A tutorial on the universality and expressiveness of fold
by Graham Hutton

narrated by Anton Trunov & Jesús Domínguez

IMDEA Software Institute
Madrid, Spain

Papers We Love. Mad
Madrid, Spain

Anton Trunov

I'm a research programmer at IMDEA Software Institute

anton-trunov @Anton_A_Trunov anton-trunov

Jesús Domínguez

I'm a PhD student at IMDEA Software Institute

My interests include:

- Type Theory
- Logic
- Concurrency
- Semantics of programming languages

jesus-dominguez @jeshecdom jeshecdom

Why we love the paper

- Highly readable
- Starts with the basics
- Teaches how to calculate functions' *implementations*
- Good overview of the area

Can you see the pattern here?

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + (sum xs)
```

Can you see the pattern here?

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + (sum xs)

all :: [Bool] -> Bool
all [] = True
all (x:xs) = x && (all xs)
```

Can you see the pattern here?

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + (sum xs)
```

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

```
all :: [Bool] -> Bool
all [] = True
all (x:xs) = x && (all xs)
```

Can you see the pattern here?

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + (sum xs)
```

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

```
all :: [Bool] -> Bool
all [] = True
all (x:xs) = x && (all xs)
```

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

Can you see the pattern here?

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + (sum xs)
```

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

```haskell
all :: [Bool] -> Bool
all [] = True
all (x:xs) = x && (all xs)
```

```haskell
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

```haskell
rec :: [a] -> b
rec [] = v
rec (x:xs) = x `f` (rec xs)
```

More examples of the same pattern

```haskell
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = x `snoc` (reverse xs)
  where snoc x xs = xs ++ [x]
```

More examples of the same pattern

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = x `snoc` (reverse xs)
  where snoc x xs = xs ++ [x]

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = x ?: (filter p xs)
  where (?:) x xs = if p x then x : xs else xs
```

More examples of the same pattern

```haskell
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = x `snoc` (reverse xs)
  where snoc x xs = xs ++ [x]

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = x ?: (filter p xs)
  where (?:) x xs = if p x then x : xs else xs

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

Abstracting out the pattern

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = x `f` (foldr f v xs)
```

What does fold do?

$$\texttt{\textbf{foldr}}\ (\oslash)\ \mathtt{v}\ [\mathtt{x}_1,\ \mathtt{x}_2,\ \mathtt{x}_3,\ \ldots,\ \mathtt{x}_{n-1}\ \mathtt{x}_n]$$

$$=$$

$$\mathtt{x}_1 \oslash (\mathtt{x}_2 \oslash (\mathtt{x}_3 \oslash \ldots (\mathtt{x}_{n-1} \oslash (\mathtt{x}_n \oslash \mathtt{v}))\ \ldots\ ))$$

Compare with

$$\mathtt{x}_1 : (\mathtt{x}_2 : (\mathtt{x}_3 : \ldots (\mathtt{x}_{n-1} : (\mathtt{x}_n : [\,]))\ \ldots\ ))$$

What does fold do?

$$\texttt{foldr}\ (\oslash)\ \mathtt{v}\ [\mathtt{x}_1,\ \mathtt{x}_2,\ \mathtt{x}_3,\ \ldots,\ \mathtt{x}_{n-1}\ \mathtt{x}_n]$$
$$=$$
$$\mathtt{x}_1 \oslash (\mathtt{x}_2 \oslash (\mathtt{x}_3 \oslash \ldots (\mathtt{x}_{n-1} \oslash (\mathtt{x}_n \oslash \mathtt{v}))\ \ldots))$$

Compare with

$$\mathtt{x}_1 : (\mathtt{x}_2 : (\mathtt{x}_3 : \ldots (\mathtt{x}_{n-1} : (\mathtt{x}_n : []))\ \ldots))$$

- `foldr` “replaces” `(:)` with $\oslash$
- and `[]` with `v`

Let's reimplement the functions we already saw

```
sum = foldr (+) 0
```

$x_1 + (x_2 + (x_3 + \ldots (x_{n-1} + (x_n + 0)) \ldots))$

Let's reimplement the functions we already saw

```
sum = foldr (+) 0
```

x_1 + (x_2 + (x_3 + ... (x_{n-1} + (x_n + 0)) ...))

```
all = foldr (&&) True
```

x_1 && (x_2 && (x_3 && ... (x_{n-1} && (x_n && True)) ...))

Let's reimplement the functions we already saw

```
sum = foldr (+) 0
```
x_1 + (x_2 + (x_3 + ... (x_{n-1} + (x_n + 0)) ...))

```
all = foldr (&&) True
```
x_1 && (x_2 && (x_3 && ... (x_{n-1} && (x_n && True)) ...))

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

Let's reimplement the functions we already saw

```
sum = foldr (+) 0
```
x$_1$ + (x$_2$ + (x$_3$ + ... (x$_{n-1}$ + (x$_n$ + 0)) ...))

```
all = foldr (&&) True
```
x$_1$ && (x$_2$ && (x$_3$ && ... (x$_{n-1}$ && (x$_n$ && True)) ...))

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

```
map f = foldr (\x r -> f x : r) []
```
f x$_1$: (f x$_2$: (f x$_3$: ... (f x$_{n-1}$: (f x$_n$: [])) ...))

Let's reimplement the functions we've already seen

```
reverse = foldr snoc []
  where snoc x xs = xs ++ [x]
```

x_1 `` `snoc` `` (x_2 `` `snoc` `` ... (x_{n-1} `` `snoc` `` (x_n `` `snoc` `` [])) ...)

Let's reimplement the functions we've already seen

```
reverse = foldr snoc []
  where snoc x xs = xs ++ [x]
```
x_1 `` `snoc` `` (x_2 `` `snoc` `` ... (x_{n-1} `` `snoc` `` (x_n `` `snoc` `` [])) ...)

```
filter p (x:xs) = foldr (?:) []
  where (?:) x xs = if p x then x : xs else xs
```
x_1 ?: (x_2 ?: (x_3 ?: ... (x_{n-1} ?: (x_n ?: [])) ...))

Let's reimplement the functions we've already seen

```
reverse = foldr snoc []
  where snoc x xs = xs ++ [x]
```
x₁ \`snoc\` (x₂ \`snoc\` ... (xₙ₋₁ \`snoc\` (xₙ \`snoc\` [])) ...)

```
filter p (x:xs) = foldr (?:) []
  where (?:) x xs = if p x then x : xs else xs
```
x_1 ?: (x_2 ?: (x_3 ?: ... (x_{n-1} ?: (x_n ?: [])) ...))

```
(++) xs ys = foldr (:) ys xs
-- or, in the point-free style: (++) = flip (foldr (:))
```
x_1 : (x_2 : (x_3 : ... (x_{n-1} : (x_n : ys)) ...))

Advantages of using fold

- Less boilerplate – we don't repeat the recursion scheme
- Can be easier to understand – the essence of the algorithm can be seen clearer
- The code can be constructed in a systematic manner
- Easier code transformations
- Makes it easier to prove the properties of functions

fold is *not* a silver bullet

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x : xs

-- example:
dropWhile even [4,2,1,2,3,4,5] = [1,2,3,4,5]
```

fold is *not* a silver bullet

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x : xs

-- example:
dropWhile even [4,2,1,2,3,4,5] = [1,2,3,4,5]
```

The folding function `f` gets

- the head of the list `x`

fold is *not* a silver bullet

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x : xs

-- example:
dropWhile even [4,2,1,2,3,4,5] = [1,2,3,4,5]
```

The folding function `f` gets

- the head of the list `x`
- the result of calling `fold` on tail

fold is *not* a silver bullet

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x : xs

-- example:
dropWhile even [4,2,1,2,3,4,5] = [1,2,3,4,5]
```

The folding function `f` gets

- the head of the list `x`
- the result of calling `fold` on tail

$$\underbrace{x_1}_{\text{head}} \oslash \underbrace{(x_2 \oslash (x_3 \oslash \ldots(x_{n-1} \oslash (x_n \oslash v))\ldots))}_{\text{foldr f v } [x_2,\ \ldots,\ x_n]}$$

fold is *not* a silver bullet

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x : xs

-- example:
dropWhile even [4,2,1,2,3,4,5] = [1,2,3,4,5]
```

The folding function `f` gets

- the head of the list `x`
- the result of calling `fold` on tail

$$\underbrace{x_1}_{\text{head}} \oslash \underbrace{(x_2 \oslash (x_3 \oslash \ldots (x_{n-1} \oslash (x_n \oslash v))\ldots))}_{\text{foldr f v } [x_2,\ \ldots,\ x_n]}$$

- but not the tail itself!

fold is *not* a silver bullet

```haskell
tail :: [a] -> [a]
tail [] = []            -- Exception in Haskell
tail (_:xs) = xs
```

fold is *not* a silver bullet

```
tail :: [a] -> [a]
tail [] = []            -- Exception in Haskell
tail (_:xs) = xs

tail :: [a] -> [a]
tail = snd . foldr (\x (xs, acc) -> (x : xs, xs)) ([], [])
```

fold is *not* a silver bullet

```
tail :: [a] -> [a]
tail [] = []            -- Exception in Haskell
tail (_:xs) = xs
```

```
tail :: [a] -> [a]
tail = snd . foldr (\x (xs, acc) -> (x : xs, xs)) ([], [])
```

Example:

foldr (\x (xs, acc) -> (x : xs, xs)) ([], []) [x_1, x_2, x_3]

fold is *not* a silver bullet

```
tail :: [a] -> [a]
tail [] = []           -- Exception in Haskell
tail (_:xs) = xs
```

```
tail :: [a] -> [a]
tail = snd . foldr (\x (xs, acc) -> (x : xs, xs)) ([], [])
```

Example:

foldr (\x (xs, acc) -> (x : xs, xs)) ([], []) [x_1, x_2, x_3]

```
[] ==> ([], [])
```

fold is *not* a silver bullet

```
tail :: [a] -> [a]
tail [] = []            -- Exception in Haskell
tail (_:xs) = xs
```

```
tail :: [a] -> [a]
tail = snd . foldr (\x (xs, acc) -> (x : xs, xs)) ([], [])
```

Example:

foldr (\x (xs, acc) -> (x : xs, xs)) ([], []) [x_1, x_2, x_3]

[] ==> ([], [])

x_3 ==> ([x_3], [])

fold is *not* a silver bullet

```
tail :: [a] -> [a]
tail [] = []            -- Exception in Haskell
tail (_:xs) = xs
```

```
tail :: [a] -> [a]
tail = snd . foldr (\x (xs, acc) -> (x : xs, xs)) ([], [])
```

Example:

foldr (\x (xs, acc) -> (x : xs, xs)) ([], []) [x_1, x_2, x_3]

[] ==> ([], [])

x_3 ==> ([x_3], [])

x_2 ==> ([x_2, x_3], [x_3])

fold is *not* a silver bullet

```haskell
tail :: [a] -> [a]
tail [] = []            -- Exception in Haskell
tail (_:xs) = xs

tail :: [a] -> [a]
tail = snd . foldr (\x (xs, acc) -> (x : xs, xs)) ([], [])
```

Example:

foldr (\x (xs, acc) -> (x : xs, xs)) ([], []) [x_1, x_2, x_3]

[] ==> ([], [])

x_3 ==> ([x_3], [])

x_2 ==> ([x_2, x_3], [x_3])

x_1 ==> ([x_1, x_2, x_3], [x_2, x_3])

foldr is *not* a silver bullet

Let's get back to `dropWhile`:

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p = snd . foldr f ([], [])
  where
    f x (xs, rec) = (x : xs, if p x then rec else x : xs)
```

foldr is *not* a silver bullet

Let's get back to `dropWhile`:

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p = snd . foldr f ([], [])
  where
    f x (xs, rec) = (x : xs, if p x then rec else x : xs)
```

Compare to:

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x : xs
```

Programs written using `fold` can be less readable than programs written using explicit recursion.

Iterators vs recursors

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Iterators vs recursors

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

```
foldrRec :: (a -> [a] -> b -> b) -> b -> [a] -> b
foldrRec f v [] = v
foldrRec f v (x:xs) = f x xs (foldrRec f v xs)
```

`foldr` is called an *iterator* a.k.a *catamorphism*
`foldrRec` is called a *recursor* a.k.a *paramorphism*
`foldrRec` implements a pattern called *primitive recursion*

dropWhile again

```
foldrRec :: (a -> [a] -> b -> b) -> b -> [a] -> b
foldrRec f v [] = v
foldrRec f v (x:xs) = f x xs (foldrRec f v xs)
```

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p = foldrRec
                (\x xs rec -> if p x then rec else x : xs)
                []
```

foldrRec via foldr

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f = foldrRec (const . f)
```

foldrRec via foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f = foldrRec (const . f)
```

```
foldrRec :: (a -> [a] -> b -> b) -> b -> [a] -> b
foldrRec f v = snd . foldr g ([], v)
  where
    g x (xs, rec) = (x : xs, f x xs rec)
```

Definitions of foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

Definitions of foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

$$\texttt{foldl}\ (\oslash)\ \texttt{v}\ [\texttt{x}_1,\ \texttt{x}_2,\ \texttt{x}_3,\ \ldots,\ \texttt{x}_{n-1}\ \texttt{x}_n]$$

$$=$$

$$(\ldots(((\texttt{v} \oslash \texttt{x}_1) \oslash \texttt{x}_2) \oslash \texttt{x}_3)\ \ldots \oslash \texttt{x}_{n-1}) \oslash \texttt{x}_n$$

Definitions of foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

$$\texttt{foldl}\ (\oslash\!\!\!\!\bigcirc)\ \texttt{v}\ [\texttt{x}_1,\ \texttt{x}_2,\ \texttt{x}_3,\ \ldots,\ \texttt{x}_{n-1}\ \texttt{x}_n]$$

$$=$$

$$(\ldots(((\texttt{v} \mathbin{\circledbslash} \texttt{x}_1) \mathbin{\circledbslash} \texttt{x}_2) \mathbin{\circledbslash} \texttt{x}_3)\ \ldots \mathbin{\circledbslash} \texttt{x}_{n-1}) \mathbin{\circledbslash} \texttt{x}_n$$

$$\texttt{x}_n \oslash (\texttt{x}_{n-1} \oslash \ldots (\texttt{x}_3 \oslash (\texttt{x}_2 \oslash (\texttt{x}_1 \oslash \texttt{v})))\ \ldots\ )$$

Definitions of foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

foldl $(\oslash)$ v $[x_1, x_2, x_3, \ldots, x_{n-1}\ x_n]$

=

$(\ldots(((v \oslash x_1) \oslash x_2) \oslash x_3) \ldots \oslash x_{n-1}) \oslash x_n$

$x_n \oslash (x_{n-1} \oslash \ldots (x_3 \oslash (x_2 \oslash (x_1 \oslash v))) \ldots)$

=

(**foldr** (**flip** $(\oslash)$) v) . **reverse**

Definitions of foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

$$\texttt{foldl}\ (\obslash)\ \texttt{v}\ [\texttt{x}_1,\ \texttt{x}_2,\ \texttt{x}_3,\ \ldots,\ \texttt{x}_{n-1}\ \texttt{x}_n]$$

$$=$$

$$(\ldots(((\texttt{v} \obslash \texttt{x}_1) \obslash \texttt{x}_2) \obslash \texttt{x}_3)\ \ldots \obslash \texttt{x}_{n-1}) \obslash \texttt{x}_n$$

$$\texttt{x}_n \oslash (\texttt{x}_{n-1} \oslash\ \ldots\ (\texttt{x}_3 \oslash (\texttt{x}_2 \oslash (\texttt{x}_1 \oslash \texttt{v})))\ \ldots\ )$$

$$=$$

```
(foldr (flip (⦸)) v) . reverse
```

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f v = (foldr (flip f) v) . reverse
```

Definitions of foldl

$\mathtt{\textbf{foldl}}\ (\oslash)\ \mathtt{v}\ \mathtt{xs}$

$=$

$(\ldots(((\mathtt{v} \oslash \mathtt{x}_1) \oslash \mathtt{x}_2) \oslash \mathtt{x}_3)\ \ldots \oslash \mathtt{x}_{n-1}) \oslash \mathtt{x}_n$

Definitions of foldl

$\texttt{\textbf{foldl}}\ (\oslash)\ \texttt{v}\ \texttt{xs}$

$=$

$(\ldots(((\texttt{v} \oslash \texttt{x}_1) \oslash \texttt{x}_2) \oslash \texttt{x}_3)\ \ldots \oslash \texttt{x}_{n-1}) \oslash \texttt{x}_n$

$=$

$(\oslash\ \texttt{x}_n)\ .\ (\oslash\ \texttt{x}_{n-1})\ \ldots\ .\ (\oslash\ \texttt{x}_3)\ .\ (\oslash\ \texttt{x}_2)\ .\ (\oslash\ \texttt{x}_1)\ \$\ \texttt{v}$

Definitions of foldl

$$\texttt{foldl } (\oslash) \texttt{ v xs}$$
$$=$$
$$(\ldots(((\texttt{v} \oslash \texttt{x}_1) \oslash \texttt{x}_2) \oslash \texttt{x}_3) \ldots \oslash \texttt{x}_{n-1}) \oslash \texttt{x}_n$$

$$=$$
$$(\oslash\ \texttt{x}_n) \;.\; (\oslash\ \texttt{x}_{n-1}) \;\ldots\; .\; (\oslash\ \texttt{x}_3) \;.\; (\oslash\ \texttt{x}_2) \;.\; (\oslash\ \texttt{x}_1) \;\$\; \texttt{v}$$

$$=$$
$$\texttt{id} \;.\; (\oslash\ \texttt{x}_n) \;.\; (\oslash\ \texttt{x}_{n-1}) \;\ldots\; .\; (\oslash\ \texttt{x}_3) \;.\; (\oslash\ \texttt{x}_2) \;.\; (\oslash\ \texttt{x}_1) \;\$\; \texttt{v}$$

Definitions of foldl

$$\textbf{foldl}\ (\oslash)\ v\ xs$$
$$=$$
$$(\ldots(((v \oslash x_1) \oslash x_2) \oslash x_3) \ldots \oslash x_{n-1}) \oslash x_n$$
$$=$$
$$(\oslash\ x_n)\ .\ (\oslash\ x_{n-1})\ \ldots\ .\ (\oslash\ x_3)\ .\ (\oslash\ x_2)\ .\ (\oslash\ x_1)\ \$\ v$$
$$=$$
$$\textbf{id}\ .\ (\oslash\ x_n)\ .\ (\oslash\ x_{n-1})\ \ldots\ .\ (\oslash\ x_3)\ .\ (\oslash\ x_2)\ .\ (\oslash\ x_1)\ \$\ v$$
$$=$$
$$((\ldots((\textbf{id}\ .\ (\oslash\ x_n))\ .\ (\oslash\ x_{n-1}))\ \ldots\ .\ (\oslash\ x_3))\ .\ (\oslash\ x_2))\ .\ (\oslash\ x_1)\ \$\ v$$

Definitions of foldl

```
foldl (⊘) v xs
  =
(...(((v ⊘ x1) ⊘ x2) ⊘ x3) ... ⊘ xn-1) ⊘ xn

  =
(⊘ xn) . (⊘ xn-1) ... . (⊘ x3) . (⊘ x2) . (⊘ x1) $ v

  =
id . (⊘ xn) . (⊘ xn-1) ... . (⊘ x3) . (⊘ x2) . (⊘ x1) $ v

  =
((...((id . (⊘ xn)) . (⊘ xn-1)) ... .
    (⊘ x3)) . (⊘ x2)) . (⊘ x1) $ v

  =
foldr (\x rec -> rec . (⊘ x)) id xs v
```

Universal Property of fold

Theorem

Given $f : a \to b \to b$, $v : b$, *and* $g : [a] \to b$, *we have,*

$$\begin{aligned} g\ [\,] &= v \\ g\ (x : xs) &= f\ x\ (g\ xs) \end{aligned} \quad \iff \quad g = \mathit{fold}\ f\ v$$

Universal Property of fold

How to prove this?

$$(+1)\ .\ \mathit{sum} = \mathit{fold}\ (+)\ 1$$

Universal Property of fold

How to prove this?

$$(+1)\ .\ \mathit{sum} = \mathit{fold}\ (+)\ 1$$

By universal property:

$$((+1)\ .\ \mathit{sum})\ [\,] = 1$$
$$((+1)\ .\ \mathit{sum})\ (x : xs) = (+)\ x\ (((+1)\ .\ \mathit{sum})\ xs)$$

$$g\ [\,] = v$$
$$g\ (x : xs) = f\ x\ (g\ xs)$$

$$\mathit{sum}\ [\,] + 1 = 1$$
$$\mathit{sum}\ (x : xs) + 1 = x + (\mathit{sum}\ xs + 1)$$

Universal Property of fold: List objects

$$1 \xrightarrow{nil} L(A) \qquad A \times L(A) \xrightarrow{cons} L(A)$$

$$1 \xrightarrow{v} B \qquad A \times B \xrightarrow{f} B$$

$$L(A) \overset{\exists!\ fold\ f\ v}{\dashrightarrow} B$$

$$\begin{tikzcd}
1 \arrow[r, "nil"] \arrow[dr, "v"'] & L(A) \arrow[d, "fold\ f\ v"] \\
 & B
\end{tikzcd}$$

$$\begin{tikzcd}
A \times L(A) \arrow[r, "cons"] \arrow[d, "id_A \times (fold\ f\ v)"'] & L(A) \arrow[d, "fold\ f\ v"] \\
A \times B \arrow[r, "f"'] & B
\end{tikzcd}$$

$$(fold\ f\ v) \circ nil = v$$

$$(fold\ f\ v) \circ cons = f \circ (id_A \times fold\ f\ v)$$

$$(fold\ f\ v)\ (cons\ (x, l)) =$$

$$f\ ((id_A \times fold\ f\ v)(x, l)) =$$

$$f\ (x, fold\ f\ v\ l)$$

Universal Property of fold: Initial Algebras

$$F_A\ X :\equiv 1 + A \times X$$
$$map\ F_A\ f :\equiv id_1 + id_A \times f$$

$$\begin{tikzcd} X \arrow[d, "f"'] \arrow[r, phantom, "\overset{map\ F_A\ f}{\longmapsto}"] & 1 + A \times X \arrow[d, "id_1 + id_A \times f"] \\ Y & 1 + A \times Y \end{tikzcd}$$

$$F_A\ L(A) \xrightarrow{In} L(A)$$
$$F_A\ B \xrightarrow{f} B$$
$$L(A) \overset{\exists!\ cata\ f}{\dashrightarrow} B$$

$$\begin{tikzcd} F_A\ L(A) \arrow[r, "In"] \arrow[d, "map\ F_A\ (cata\ f)"'] & L(A) \arrow[d, "cata\ f"] \\ F_A\ B \arrow[r, "f"'] & B \end{tikzcd}$$

$$\begin{aligned} (cata\ f) \circ In &= f \circ (map\ F_A\ (cata\ f)) \\ &= f \circ (id_1 + id_A \times (cata\ f)) \end{aligned}$$

Fold and other datatypes

$$\begin{aligned} id\ [\,] &= [\,] \\ id\ (x : xs) &= (:)\ x\ (id\ xs) \end{aligned} \iff id = fold\ (:)\ [\,]$$

Fold and other datatypes

$$\begin{aligned} id\ [\,] &= [\,] \\ id\ (x : xs) &= (:)\ x\ (id\ xs) \end{aligned} \iff id = fold\ (:)\ [\,]$$

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Fold (catamorphism) and trees

```
data LTree a =
   Leaf a | Split (LTree a) (LTree a)
data BTree a =
   Empty | Node a (BTree a) (BTree a)
```

Fold (catamorphism) and trees

```
data LTree a =
   Leaf a | Split (LTree a) (LTree a)
data BTree a =
   Empty | Node a (BTree a) (BTree a)

cata_LTree :: ....... -> LTree a -> b
cata_BTree :: ....... -> BTree a -> b
```

Fold (catamorphism) and trees

```
data LTree a =
   Leaf a | Split (LTree a) (LTree a)
data BTree a =
   Empty | Node a (BTree a) (BTree a)

cata_LTree :: ....... -> LTree a -> b
cata_BTree :: ....... -> BTree a -> b

cata_LTree :: (b -> b -> b) -> (a -> b) -> LTree a -> b
cata_BTree :: (a -> b -> b -> b) -> b -> BTree a -> b
```

Since:

```
Split :: LTree a -> LTree a -> LTree a
Leaf :: a -> LTree a
Node :: a -> BTree a -> BTree a -> BTree a
Empty :: BTree a
```

Fold (catamorphism) and trees

```
cata_LTree :: (b -> b -> b) -> (a -> b) -> LTree a -> b
cata_LTree fs fl (Leaf x) = fl x
cata_LTree fs fl (Split l r) =
  fs (cata_LTree fs fl l) (cata_LTree fs fl r)

cata_BTree :: (a -> b -> b -> b) -> b -> BTree a -> b
cata_BTree fn v Empty = v
cata_BTree fn v (Node x l r) =
  fn x (cata_BTree fn v l) (cata_BTree fn v r)
```

Compiler optimizations: rewrite rules

```
{-# RULES
  "map" [~1] forall f xs.
             map f xs =
             build (\c n -> foldr (mapFB c f) n xs)
-}
```

where

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []

mapFB :: (e -> lst -> lst) -> (a -> e) -> a -> lst -> lst
mapFB c f = \x ys -> c (f x) ys
```

The equations guiding the compiler must be correct!

Advantage: Reasoning made easier!

Interactive session in Coq: the fusion property of `fold`.

Outro

Thank you!

Get in touch with us after the talk!

github.com/anton-trunov/fold-tutorial-talk

Questions?