# Higher-order functions

## Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOG
Robertino Martinez, IOG
Karina Lopez, IOG

August, 2023

# Goals

- Abstracting standard design pattern on lists to `foldr`.
- Abstracting accumulating parameter pattern on lists to `foldl'`.
- The importance of strictness in `foldl'` (introduce `seq` and bang patterns).
- Generalizing `foldr` / `foldl'` to other data structures: `Foldable`.
- Generalizing `map` to other data structures: `Functor`.
- Generalizing `foldr` to catamorphisms

# Higher-order functions

# Functions, functions, functions

A function parameterized by another function or returning a function is called a *higher-order function*.

# Functions, functions, functions

A function parameterized by another function or returning a function is called a *higher-order function*.

Strictly speaking, every curried function in Haskell is a function returning another function:

```
elem   :: Eq a => a -> ([a] -> Bool)
elem 3 :: (Eq a, Num a) => [a] -> Bool
```

# Filtering and mapping

Two of the most useful list functions are higher-order, as they each take a function as an argument:

```
filter :: (a -> Bool) -> [a] -> [a]
map    :: (a -> b) -> [a] -> [b]
```

# Filtering and mapping

Two of the most useful list functions are higher-order, as they each take a function as an argument:

```
filter :: (a -> Bool) -> ([a] -> [a])
map    :: (a -> b) -> ([a] -> [b])
```

The use of a function `a -> Bool` to express a predicate is generally common. And mapping a function over a data structure is an operation that isn't limited to lists.

# Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: ...
(f . g) x = f (g x)
```

For once – rather than starting from a type – let's infer the type from the code.

# Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: ... -> ... -> ... -> ...
(f . g) x = f (g x)
```

It's apparently a curried function taking three arguments `f`, `g` and `x`.

# Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (... -> ...) -> (... -> ...) -> ... -> ...
(f . g) x = f (g x)
```

Both `f` and `g` are applied to something, so they must be functions.

# Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (... -> ...) -> (... -> ...) -> a -> ...
(f . g) x = f (g x)
```

No requirements seem to be made about the type of `x`, except that its passed to `g`, so let's assume a type variable here …

# Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (... -> ...) -> (a -> ...) -> a -> ...
(f . g) x = f (g x)
```

...which then should be the source type of `g` as well.

# Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (b -> ...) -> (a -> b) -> a -> ...
(f . g) x = f (g x)
```

The target type of `g` should match the source type of `f`.

# Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

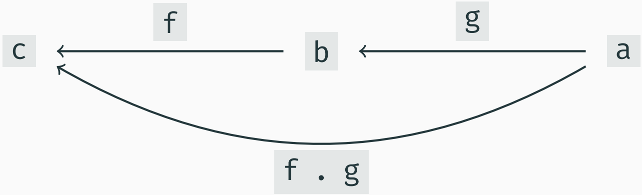
The target type of `f` is also the type of the overall result.

# Function composition

One of the most ubiquitous higher-order functions is function composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

Putting extra parentheses in the type may make it more obvious that we are indeed composing two matching functions.



# Composing functions

We can often build functions from existing functions simply by composing them.

## Composing functions

We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

```
example :: [Int]
example =
                                                  [1 ..]
```

We start by generating all numbers (lazy evaluation in action).

# Composing functions

We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

```
example :: [Int]
example =
                        map (\ x -> x * x)  [1 ..]
```

We use `map` to compute the square numbers. Note that `map` and `filter` are often used with *anonymous* functions.

# Composing functions

We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

```
example :: [Int]
example =
  (          filter odd . map (\ x -> x * x)) [1 ..]
```

We use function composition composition (and partial application) to subsequently filter the odd square numbers.

# Composing functions

We can often build functions from existing functions simply by composing them.

Example: Computing the first 100 odd square numbers.

```
example :: [Int]
example =
  (take 100 . filter odd . map (\ x -> x * x)) [1 ..]
```

Finally, we use composition again to take the first 100 elements of this list.

# Composition as a design pattern

- Function composition gives you a way to split one programming problem into several, possibly smaller, programming problems.
- In general, higher-order functions are part of your toolbox for attacking programming problems. Recognizing something as a `map` or `filter` is also useful.
- Of course, you should never forget the standard design principle of following the datatype structure as a good way of defining most functions, if applying a higher-order function fails.

# Lessons

- Function composition is a bit like the functional semicolon. It allows us to decompose larger tasks into smaller ones.
- Lazy evaluation allows us to separate the generation of possible results from selecting interesting results. This allows more modular programs in many situations.
- Partial application and anonymous functions help to keep such composition chains concise.

# Operating on functions

# Flipping a function

If you want to change the order of arguments of a two-argument curried function, you can use

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

# Flipping a function

If you want to change the order of arguments of a two-argument curried function, you can use

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Note once again that the function arrow associates to the right, so `flip` can really be seen as a function with three arguments:

```
f :: a -> b -> c
x :: b
y :: a
```

# Flipping a function

If you want to change the order of arguments of a two-argument curried function, you can use

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Example use:

```
foreach = flip map
example = foreach [1, 2, 3] (\ x -> x * x)
```

# Currying and uncurrying

Sometimes, you end up with a pair and want to apply a function to it that typically (in Haskell) is in curried form. Fortunately, we can convert between curried and uncurried form easily:

```haskell
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y

curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
```

# Currying and uncurrying

Sometimes, you end up with a pair and want to apply a function to it that typically (in Haskell) is in curried form. Fortunately, we can convert between curried and uncurried form easily:

```haskell
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y

curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
```

Example:

```haskell
map (uncurry (*)) (zip [1 .. 3] [4 .. 6])
```

# Capturing design patterns

# Abstraction

One of the strengths of Haskell's flexibility with functions is that they really allow to abstract from reoccuring patterns and thereby save code.

# Abstraction

One of the strengths of Haskell's flexibility with functions is that they really allow to abstract from reoccuring patterns and thereby save code.

The standard design principle for lists we've been using all the time works as follows:

```
fun :: [someType] -> someResult
fun []       = ...   -- code
fun (x : xs) = ...   -- code that can use x and fun xs
```

# From an informal pattern to a function

```
fun :: [someType] -> someResult
fun []       = ...   -- code
fun (x : xs) = ...   -- code that can use x and fun xs
```

We have two *interesting* positions where we have to fill in situation-specific code. Let's abstract!

# From an informal pattern to a function

```
fun :: [someType] -> someResult
fun []       = nil
fun (x : xs) = cons x (fun xs)
```

- We give names to the cases that correspond to the constructors.
- The case `cons` can use `x` and `fun xs`, so we turn it into a function.
- At the moment, this is not a valid function, because `nil` and `cons` come out of nowhere – but we can turn them into parameters of `fun`!

# From an informal pattern to a function

```
fun :: ... -> ... -> [someType] -> someResult
fun cons nil []       = nil
fun cons nil (x : xs) = cons x (fun cons nil xs)
```

We now have to look at the types of `cons` and `nil`:

- `nil` is used as a result, so `nil :: someResult`;
- `cons` takes a list element and a result to a result, so `cons :: someType -> someResult -> someResult`.

# From an informal pattern to a function

```
fun :: (someType -> someResult -> someResult)
    -> someResult
    -> [someType] -> someResult
fun cons nil []       = nil
fun cons nil (x : xs) = cons x (fun cons nil xs)
```

We can give shorter names to `someType` and `someResult`

…

# From an informal pattern to a function

```
fun :: (a -> r -> r) -> r -> [a] -> r
fun cons nil []       = nil
fun cons nil (x : xs) = cons x (fun cons nil xs)
```

This function is called `foldr` ...

# From an informal pattern to a function

```
foldr :: (a -> r -> r) -> r -> [a] -> r
foldr cons nil []       = nil
foldr cons nil (x : xs) = cons x (foldr cons nil xs)
```

We could equivalently define it using `where` ...

# From an informal pattern to a function

```haskell
foldr :: (a -> r -> r) -> r -> [a] -> r
foldr cons nil = go
  where
    go []       = nil
    go (x : xs) = cons x (go xs)
```

The arguments `cons` and `nil` never change while traversing the list, so we can just refer to them in the local definition `go`, without explicitly passing them around.

# Using `foldr`

```haskell
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

```haskell
foldr :: (a -> r -> r) -> r -> [a] -> r
foldr cons nil = go
  where
    go []       = nil
    go (x : xs) = cons x (go xs)
```

# Using `foldr`

```
length :: [a] -> Int
length []       = 0
length (x : xs) = 1 + length xs
```

```
foldr :: (a -> r -> r) -> r -> [a] -> r
foldr cons nil = go
  where
    go []       = nil
    go (x : xs) = cons x (go xs)
```

```
length = foldr (\ x r -> 1 + r) 0
```

```
length = foldr (const (1 +)) 0
```

# Examples of using `foldr`

```
(++) :: [a] -> [a] -> [a]
(++) xs ys = foldr (:) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p =
  foldr (\ x r -> if p x then x : r else r) []

map :: (a -> b) -> [a] -> [b]
map f = foldr (\ x r -> f x : r) []
```

# Examples of using `foldr`

```
(++) :: [a] -> [a] -> [a]
(++) xs ys = foldr (:) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p =
  foldr (\ x r -> if p x then x : r else r) []

map :: (a -> b) -> [a] -> [b]
map f = foldr (\ x r -> f x : r) []
```

```
and :: [Bool] -> Bool
and = foldr (&&) True

any :: (a -> Bool) -> [a] -> Bool
any p = foldr (\ x r -> p x || r) False
```

# The role of `foldr`

- When a list function is easy to express using `foldr`, then you should.
- Makes it immediately recognizable for the reader that it follows the standard design principle.
- Some functions can be expressed using `foldr`, but that does not necessarily make them any clearer. In such cases, aim for clarity.

# Accumulating parameter pattern

```haskell
reverse :: [a] -> [a]
reverse = go []
  where
    go :: [a] -> [a] -> [a]
    go acc []        = acc
    go acc (x : xs) = go (x : acc) xs
```

```haskell
sum :: Num a => [a] -> a
sum = go 0
  where
    go :: Num a => a -> [a] -> a
    go acc []        = acc
    go acc (x : xs) = go (x + acc) xs
```

What can we abstract from here?

# Accumulating parameter pattern

```haskell
reverse :: [a] -> [a]
reverse = go []
  where
    go :: [a] -> [a] -> [a]
    go acc []       = acc
    go acc (x : xs) = go (x : acc) xs
```

```haskell
sum :: Num a => [a] -> a
sum = go 0
  where
    go :: Num a => a -> [a] -> a
    go acc []       = acc
    go acc (x : xs) = go (x + acc) xs
```

What can we abstract from here?

# Abstracting

```
fun :: [a] -> r
fun = go ...
  where
    go acc []       = acc
    go acc (x : xs) = go (... acc ... x ...) xs
```

We apply the same tactics as before: let's abstract from the interesting positions and introduce names.

# Abstracting

```
fun :: [a] -> r
fun = go e
  where
    go acc []       = acc
    go acc (x : xs) = go (op acc x) xs
```

Now we need to introduce `e` and `op` as parameters.

# Abstracting

```
fun :: ... -> ... -> [a] -> r
fun op e = go e
  where
    go acc []       = acc
    go acc (x : xs) = go (op acc x) xs
```

And we have to figure out the types (or let the compiler infer them).

# Abstracting

```haskell
fun :: (r -> a -> r) -> r -> [a] -> r
fun op e = go e
  where
    go acc []       = acc
    go acc (x : xs) = go (op acc x) xs
```

This function is called `foldl`.

# Abstracting

```
foldl :: (r -> a -> r) -> r -> [a] -> r
foldl op e = go e
  where
    go acc []       = acc
    go acc (x : xs) = go (op acc x) xs
```

This function is called `foldl`.

# `foldr` and `foldl`

```
foldr (⊕) e [x, y, z] = x ⊕ (y ⊕ (z ⊕ e))
foldl (⊕) e [x, y, z] = ((e ⊕ x) ⊕ y) ⊕ z
```

# `foldr` and `foldl`

```
foldr (⊕) e [x, y, z] = x ⊕ (y ⊕ (z ⊕ e))
foldl (⊕) e [x, y, z] = ((e ⊕ x) ⊕ y) ⊕ z
```

```
foldr (+) 0 [1, 2, 3] = 1 + (2 + (3 + 0))
foldl (+) 0 [1, 2, 3] = ((0 + 1) + 2) + 3
```

# Enforcing strictness

- The accumulator is a *thunk*, an unevaluated expression, growing larger and larger (laziness!).
- Up til now, we have not seen a way to force evaluation.

# Enforcing strictness

- The accumulator is a *thunk*, an unevaluated expression, growing larger and larger (laziness!).
- Up til now, we have not seen a way to force evaluation.

**`seq`**

Haskell has a *primitive* `seq :: a -> b -> b`, which evaluates `a` if `seq a b` is evaluated and returns `b`.

# From `foldl` to `foldl'`

```haskell
foldl :: (r -> a -> r) -> r -> [a] -> r
foldl op e = go e
  where
    go acc []       = acc
    go acc (x : xs) = go (op acc x) xs
```

# From `foldl` to `foldl'`

```haskell
foldl :: (r -> a -> r) -> r -> [a] -> r
foldl op e = go e
  where
    go acc []         = acc
    go acc (x : xs) = let acc' = op acc x
                      in go acc' xs
```

# From `foldl` to `foldl'`

```haskell
foldl' :: (r -> a -> r) -> r -> [a] -> r
foldl' op e = go e
  where
    go acc []       = acc
    go acc (x : xs) = let acc' = op acc x
                      in acc' `seq` go acc' xs
```

# Bang patterns

Using the language extension *BangPatterns*, we can write this as follows:

```
foldl' :: (r -> a -> r) -> r -> [a] -> r
foldl' op e = go e
  where
    go acc []       = acc
    go acc (x : xs) = let !acc' = op acc x
                      in  go acc' xs
```

# Bang patterns

Using the language extension *BangPatterns*, we can write this as follows:

```
foldl' :: (r -> a -> r) -> r -> [a] -> r
foldl' op e = go e
  where
    go acc []       = acc
    go acc (x : xs) = let !acc' = op acc x
                      in  go acc' xs
```

- You should (almost) always use `foldl'` instead of `foldl`.
- The actual implementation of `foldl'` in `Data.List` is actually different and highly optimized.

# Beyond lists

# Generic concepts

Some of the concepts we have seen are not specific to lists:

- the function `foldr` replaces data constructors by suitable functions and follows the structure of the datatype, just like the standard design principle;
- the function `elem` traverses a data structure and checks whether it contains a particular element;
- the function `filter` traverses a data structure and produces a substructure containing just the elements with a certain property;
- the function `map` traverses a data structure and produces a new structure of the same shape, but with modified elements.

For some of these concepts, Haskell therefore offers more type classes.

# Foldable

Data structures that can be viewed as a list:

```
class Foldable t where
  foldr   :: (a -> b -> b) -> b -> t a -> b
  foldl'  :: (b -> a -> b) -> b -> t a -> b
  toList  :: t a -> [a]
  null    :: t a -> Bool
  length  :: t a -> Int
  elem    :: Eq a => a -> t a -> Bool
  maximum :: Ord a => t a -> a
  product :: Num a => t a -> a
  ...
```

Some of these are only available via `Data.Foldable`.

Note that `Foldable` abstracts over a parameterized type `t`.

# Other foldable types

The `Maybe` type is a container with 0 or 1 elements:

```
GHCi> null (Just 3)
False
GHCi> null Nothing
True
GHCi> product Nothing
1
```

# Possible pitfall: foldable pairs

A pair is a container containing exactly 1 element (its second component). (Tagged value.)

```
GHCi> toList (3, 4)
[4]
GHCi> toList ("foo", True)
[True]
GHCi> sum (3, 4)
4
```

# Possible pitfall: foldable `Either`

An `Either` is like `Maybe` where `Nothing` is replaced by `Left`. So `Right` injects an element, `Left` does not.

```
GHCi> length (Right 3)
1
GHCi> length (Left 3)
0
```

# Mapping over other types

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x)   = Leaf (f x)
mapTree f (Node l r) =
  Node (mapTree f l) (mapTree f r)
```

# Mapping over other types

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x)   = Leaf (f x)
mapTree f (Node l r) =
  Node (mapTree f l) (mapTree f r)
```

```haskell
data Maybe a = Nothing | Just a

mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing  = Nothing
mapMaybe f (Just x) = Just (f x)
```

# The `Functor` class

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The class `Functor` also abstracts over a parameterized type.

# The `Functor` class

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The class `Functor` also abstracts over a parameterized type.

```
instance Functor [] where
  fmap = map

instance Functor Tree where
  fmap = mapTree

instance Functor Maybe where
  fmap = mapMaybe
```

# The `Functor` class

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The class `Functor` also abstracts over a parameterized type.

```haskell
instance Functor [] where
  fmap = map

instance Functor Tree where
  fmap = mapTree

instance Functor Maybe where
  fmap = mapMaybe
```

```haskell
(<$>) :: Functor f => (a -> b) -> f a -> f b
f <$> x = fmap f x   -- just a different name
```

# `Functor` laws

Instances of the `Functor` class should obey the following *laws*:

# `Functor` laws

Instances of the `Functor` class should obey the following *laws*:

- `fmap id == id`
- `fmap (f . g) == fmap f . fmap g`

# `Functor` laws

Instances of the `Functor` class should obey the following *laws*:

- `fmap id == id`
- `fmap (f . g) == fmap f . fmap g`

The Haskell type system is not strong enough to actually *enforce* these laws, but you should nevertheless avoid writing an instance where the laws do not hold. Such an instance would confuse users of your instance.

# `Functor` laws

Instances of the `Functor` class should obey the following *laws*:

- `fmap id == id`
- `fmap (f . g) == fmap f . fmap g`

$$\begin{tikzcd} \texttt{c} & \texttt{b} \arrow[l, "\texttt{f}"] & \texttt{a} \arrow[l, "\texttt{g}"] \arrow[ll, bend right, "\texttt{f . g}"'] \end{tikzcd}$$

$$\begin{tikzcd} \texttt{f c} & \texttt{f b} \arrow[l, "\texttt{fmap f}"'] & \texttt{f a} \arrow[l, "\texttt{fmap g}"'] \arrow[ll, bend left, "\texttt{fmap (f . g) == fmap f . fmap g}"] \end{tikzcd}$$

# Deriving `Functor` and `Foldable`

Class instances for `Functor` and `Foldable` (and a few other classes) can be derived via language extensions:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable #-}
```

Language pragmas have to appear at the top of the module.

# Deriving `Functor` and `Foldable`

Class instances for `Functor` and `Foldable` (and a few other classes) can be derived via language extensions:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable #-}
```

Language pragmas have to appear at the top of the module.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving (Show, Eq, Functor, Foldable)
```

# Deriving `Functor` and `Foldable`

Class instances for `Functor` and `Foldable` (and a few other classes) can be derived via language extensions:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable #-}
```

Language pragmas have to appear at the top of the module.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
  deriving (Show, Eq, Functor, Foldable)
```

```
GHCi> length (Node (Leaf 3) (Leaf 4))
2
GHCi> (+ 1) <$> Node (Leaf 3) (Leaf 4)
Node (Leaf 4) (Leaf 5)
```

# More Catamorphisms

# More catamorphisms – `Bool`

```haskell
fun :: Bool -> r
fun True  = ...   -- code
fun False = ...   -- code
```

We apply the same tactics as before: let's abstract from the interesting positions and introduce names.

# More catamorphisms – `Bool`

```
fun :: Bool -> r
fun True  = then'
fun False = else'
```

Now we need to introduce `then'` and `else'` as parameters.

# More catamorphisms – `Bool`

```
fun :: ... -> ... -> Bool -> r
fun then' else' = go
  where
    go True  = then'
    go False = else'
```

And we have to figure out the types (or let the compiler infer them).

# More catamorphisms – `Bool`

```haskell
fun :: r -> r -> Bool -> r
fun then' else' = go
  where
    go True  = then'
    go False = else'
```

# More catamorphisms – `Bool`

```haskell
ifThenElse :: r -> r -> Bool -> r
ifThenElse then' else' = go
  where
    go True  = then'
    go False = else'
```

This function is basically the built-in `if ... then ... else` construct.

# More catamorphisms – `Maybe`

```haskell
fun :: Maybe a -> r
fun Nothing  = ...   -- code
fun (Just x) = ...   -- code that can use x
```

We apply the same tactics as before: let's abstract from the interesting positions and introduce names.

# More catamorphisms – `Maybe`

```haskell
fun :: Maybe a -> r
fun Nothing  = nothing
fun (Just x) = just x
```

Now we need to introduce `nothing` and `just` as parameters.

# More catamorphisms – `Maybe`

```
fun :: ... -> ... -> Maybe a -> r
fun nothing just = go
  where
    go Nothing  = nothing
    go (Just x) = just x
```

And we have to figure out the types (or let the compiler infer them).

# More catamorphisms – `Maybe`

```
fun :: r -> (a -> r) -> Maybe a -> r
fun nothing just = go
  where
    go Nothing  = nothing
    go (Just x) = just x
```

This function is called `maybe` (from module `Data.Maybe`).

# More catamorphisms – `Maybe`

```haskell
maybe :: r -> (a -> r) -> Maybe a -> r
maybe nothing just = go
  where
    go Nothing  = nothing
    go (Just x) = just x
```

This function is called `maybe` (from module `Data.Maybe`).

# More catamorphisms – `Tree`

```
fun :: Tree a -> r
fun (Leaf x)   = ...
    -- code that can use x
fun (Node l r) = ... (... fun l ... fun r ...)
    -- code that can use l and r
```

We apply the same tactics as before: let's abstract from the interesting positions and introduce names.

# More catamorphisms – `Tree`

```haskell
fun :: Tree a -> r
fun (Leaf x)   = leaf x
fun (Node l r) = node (fun l) (fun r)
```

Now we need to introduce `leaf` and `node` as parameters.

# More catamorphisms – `Tree`

```
fun :: ... -> ... -> Tree a -> r
fun leaf node = go
  where
    go (Leaf x)   = leaf x
    go (Node l r) = node (go l) (go r)
```

And we have to figure out the types (or let the compiler infer them).

# More catamorphisms – `Tree`

```haskell
fun :: (a -> r) -> (r -> r -> r) -> Tree a -> r
fun leaf node = go
  where
    go (Leaf x)   = leaf x
    go (Node l r) = node (go l) (go r)
```

# Catamorphism

- By now, you are hopefully convinced that every (algebraic) datatype comes with its own “standard design pattern function” or *recursion scheme*.
- Functions defined this way are called *catamorphisms*.
- For the most common Haskell types, these higher order functions are often known under different name (`foldr`, `maybe`,...).

# Catamorphism

- By now, you are hopefully convinced that every (algebraic) datatype comes with its own “standard design pattern function” or *recursion scheme*.
- Functions defined this way are called *catamorphisms*.
- For the most common Haskell types, these higher order functions are often known under different name (`foldr`, `maybe`,...).
- It is actually possible (using *higher rank polymorphism*) to *define* algebraic datatypes by their recursion schemes, so in a sense, the type `[]` is completely determined by `foldr`, `Maybe` by `maybe` etc.