

Recursion Schemes



Our Speakers



Nicolas François
Software Engineer
nfrancois@mediarithmics.com
mediarithmics



Virgile Quintin
Software Engineer
vquintin@mediarithmics.com
mediarithmics

Contents

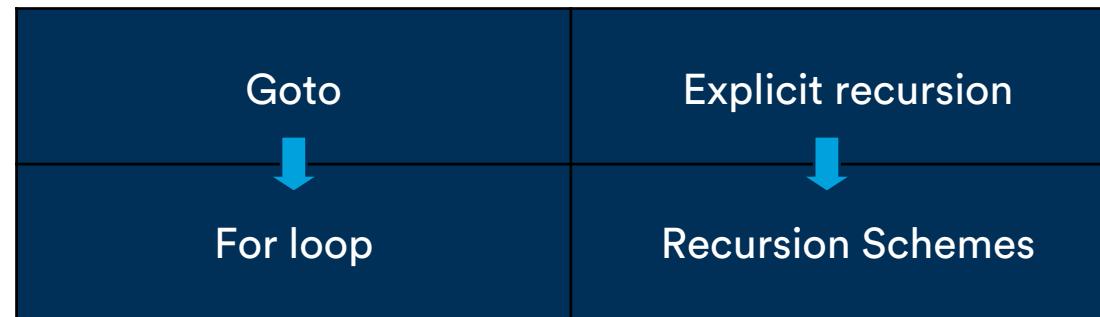
- Why recursion schemes are useful ?
- What are they ?
- Quick definition
- Folding
- Unfolding
- Refolding
- Demo
- Conclusion

Why recursion scheme?

- Recursive functions can be hard to maintain over time (as any complex control flow).
- Explicit recursion can be hard to test
- Guarantee to finish (if you want too)

What are recursion schemes?

Recursion schemes: work with recursive data structures without explicit recursion.



More generic, more readable, more testable

One more tool to help manage complex code

What are recursion schemes?

Bonus:

It allows us to easily
write performant stack safe code !*



*offer subject to conditions

Already known technique

```
List(1, 2, 3).foldl(0)(_ + _)
```

// foldl removes the explicit recursion.

*// FOLD, REDUCE, ... already are
recursion scheme*



Quick Definitions



Recursive data structures

// A type which holds a reference to itself:

```
sealed abstract class List[+A]
final case class Cons[+A](head: A, tail: List[A]) extends List[A]
final case object Nil extends List[Nothing]
```



Explicit recursion

//A function which calls itself explicitly:

```
def length[A](list: List[A]) = list match {  
    case h::t => 1 + length(t)  
    case Nil => 0  
}
```



Functor

```
def map(fa: Foo[A])(f: A => B): Foo[B] = ???
```

Functor

```
case class Foo[A](...)  
object Foo {  
    implicit val functor: Functor[A] =  
        new Functor[A]{  
            def map(fa: Foo[A])(f: A => B): Foo[B] = ???  
        }  
}
```



Separating concern

A recursive data structure has two things:

- A shape: list, tree
- Recursiveness

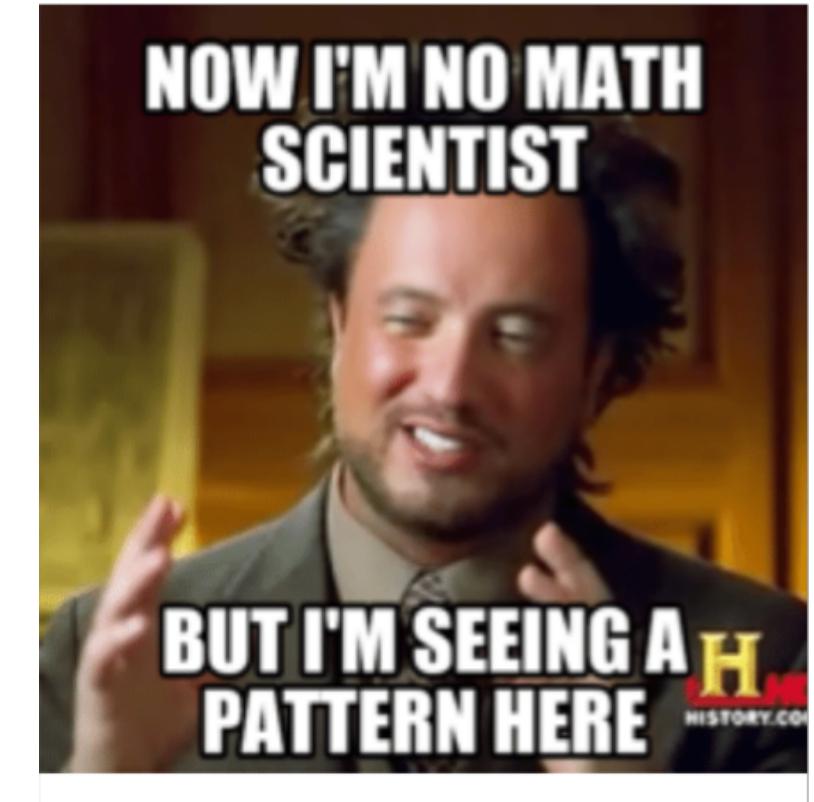
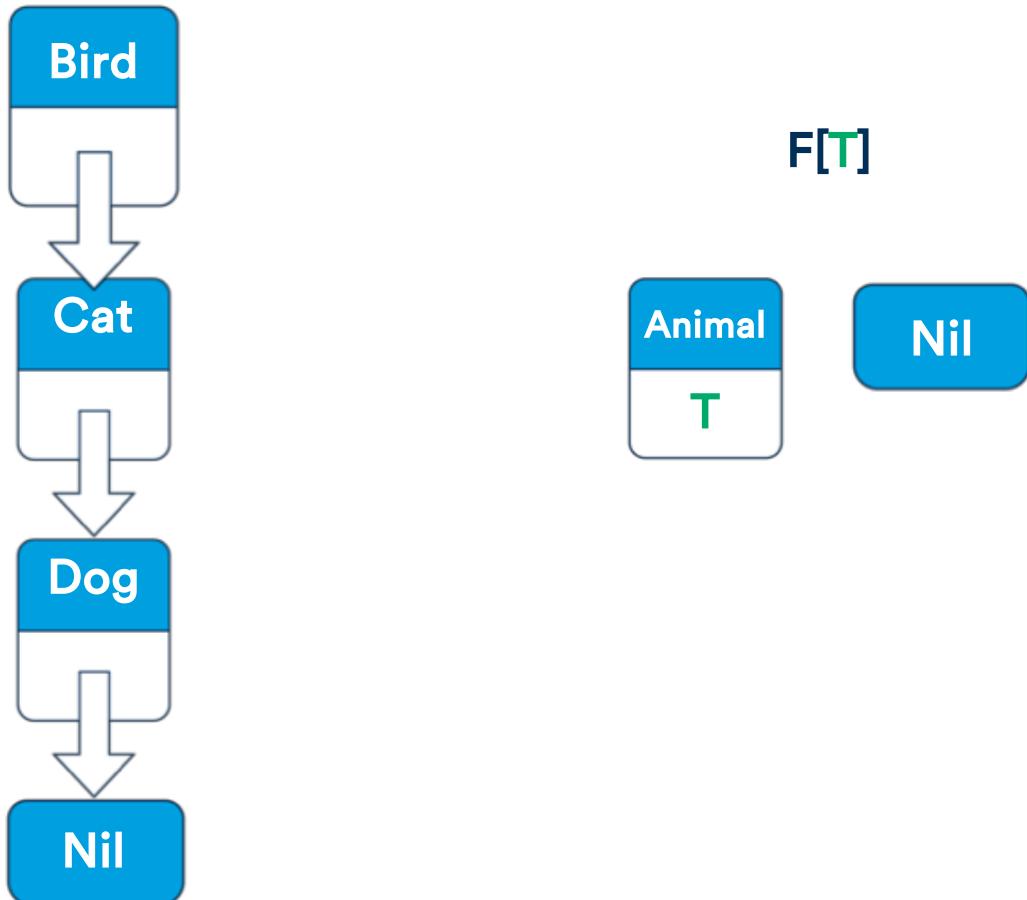
Separating concern

A recursive data structure has two things:

- A shape: list, tree  Pattern functor
- Recursiveness  Fix

Pattern functor and Fix help separating the concerns in the recursive data structure definition.

Pattern Functor



Pattern Functor

```
sealed trait ListF[+E, +T]
case class NilF() extends ListF[Nothing, Nothing]
case class ConsF[E, T](head: E, tail: T) extends ListF[E, T]

type F[T] = ListF[String, T]
```

Fix: type problems

```
ConsF(1, NilF)
```

// res0: ConsF[Int, NilF]

```
val b = ConsF(1, ConsF(2, NilF))
```

// res1: ConsF[Int, ConsF[Int, NilF]]

//Put lists of every length under the same type umbrella !



Fix: the trick

```
case class Fix[F[_]](unFix: F[Fix[F]])  
  
type List[A] = Fix[List[A, ?]]
```

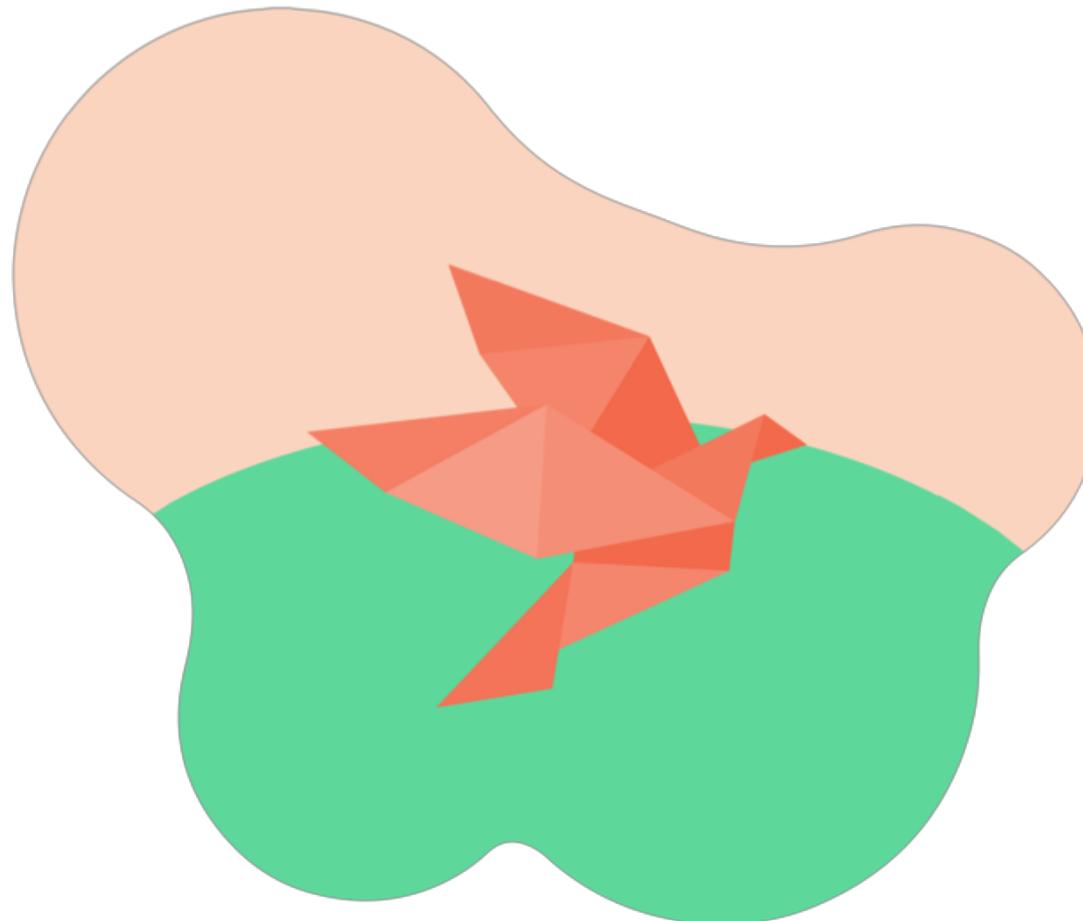


Fix

```
val a: List[Int] = Fix(ConsF(1, Fix(NilF)))  
// Fix[ConsF[Int, ?]]  
val b: List[Int] = Fix(ConsF(1, Fix(ConsF(2, Fix(NilF())))))  
// Fix[ConsF[Int, ?]]
```



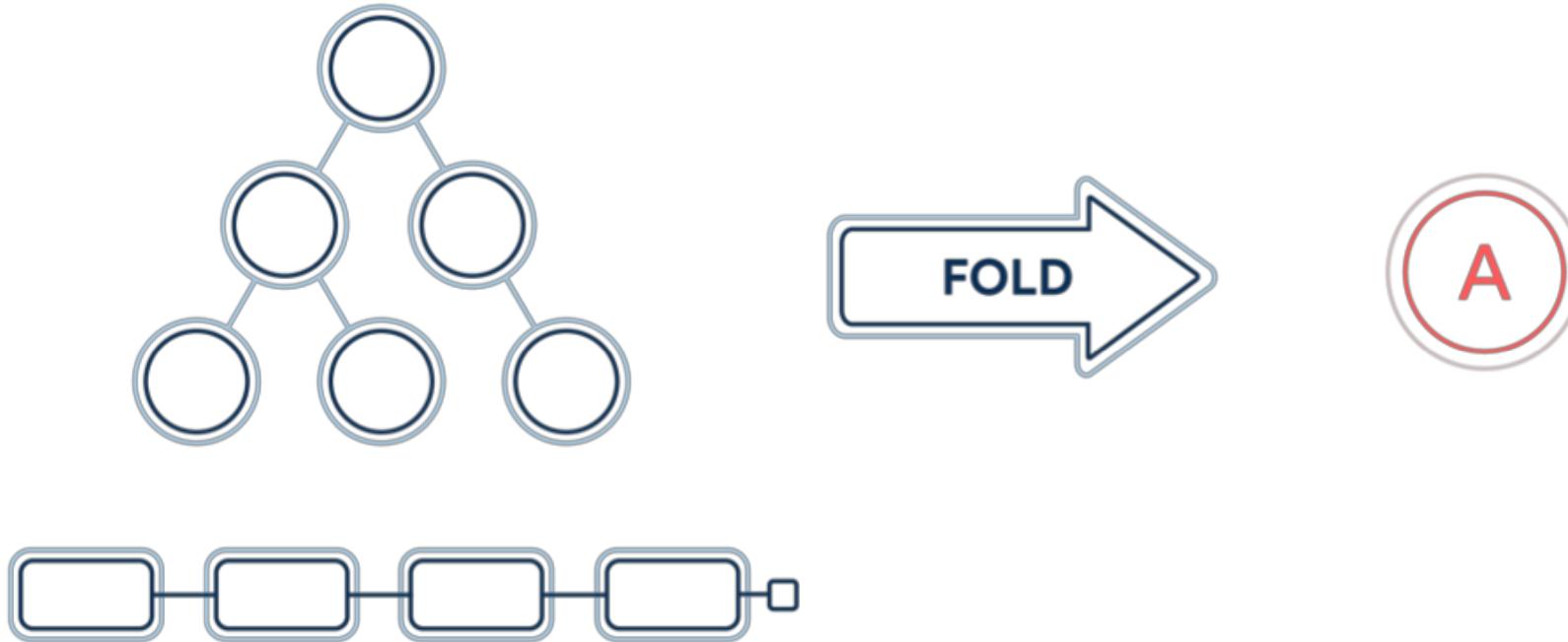
Types of recursion schemes



Types of recursion schemes

- Folds
- Unfolds
- Refolds

Fold



Folds: catamorphism

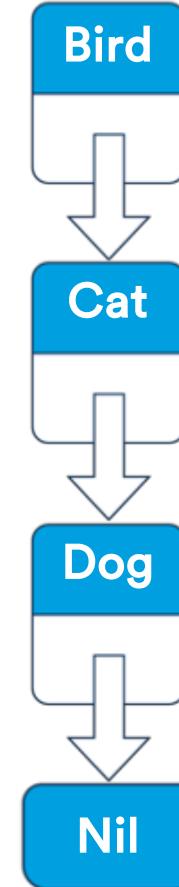
```
def cata[F[_]: Functor, A](f: F[A] => A)(v: Fix[F]): A
```

- One level at a time
- From greek κατά as in catastrophe, catalyst, catapult



Example: length of a list of String

```
def cata[F: Functor, A](f: F[A] => A)(c: Fix[F]): A = ???  
cata[ListF[String, ?]], Int]{  
    case ConsF(_, previousLength) => 1 + previousLength  
    case NilF() => 0  
}(list)
```



Example: length of a list of String

```
def cata[F: Functor, A](f: F[A] => A)(c: Fix[F]): A = ???  
cata[ListF[String, ?]], Int]{  
    case ConsF(_, previousLength) => 1 + previousLength  
    case NilF() => 0  
}(list)
```



Example: length of a list of String

```
def cata[F: Functor, A](f: F[A] => A)(c: Fix[F]): A = ???  
cata[ListF[String, ?]], Int]{  
    case ConsF(_, previousLength) => 1 + previousLength  
    case NilF() => 0  
}(list)
```



Example: length of a list of String

```
def cata[F: Functor, A](f: F[A] => A)(c: Fix[F]): A = ???  
cata[ListF[String, ?]], Int]{  
    case ConsF(_, previousLength) => 1 + previousLength  
    case NilF() => 0  
}(list)
```

Bird
2



Example: length of a list of String

```
def cata[F: Functor, A](f: F[A] => A)(c: Fix[F]): A = ???  
cata[ListF[String, ?]], Int]{  
    case ConsF(_, previousLength) => 1 + previousLength  
    case NilF() => 0  
}(list)
```

2

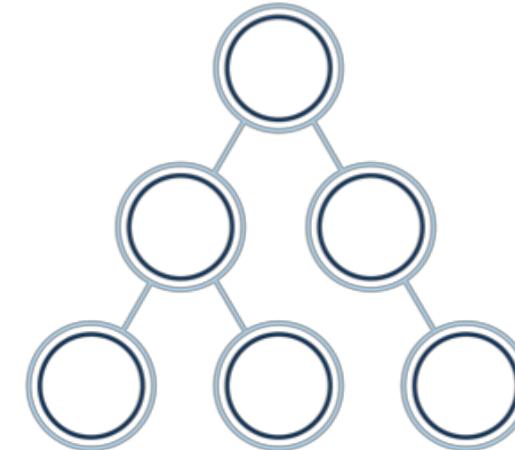
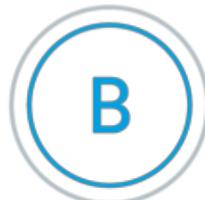


Testability through Pattern functor

```
assert(f(Nil) === 0)  
assert(f(ConsF('a', 0)) === 1)  
assert(f(ConsF('a', 42)) === 43)
```

Unfolds

Picture of a value on the left
and a tree on the right



Unfolds: anamorphism

```
def ana[F[_]: Functor, A](f: A => F[A])(a: A): Fix[F] = ???
```

- One level at a time
- From greek $\alpha\omega$ (up) as in anatomy, analogy, analyse, anabolic



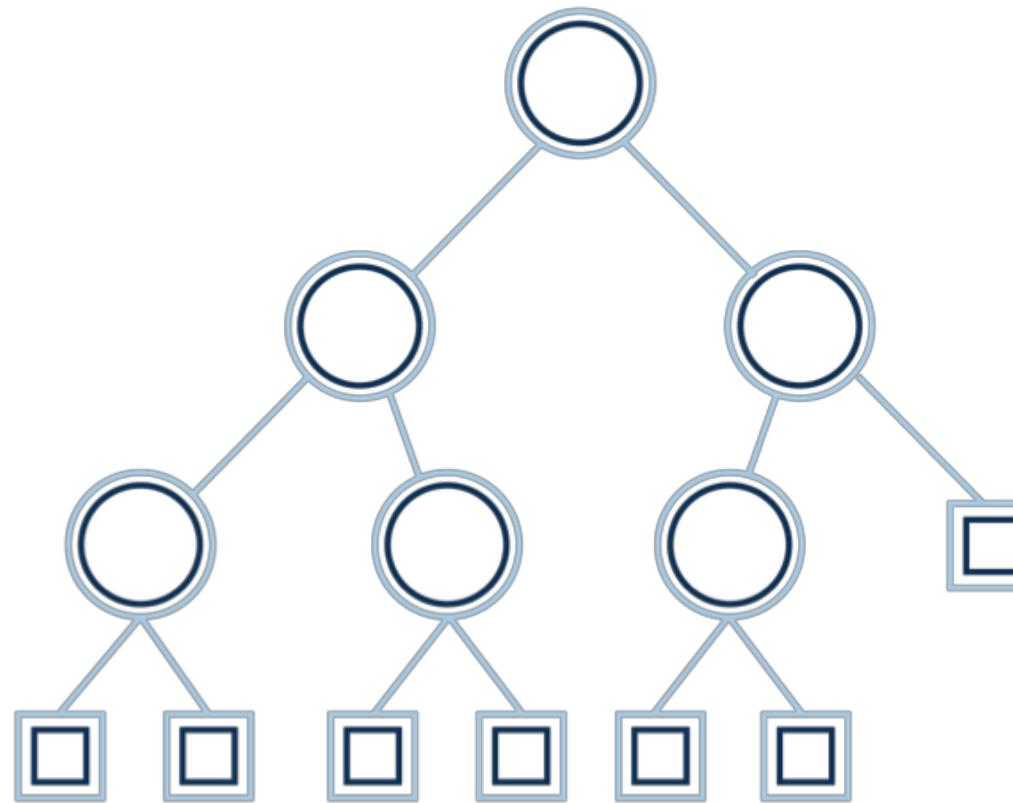
Unfolds: anamorphism

```
def ana[F[_]: Functor, A](f: A => F[A])(a: A)    : Fix[F] = ???  
def cata[F[_]: Functor, A](f: F[A] => A)(c: Fix[F]): A    = ???
```

- One level at a time
- From greek ἀνά (up) as in anatomy, analogy, analyse, anabolic



Example: balanced bst from sorted string List



Example: balanced bst from sorted string List

```
sealed trait SearchTreeF[+E, +T]
case class BranchF[E, B](value: E, left: T, right:
T) extends SearchTreeF[E, T]
case class LeafF() extends
SearchTreeF[Nothing, Nothing]
type F[T] = SearchTreeF[String, T]
```



Example: balanced bst from sorted string List

```
ana{  
    case Nil => LeafF()  
    case list =>  
        val (left, v, right) = splitList(list)  
        BranchF(v, left, right)  
}(myList)
```

[“bird”, “cat”,
“dog”, “fish”]



Example: balanced bst from sorted string List

```
ana{  
    case Nil => LeafF()  
    case list => [+E, +T]  
        val (left, v, right) = splitList(list)  
        BranchF(v, left, right)  
    }(myList)
```

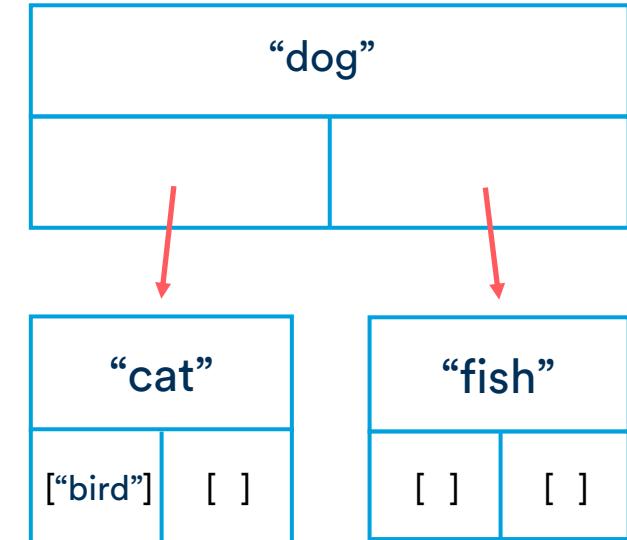
“dog”	
[“bird”, “cat”]	[“fish”]



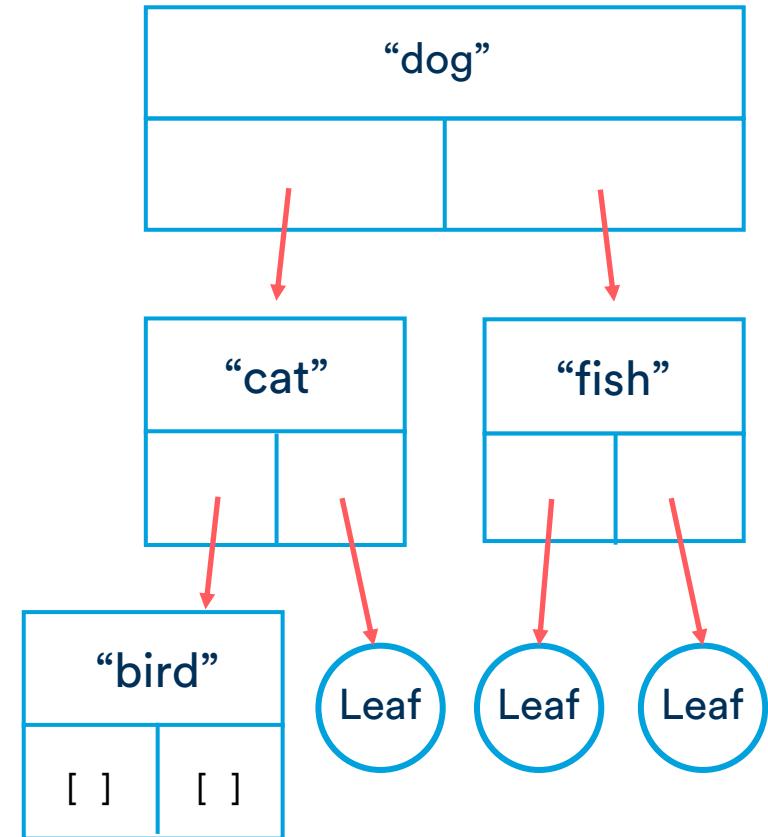
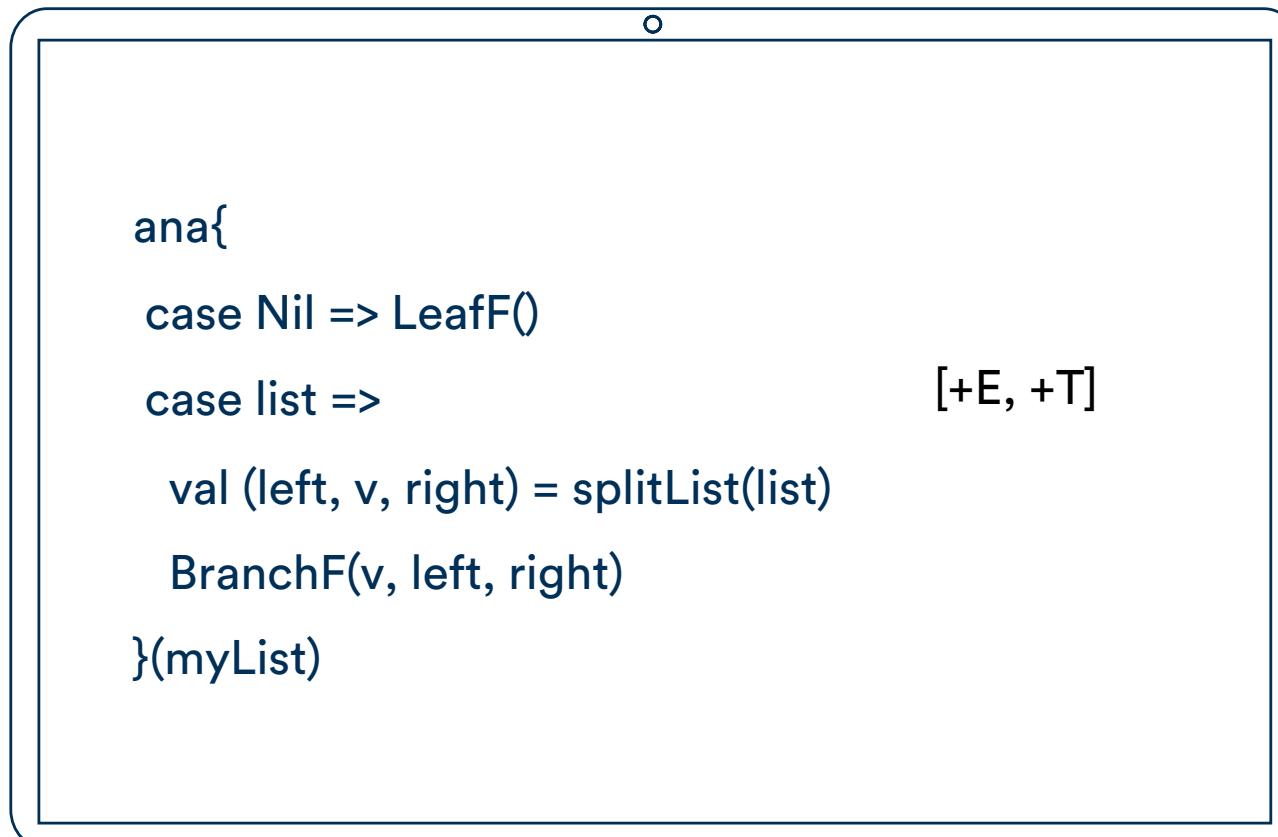
Example: balanced bst from sorted string List

```
ana{  
  case Nil => LeafF()  
  case list =>  
    val (left, v, right) = splitList(list)  
    BranchF(v, left, right)  
}(myList)
```

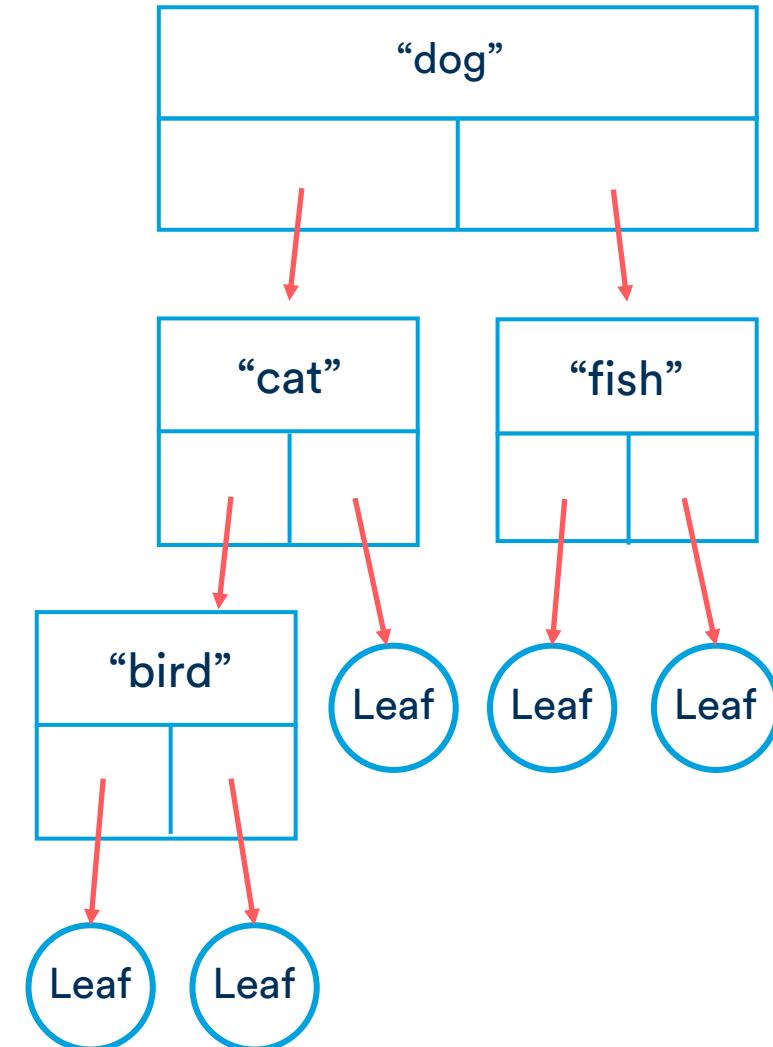
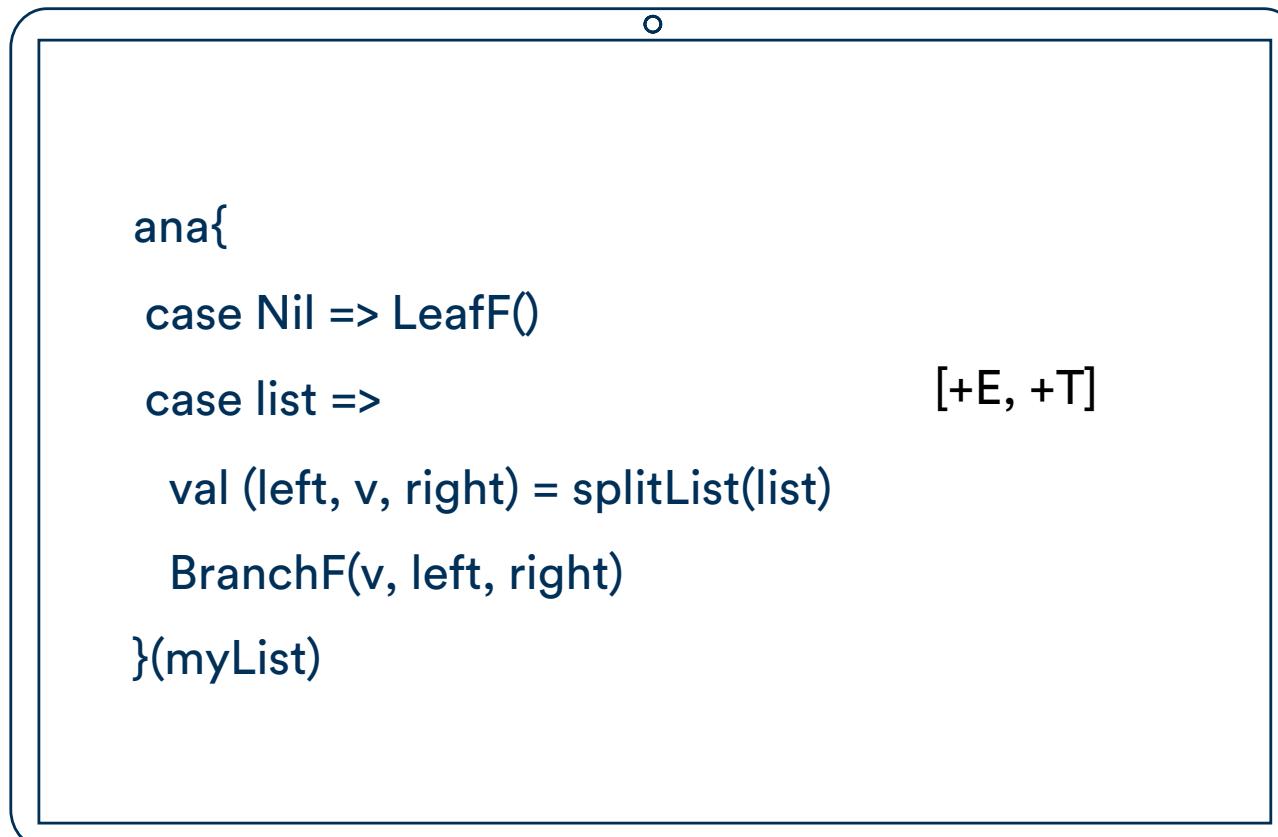
[+E, +T]



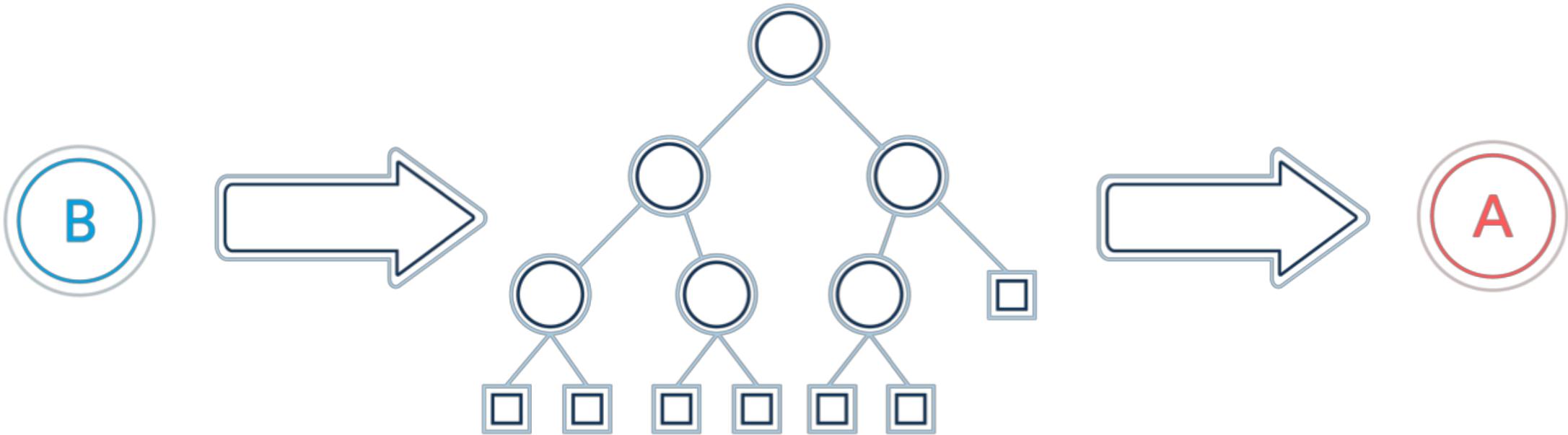
Example: balanced bst from sorted string List



Example: balanced bst from sorted string List

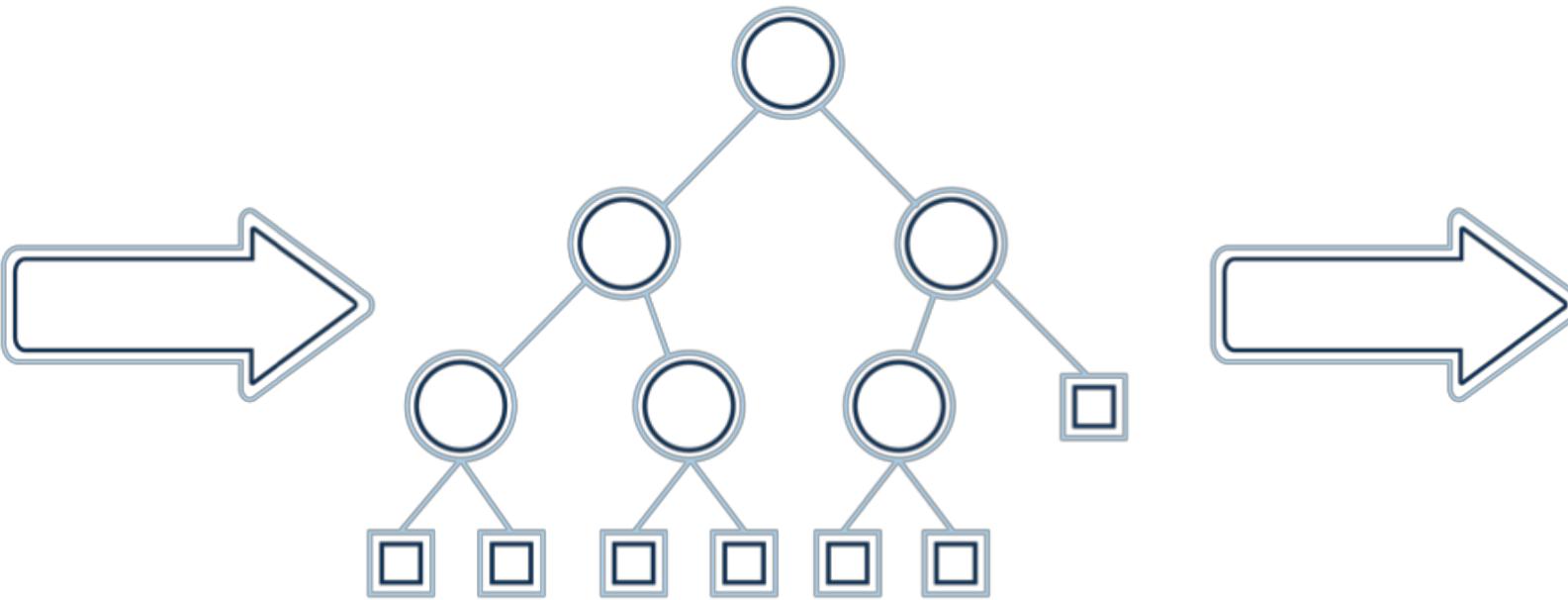


Refold



Refold

<https://url.com>



42

Refolds: hylomorphism

- Building one step at a time
- Consuming one step at time
- It is just ana and then cata !
- But more efficient

Refolds: hylomorphism

```
def hylo[F[_], A, B](g: A => F[A])(f: F[B] => B)(a: A)(c:Fix[F]): B = ???
```

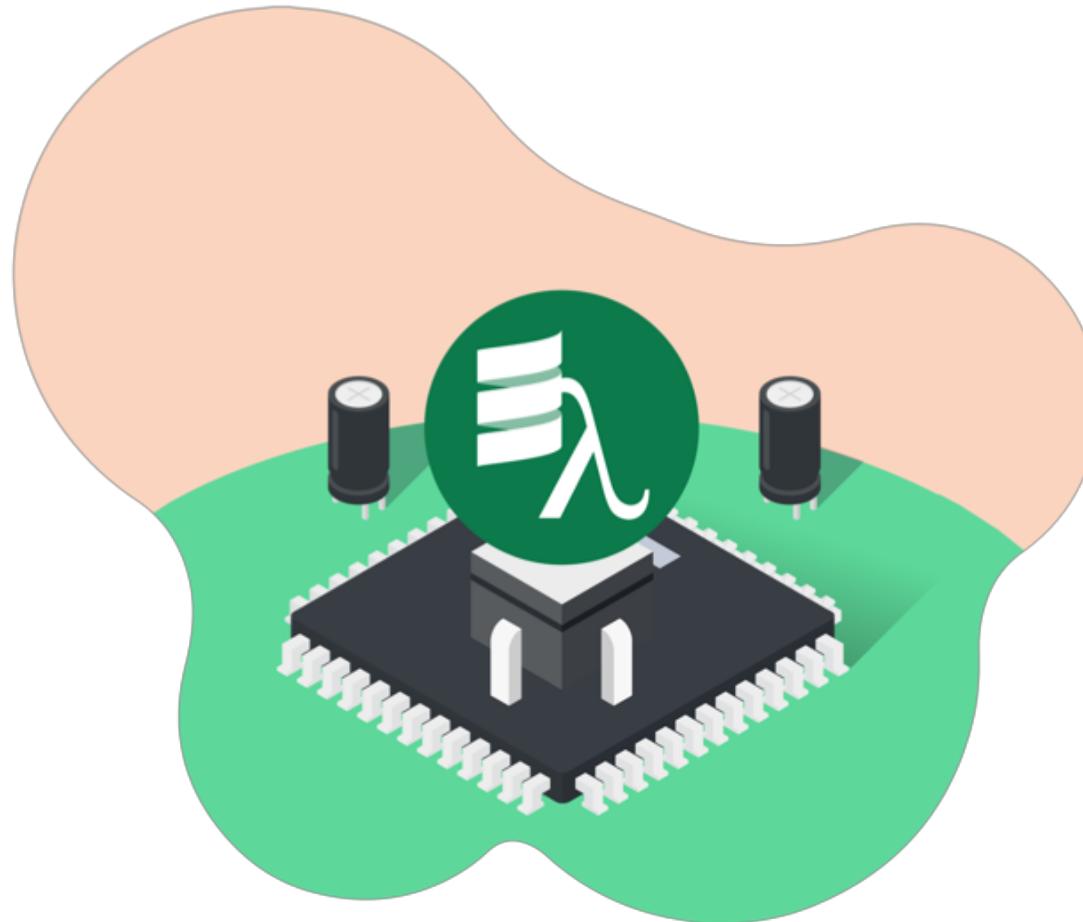


Refolds: hylomorphism

```
def cata[F[_]: Functor, B] (f: F[B] => B)      (c: Fix[F]): B = ???  
def ana[F[_]: Functor, A]  (g: A => F[A])  (a: A) : Fix[F] = ???  
def hylo[F[_]: Functor, A, B](g: A => F[A])(f: F[B] => B)(a: A)(c: Fix[F]) : B = ???
```



Types of recursion schemes



Example: Factorial

Computing a factorial:

- building a list: `List(n, ..., 3, 2, 1)`
- computing the product of all the numbers in the list

```
def factorial(n: Int): Int = (1 to n).foldl(1)(_ * _)
```



Example: Factorial

```
○  
val fold: ListF[Int, Int] => Int = {  
    case ConsF(n, previousProduct) = n * previousProduct  
    case NilF() = 1  
}  
val unfold: Int => ListF[Int, Int] = {  
    case 0 => NilF()  
    case n => ConsF(n, n-1)  
}  
def fac(n: Int) = hylo(unfold)(fold)(n)
```

3



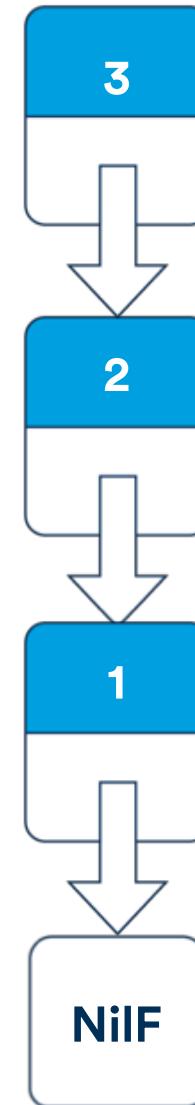
Example: Factorial

```
○  
val fold: ListF[Int, Int] => Int = {  
    case ConsF(n, previousProduct) = n * previousProduct  
    case NilF() = 1  
}  
val unfold: Int => ListF[Int, Int] = {  
    case 0 => NilF()  
    case n => ConsF(n, n-1)  
}  
def fac(n: Int) = hylo(unfold)(fold)(n)
```



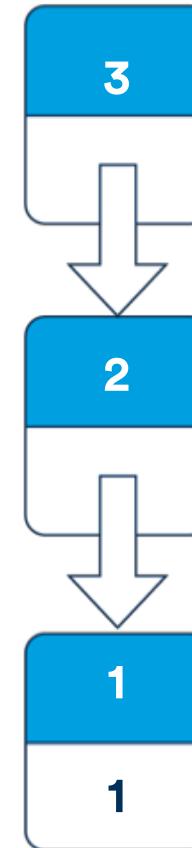
Example: Factorial

```
val fold: ListF[Int, Int] => Int = {  
    case ConsF(n, previousProduct) = n * previousProduct  
    case NilF() = 1  
}  
val unfold: Int => ListF[Int, Int] = {  
    case 0 => NilF  
    case n => ConsF(n, n-1)  
}  
def fac(n: Int) = hylo(unfold)(fold)(n)
```



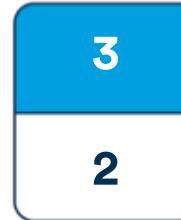
Example: Factorial

```
○  
val fold: ListF[Int, Int] => Int = {  
    case ConsF(n, previousProduct) = n * previousProduct  
    case NilF() = 1  
}  
val unfold: Int => ListF[Int, Int] = {  
    case 0 => NilF  
    case n => ConsF(n, n-1)  
}  
def fac(n: Int) = hylo(unfold)(fold)(n)
```



Example: Factorial

```
○  
val fold: ListF[Int, Int] => Int = {  
    case ConsF(n, previousProduct) = n * previousProduct  
    case NilF() = 1  
}  
val unfold: Int => ListF[Int, Int] = {  
    case 0 => NilF()  
    case n => ConsF(n, n-1)  
}  
def fac(n: Int) = hylo(unfold)(fold)(n)
```



3
2



Example: Factorial

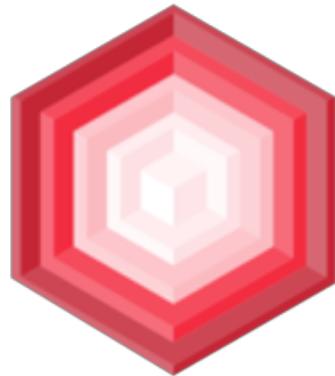
```
○  
val fold: ListF[Int, Int] => Int = {  
    case ConsF(n, previousProduct) = n * previousProduct  
    case NilF() = 1  
}  
val unfold: Int => ListF[Int, Int] = {  
    case 0 => NilF()  
    case n => ConsF(n, n-1)  
}  
def fac(n: Int) = hylo(unfold)(fold)(n)
```

6



Demo

Main Libraries



Matryoshka
([ScalaZ](#))



Droste
([Typelevel](#))

Useful knowledge when working with libraries

- The data structure are encapsulate in Algebra(fold) and CoAlgebra(unfold)
- Some other (un)folding function exist to embody final stop or other special behavior.
However this can be simpler to embed this behavior in the data structure
- Also exist with an effect:

```
def cataM[F[_] : Traverse, M[_] : Monad, A](f: F[A] => M[A])(v: Fix[F]): M[A]
```

Useful Resources

- <https://github.com/passy/awesome-recursion-schemes>
- <https://blog.sumtypeofway.com/an-introduction-to-recursion-schemes/>
- <http://free.cofree.io/2017/11/13/recursion/>
[\(https://github.com/zliu41/zliu41.github.io/blob/master/_posts/2017-11-13-recursion.md\)](https://github.com/zliu41/zliu41.github.io/blob/master/_posts/2017-11-13-recursion.md)
- Both Droste and matryoshka have an awesome gitter
- <https://www.youtube.com/watch?v=oRLkb6mqvVM&list=PL9KXdMOfekvYYX53KHR-28e3BjITIEjbT&index=5>
- <https://www.youtube.com/watch?v=XZ9nPZbaYfE>
- <https://www.youtube.com/watch?v=zkDVCQiveEo>

Advantages and drawbacks

Advantages:

- Better readability for complexe cases
- More efficient than naive implementations (Template Driven Development)
- Stack safe (if the data structure is stack safe)

Drawbacks:

- Learning curve
- Documentation is very *academic*
- Lot of boilerplate for simple recursion

Thank you !

