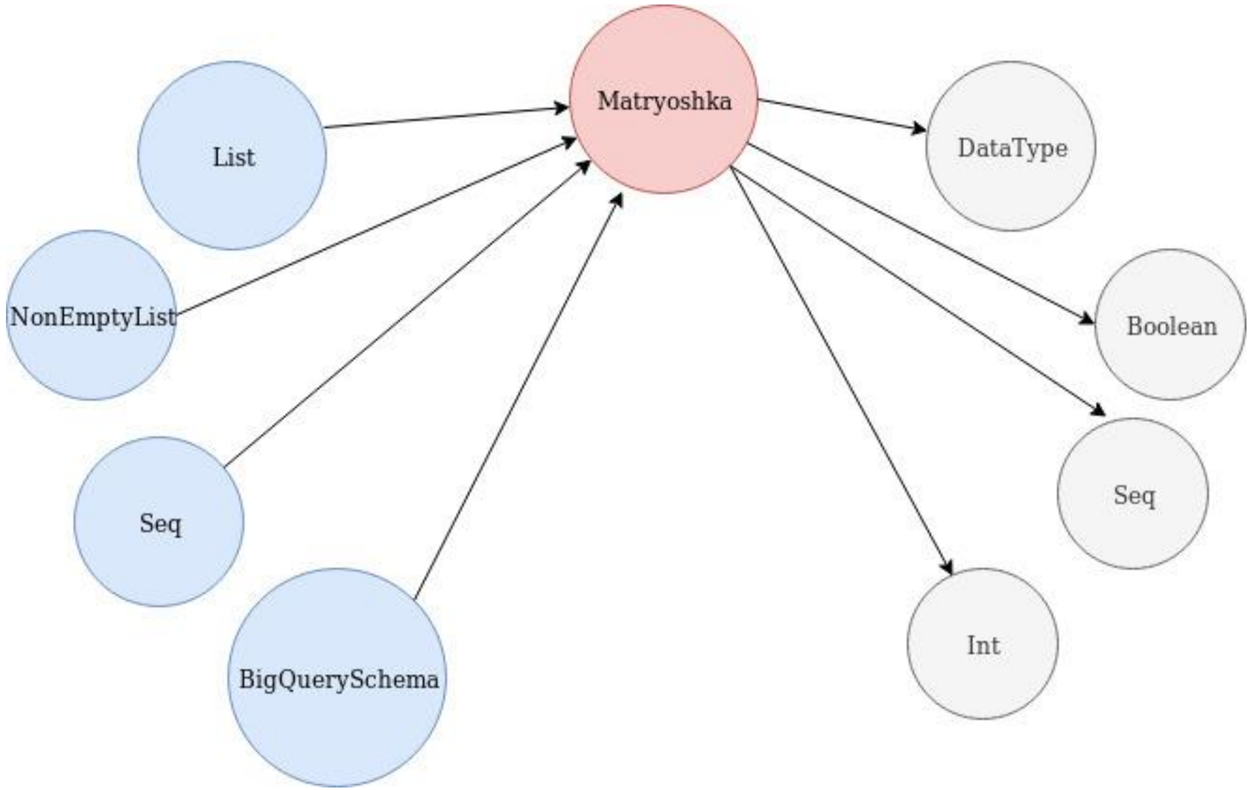
Etape 3: Hylomorphism

```
def hylo[F[_]: Functor, B](f: Algebra[F, B], g: Coalgebra[F, A]): B = matryoshka.hylo(self)(f, g)
def hylo[F[_]: Functor, A, B](a: A)(φ: Algebra[F, B], ψ: Coalgebra[F, A]): B = φ(ψ(a) ◦ (hylo(_)(φ, ψ)))
final def ◦[B](f: A => B): F[B] = F.map(self)(f)

val algebraBool: Algebra[IntList, Boolean]
val algebraProduct: Algebra[IntList, Int]
val algebraSum: Algebra[IntList, Int]
val coalgebraInt: Coalgebra[IntList, Int]

def even(n: Int): Boolean       = n.hylo(algebraBool, coalgebraInt)
def factorial(n: Int): Int  = n.hylo(algebraProduct, coalgebraInt)
def sum(n: Int): Int                 = n.hylo(algebraSum, coalgebraInt)
```

```
factorial(4)    // 24
sum(4)          // 10
even(4)         // true
```



Problématique

- Ne pas écrire 100 objets si on a 100 data sources (ou si la requête change)
  - Inférer n’importe quel schéma
- Mapping de schéma
  - BigQuery ⇔ Spark
  - Json ⇔ Avro

```scala
sealed trait SchemaR[T]

final case class StructR[T](fields: Map[String, T]) extends SchemaR[T]
final case class ArrayR[T](element: T) extends SchemaR[T]
final case class IntR[T]() extends SchemaR[T]
final case class StringR[T]() extends SchemaR[T]
final case class TimestampR[T]() extends SchemaR[T]
final case class DoubleR[T]() extends SchemaR[T]
final case class BooleanR[T]() extends SchemaR[T]
```

```
implicit val schemaRFunctor: Functor[SchemaR] = new Functor[SchemaR] {

 override def map[A, B](fa: SchemaR[A])(f: A ⇒ B): SchemaR[B] = fa match {

   case StructR(fields)  ⇒ StructR(fields.map { case (k, v) ⇒ k -> f(v) })
   case ArrayR(e)        ⇒ ArrayR(f(e))
   case IntR()           ⇒ IntR[B]()
   case StringR()        ⇒ StringR[B]()
   case TimestampR()     ⇒ TimestampR[B]()
   case DoubleR()        ⇒ DoubleR[B]()
   case BooleanR()       ⇒ BooleanR[B]()

 }

}
```

Unfold

```
/* Construct SchemaR from List[Field] */

val coalgebra: Coalgebra[SchemaR, List[Field]] = {
 case Nil             ⇒ StructR(Map.empty)
 case x :: Nil        ⇒ x.getType match {
   case LegacySQLTypeName.STRING        ⇒ StringR()
   case LegacySQLTypeName.INTEGER       ⇒ IntR()
   case LegacySQLTypeName.TIMESTAMP     ⇒ TimestampR()
   case LegacySQLTypeName.NUMERIC       ⇒ DoubleR()
   case LegacySQLTypeName.FLOAT         ⇒ DoubleR()
   case LegacySQLTypeName.BOOLEAN       ⇒ BooleanR()
   case LegacySQLTypeName.RECORD        ⇒ StructR(Map(x.getName -> List(x)))
 }
 case x :: xs         ⇒ StructR((x :: xs).map { f ⇒ f.getName -> List(f) }.toMap)
}
```

Fold

```
/* Deconstruct SchemaR to DataType */

val algebra: Algebra[SchemaR, DataType] = {
 case StructR(fields)     ⇒ StructType(fields.map { case (k, v) ⇒ StructField(k, v) }.toSeq)
 case ArrayR(e)           ⇒ ArrayType(e)
 case IntR()              ⇒ IntegerType
 case StringR()           ⇒ StringType
 case TimestampR()        ⇒ TimestampType
 case DoubleR()           ⇒ DoubleType
 case BooleanR()          ⇒ BooleanType
}
```

```
implicit val schemaRFunctor: Functor[SchemaR]
val coalgebra: Coalgebra[SchemaR, List[Field]]       // List[Field] => SchemaR[List[Field]]
val algebra: Algebra[SchemaR, DataType]              // SchemaR[DataType] => DataType

public final class Field implements Serializable {
 private final String name;
 private final LegacySQLTypeName type;
 private final FieldList subFields;
 private final String mode;
 private final String description;
}
def getDataType(listField: List[Field]): DataType = listField.hylo(algebra, coalgebra)
```

https://speakerdeck.com/ogirardot/high-performance-privacy-by-design-using-matryoshka-and-spark

https://www.youtube.com/watch?v=HyHRI5LzVMk