Folds in Haskell

Mark P Jones
Portland State University

Folds!

- A list xs can be built by applying the (:) and [] operators to a sequence of values:

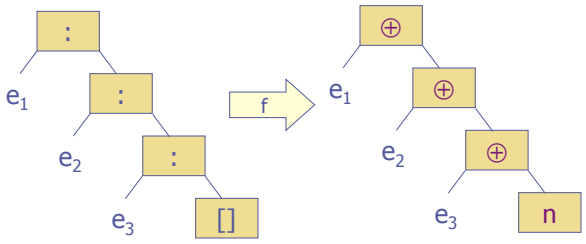
  $xs = x_1 : x_2 : x_3 : x_4 : \ldots : x_k : []$

- Suppose that we are able to replace every use of (:) with a binary operator (⊕), and the final [] with a value n:

  $xs = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus \ldots \oplus x_k \oplus n$

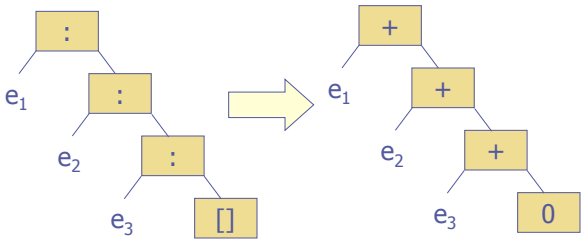
- The resulting value is called fold (⊕) n xs
- Many useful functions on lists can be described in this way.

Graphically:



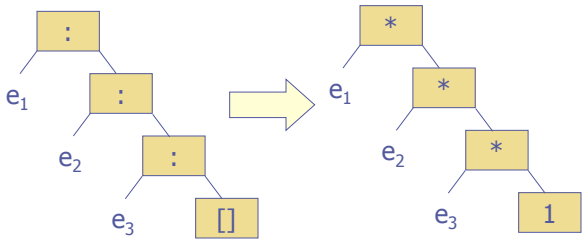
f = foldr (⊕) n

Example: sum



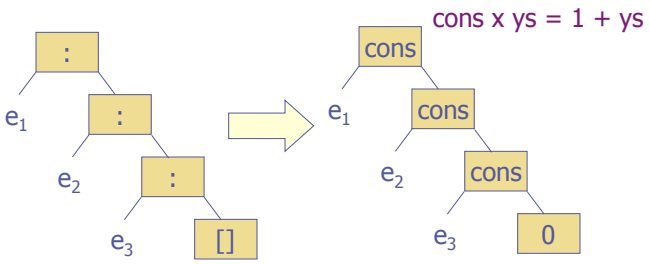
sum = foldr (+) 0

Example: product



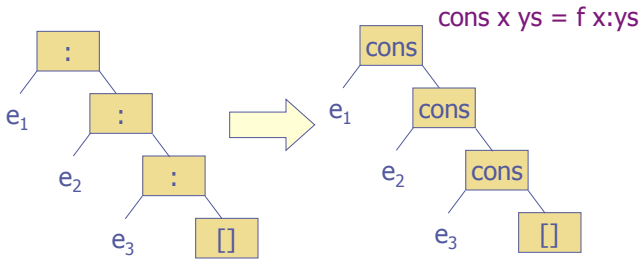
product = foldr (*) 1

Example: length



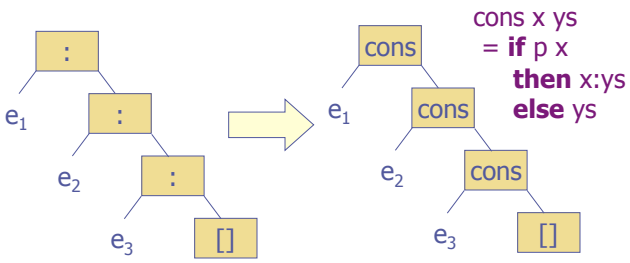
length = foldr (\x ys -> 1 + ys) 0

Example: map



map f = foldr (\x ys -> f x : ys) []

Example: filter



filter p = foldr (\x ys -> **if** p x **then** x:ys **else** ys) []

Formal Definition:

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


Applications:

```
sum          = foldr (+) 0
product      = foldr (*) 1
length       = foldr (\x ys -> 1 + ys) 0
map f        = foldr (\x ys -> f x : ys) []
filter p     = foldr c []
  where c x ys = if p x then x:ys else ys
xs ++ ys     = foldr (:) ys xs
concat       = foldr (++) []
and          = foldr (&&) True
or           = foldr (||)  False
```


Patterns of Computation:

- foldr captures a common pattern of computations over lists
- As such, it's a very useful function in practice to include in the Prelude
- Even from a theoretical perspective, it's very useful because it makes a deep connection between functions that might otherwise seem very different …
- From the perspective of lawful programming, one law about foldr can be used to reason about many other functions

A law about foldr:

- If (⊕) is an associative operator with unit n, then
  foldr (⊕) n xs ⊕ foldr (⊕) n ys
  = foldr (⊕) n (xs ++ ys)
- $(x_1 \oplus \ldots \oplus x_k \oplus n) \oplus (y_1 \oplus \ldots \oplus y_j \oplus n)$
  $= (x_1 \oplus \ldots \oplus x_k \oplus y_1 \oplus \ldots \oplus y_j \oplus n)$
- All of the following laws are special cases:

```
sum xs     +  sum ys      = sum (xs ++ ys)
product xs *  product ys  = product (xs ++ ys)
concat xss ++ concat yss  = concat (xss ++ yss)
and xs     && and ys      = and (xs ++ ys)
or xs      || or  ys      = or (xs ++ ys)
```

foldl:

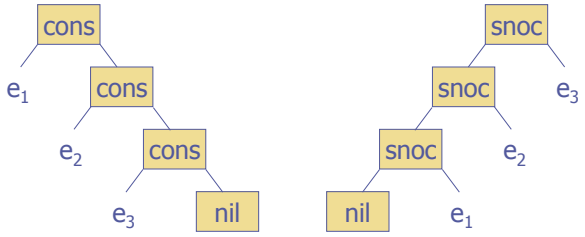
- There is a companion function to foldr called foldl:

```
foldl           :: (b -> a -> b) -> b -> [a] -> b
foldl s n []    = n
foldl s n (x:xs) = foldl s (s n x) xs
```

- For example:

$$\begin{aligned} &\text{foldl s n } [e_1, e_2, e_3] \\ &\quad = s\ (s\ (s\ n\ e_1)\ e_2)\ e_3 \\ &\quad = ((n\ \text{`s`}\ e_1)\ \text{`s`}\ e_2)\ \text{`s`}\ e_3 \end{aligned}$$

foldr vs foldl:



foldr foldl

Uses for foldl:

- Many of the functions defined using foldr can be defined using foldl:

```
sum     = foldl (+) 0
product = foldl (*) 1
```

- There are also some functions that are more easily defined using foldl:

```
reverse = foldl (\ys x -> x:ys) []
```

- When should you use foldr and when should you use foldl? When should you use explicit recursion instead?

foldr1 and foldl1:

- Variants of foldr and foldl that work on non-empty lists:

```
foldr1          :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)

foldl1          :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
```

- Notice:
  - No case for empty list
  - No argument to replace empty list
  - Less general type (only one type variable)

Uses of foldl1, foldr1:

From the prelude:

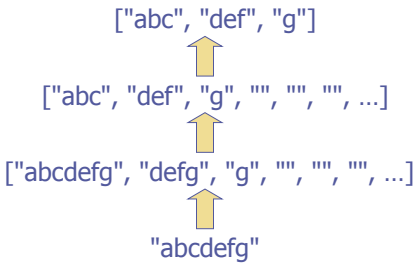
```
minimum = foldl1 min
maximum = foldl1 max
```

Not in the prelude:

```
commaSep = foldr1 (\s t -> s ++ ", " ++ t)
```

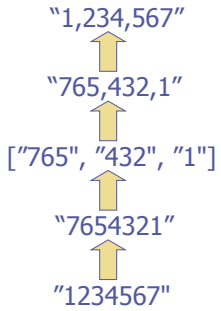

Example: Grouping

```
group n = takeWhile (not.null)
        . map (take n)
        . iterate (drop n)
```



Example: Adding Commas

```
group n  = reverse
         . foldr1 (\xs ys -> xs++","++ys)
         . group 3
         . reverse
```



Example: transpose

```
transpose              :: [[a]] -> [[a]]
transpose []           = []
transpose ([] : xss)   = transpose xss
transpose ((x:xs) : xss)
                = (x : [h | (h:t) <- xss])
                    : transpose (xs : [ t | (h:t) <- xss])
```

Example:

transpose [[1,2,3],[4,5,6]] = [[1,4],[2,5],[3,6]]

Example: say

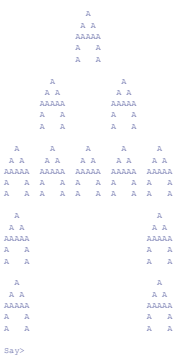
```
Say> putStr (say "hello")

H    H   EEEEE   L       L        OOO
H    H   E       L       L       O   O
HHHHH   EEEEE   L       L       O   O
H    H   E       L       L       O   O
H    H   EEEEE   LLLLL   LLLLL    OOO

Say>
```


… continued:

```
say   = ('\n':)
      . unlines
      . map (foldr1 (\xs ys->xs++"  "++ys))
      . transpose
      . map picChar
where
 picChar 'A' = [  "  A  ",
                  " A A ",
                  "AAAAA",
                  "A   A",
                  "A   A" ]
etc…
```


Composition and Reuse:

```
Say> (putStr . concat . map say . lines . say) "A"
```



```
Say>
```


Summary:

- Folds on lists have many uses
- Folds capture a common pattern of computation on list values
- In fact, there are similar notions of fold functions on many other algebraic datatypes …)