# ALL ABOUT A FOLD\*

**GClaramunt** 

# YOU COULD'VE INVENTED FOLD...

### HOW TO SUM ALL ELEMENTS OF A LIST?

```
[1, 7, 4, 11, 3, 9]
sum :: [Int] -> Int
```

### HOW TO SUM ALL ELEMENTS OF A LIST?

```
[1, 7, 4, 11, 3, 9]
sum :: [Int] -> Int
sum [] = ?
sum (x:xs) = ?
```

### HOW TO SUM ALL ELEMENTS OF A LIST?

```
[1, 7, 4, 11, 3, 9]
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum 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)
```

### HOW TO CONCATENATE ALL ELEMENTS OF A LIST?

```
[1, 7, 4, 11, 3, 9]
toString :: [Int] -> String
toString [] = ?
toString (x:xs) = ?
```

### CONVERT TO STRING ALL ELEMENTS OF A LIST?

```
[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?

```
[1, 7, 4, 11, 3, 9]
all :: ( a->Bool ) -> [a] -> Bool
all _ [] = ?
all p (x:xs) = ?
```

### ALL ELEMENTS OF A LIST SATISFY A PROPERTY?

```
[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:

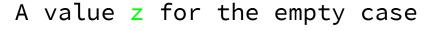
- One definition for the empty case
- One definition for the head/tail case

We are doing recursion in the structure of the list!

```
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 [] = 0
sum (x:xs) = x + sum xs
                                       sum (x:xs) = (+) x (sum xs)
toString [] = ""
                                       toString [] = ""
toString (x:xs) =
                                       toString (x:xs) =
                                           ((++).show) x (toString xs)
    show x ++ toString xs
all [] = True
                                       all [] = True
all p(x:xs) = p \times \&\&  all p(xs) = ((\&\&).p) \times (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)
```



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

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

```
rec_list f z [] = z
rec_list f z (x:xs) = f x (rec_list f z xs)
```

Is "foldr"!

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 rec\_list f z [] = z

rec\_list f z (x:xs) = f x (rec\_list f z xs)

### 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)

# WHAT ABOUT OTHER DATATYPES?

### FOLD!

```
What happens with other datatypes ?

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

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) = sum t1 ++ sum t2
```

### HOW TO FOLD A TREE?

```
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 )
```

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

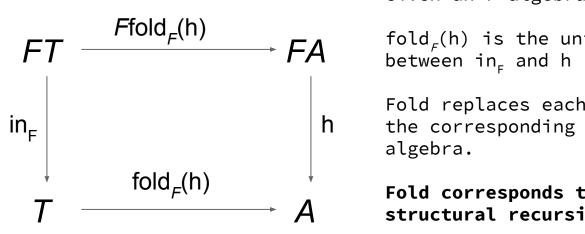
Catamorphism

Greek ' $\kappa\alpha\tau\alpha$ -' 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"



Given an F-algebra  $h : F A \rightarrow A$ ,

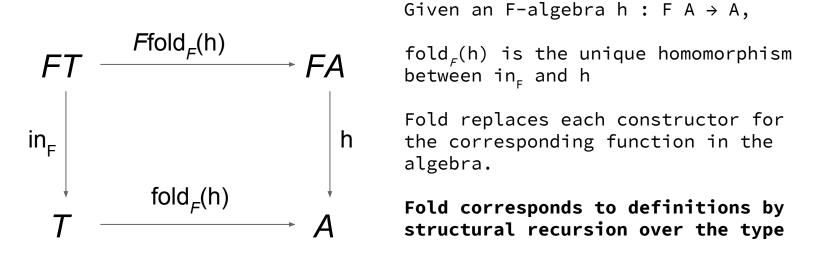
fold<sub>F</sub>(h) is the unique homomorphism

Fold replaces each constructor for the corresponding function in the

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"



(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