All about a fold*

GClaramunt
Scalents

* Not all, actually

You could've invented fold...

How to sum all elements of a list? (in Scala)

```
List(1, 7, 4, 11, 3, 9)

def sum(nums: List[Int]): Int = ???
```

How to sum all elements of a list? (in Scala)

```
List(1, 7, 4, 11, 3, 9)

def sum(nums: List[Int]): Int = nums match {

  case Nil => ???

  case x::xs => ???

}
```

How to sum all elements of a list? (in Scala)

```
List(1, 7, 4, 11, 3, 9)

def sum(nums: List[Int]): Int = nums match {
  case Nil => 0
  case x::xs => x + sum(xs)
}
```

Convert to string all elements of a list? (in Scala)

```
List(1, 7, 4, 11, 3, 9)

def toString(nums: List[Int]): String = nums match {

  case Nil => ???

  case x::xs => ???

}
```

Convert to string all elements of a list? (in Scala)

```
List(1, 7, 4, 11, 3, 9)

def toString(nums: List[Int]): String = nums match {
  case Nil => “”
  case x::xs => x ++ toString(xs)
}
```

all elements of a list satisfy a property? (in Scala)

```
List(1, 7, 4, 11, 3, 9)

def all[A](prop: A => Bool)(l: List[A]): Bool = l match {
  case Nil => ???
  case x::xs => ???
}
```

all elements of a list satisfy a property? (in Scala)

```
List(1, 7, 4, 11, 3, 9)

def all[A](prop: A => Bool)(l: List[A]): Bool = l match {
  case Nil => True
  case x::xs => prop(x) && all(p)(xs)
}
```

How to sum all elements of a list? (Haskell)

```
[1, 7, 4, 11, 3, 9]

sum :: [Int] -> Int

sum [] = 0

sum (x:xs) = x + sum xs
```

Convert to string all elements of a list? (Haskell)

```
[1, 7, 4, 11, 3, 9]

toString :: [Int] -> String

toString [] = “”

toString (x:xs) = show x ++ toString xs
```

ALL ELEMENTS OF A LIST SATISFY A PROPERTY? (HASKELL)

```
[1, 7, 4, 11, 3, 9]

all :: ( a->Bool ) -> [a] -> Bool

all _ [] = True

all p (x:xs) = p x && all p xs
```

How we did recursion?

We have:

- One definition for the empty case
- One definition for the head/tail case

How we did recursion?

We have:

- One definition for the empty case
- One definition for the head/tail case

We are doing recursion in the structure of the list...

Let's generalize!

Let's generalize!

```
def sum(nums: List[Int]): Int = nums match {
  case Nil => 0
  case x::xs => x + sum(xs)
}

def toString(nums: List[Int]): String = nums match {
  case Nil => “”
  case x::xs => x ++ toString(xs)
}

def all[A](prop: A => Bool)(l: List[A]): Bool = l match {
  case Nil => True
  case x::xs => prop(x) && all(p)(xs)
}
```

Let's generalize!

A value z for the empty case

A function f for the head/tail case that combines the head with the result of the recursive call on the tail

Let's generalize!

A value z for the empty case

A function f for the head/tail case that combines the head with the result of the recursive call on the tail

```
def recList[A,B](z: B)(f: (A,B) => B)(l: List[A]): B =
l match {
  case Nil => z
  case x:xs => f(x, recList(z)(f)(xs)
}
```

Let's generalize!

A value z for the empty case

A function f for the head/tail case that combines the head with the result of the recursive call on the tail

```
def recList[A,B](z: B)(f: (A,B) => B)(l: List[A]): B =
l match {
  case Nil => z
  case x:xs => f(x, recList(z)(f)(xs))
}
```

foldRight!

FOLD !

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Usually “given a function f that combines an element with the accumulation and an initial value z, starting with z traverses the list (backwards) applying f producing a single result”

The result is $f(a_1\ f(a_2\ ...(f\ a_n\ z))...))$

(sadly, not tail recursive)

Let's generalize!

```
sum [] = 0

sum (x:xs) = x + sum xs

toString [] = “”

toString (x:xs) =

    show x ++ toString xs

all _ [] = True

all p (x:xs) = p x && all p xs
```

```
sum [] = 0

sum (x:xs) = (+) x (sum xs)

toString [] = “”

toString (x:xs) =

    ((++).show) x (toString xs)

all _ [] = True

all p (x:xs) = ((&&).p) x (all p xs)
```

Let's generalize!

```
sum [] = 0

sum (x:xs) = (+) x (sum xs)

toString [] = “”

toString (x:xs) =

    ((++).show) x (toString xs)

all _ [] = True

all p (x:xs) = ((&&).p) x (all p xs)
```

A value z for the empty case

A function f for the head/tail case that combines the head with the result of the recursive call on the tail

What about other datatypes?

What about other datatypes?

What happens with other datatypes ?

```
data BTree a = Branch (BTree a) (BTree a) | Leaf a
```

(and what about Either or Maybe?)

How to sum all elements of a tree?

```
data BTree a = Branch (BTree a) (BTree a) | Leaf a

sum :: BTree Int -> Int
```

How to sum all elements of a tree?

```
data BTree a = Branch (BTree a) (BTree a) | Leaf a

sum :: BTree Int -> Int

sum (Leaf a) = ?

sum (Branch t1 t2) = ?
```

How to sum all elements of a tree?

```
data BTree a = Branch (BTree a) (BTree a) | Leaf a

sum :: BTree Int -> Int

sum (Leaf a) = a

sum (Branch t1 t2) = sum t1 + sum t2
```

Convert to string all elements of a tree?

```
data BTree a = Branch (BTree a) (BTree a) | Leaf a

toString :: BTree Int -> String

toString (Leaf a) = ?

toString (Branch t1 t2) = ?
```

Convert to string all elements of a tree?

```
data BTree a = Branch (BTree a) (BTree a) | Leaf a

toString :: BTree Int -> String

toString (Leaf a) = show a

toString (Branch t1 t2) = toString t1 ++ toString t2
```

How to fold a tree?

```
data BTree a = Branch (BTree a) (BTree a) | Leaf a

rec_tree :: (b -> b -> b) ->(a -> b) -> Tree a -> b

rec_tree _ g (Leaf a) = ?

rec_tree f g (Branch t1 t2) = ?
```

How to fold a tree?

```
data BTree a = Branch (BTree a) (BTree a) | Leaf a

rec_tree :: (b -> b -> b) ->(a -> b) -> Tree a -> b

rec_tree _ g (Leaf a) = g a

rec_tree f g (Branch t1 t2) =

             f (rec_tree f g t1)(rec_tree f g t2)
```

What is a fold?

What is a fold?

```
data List a = Cons a (List a) | Nil

                              (or data [] a = a : [a] | [] )

rec_list :: (a -> b -> b) -> b -> List a -> b



Cons 1 (Cons 2 (Cons 3 (Nil))) ~>

   f 1 (f 2 (f 3 z))
```

What is a fold?

```
data BTree a = Branch (BTree a) (BTree a) | Leaf a


rec_tree :: (b -> b -> b) ->(a -> b) -> BTree a -> b


Branch (Branch (Leaf 1) (Leaf 2)) (Leaf 3) ~>
        f (f (g 1) (g 2)) (g 3)
```

A fold replaces the datatype constructors with functions

What about other datatypes?

```
Maybe a = Nothing | Just a
```

```
Either a b = Left a | Right b
```

What about other datatypes?

```
Maybe a = Nothing | Just a

    fold_m :: b -> (a -> b) -> Maybe a -> b

    ( “maybe” in Haskell )

Either a b = Left a | Right b

    fold_e :: (a->c) -> (b->c) -> Either a b -> c

    ( “either” in Haskell )
```

What is a fold?

Transforms the input into something else, following the structure of the datatype

Catamorphism

Greek 'κατα-' meaning "downward or according to"

“There’s a truly marvellous category theory explanation for this which this slide is too narrow to contain”

Catamorphisms!

"Catamorphisms are generalizations of the concept of a fold in functional programming. A catamorphism deconstructs a data structure with an F-algebra for its underlying functor"

$$
\begin{tikzcd}
FT \arrow[r, "F\mathrm{fold}_F(h)"] \arrow[d, "\mathrm{in}_F"'] & FA \arrow[d, "h"] \\
T \arrow[r, "\mathrm{fold}_F(h)"] & A
\end{tikzcd}
$$

Given an F-algebra h : F A → A,

$\mathrm{fold}_F(h)$ is the unique homomorphism between in_F (initial algebra) and h

Fold replaces each constructor for the corresponding function in the algebra.

Fold corresponds to definitions by structural recursion over the type

Catamorphisms!

"Catamorphisms are generalizations of the concept of a fold in functional programming. A catamorphism deconstructs a data structure with an F-algebra for its underlying functor"

$$
\begin{tikzcd}
FT \arrow[r, "F\mathrm{fold}_F(h)"] \arrow[d, "\mathrm{in}_F"'] & FA \arrow[d, "h"] \\
T \arrow[r, "\mathrm{fold}_F(h)"] & A
\end{tikzcd}
$$

Given an F-algebra h : F A → A,

$\mathrm{fold}_F(h)$ is the unique homomorphism between in_F and h

Fold replaces each constructor for the corresponding function in the algebra.

Fold corresponds to definitions by structural recursion over the type

(An algebra of functors 1,K,I,+,* can describe regular datatypes and be an initial algebra for all of them)

"After all, a fold is originated by the unique homomorphism that exists between the initial algebra and any other algebra, what's the problem?"

Thank you!

@Gclaramunt

Bonus track

```
foldr (:) [] [1,2,3] == [1,2,3]
```

What about:

```
 foldr ((:).f) [] or fold ((Leaf).f)(Branch)
```

```
 What's the result of foldr ((:).(+2)) [] [1,2,3] ?
```

Bonus track

```
foldr (:) [] [1,2,3] == [1,2,3]
```

What about:

```
foldr ((:).f) [] or fold ((Leaf).f)(Branch)
```

What’s the result of foldr ((:).(+2)) [] [1,2,3] ?

That’s the map function!