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

- Suppose that we are able to replace every use of \((:)\) with a binary operator \((\oplus)\), 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 \( \text{fold} (\oplus) n xs \)

Many useful functions on lists can be described in this way.

Graphically:

\[ f = \text{foldr} (\oplus) n \]

Example: sum

\[ \text{sum} = \text{foldr} (+) 0 \]

Example: product

\[ \text{product} = \text{foldr} (*) 1 \]

Example: length

\[ \text{length} = \text{foldr} (\lambda x ys \rightarrow 1 + ys) 0 \]
Example: map

\[
\text{map } f = \text{foldr } (\lambda x \ ys \rightarrow f \ x : ys) \ []
\]

Example: filter

\[
\text{filter } p = \text{foldr } (\lambda x \ ys \rightarrow \text{if } p x \text{ then } x:ys \text{ else } ys) \ []
\]

Formal Definition:

\[
\text{foldr } (::) :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]

\[
\text{foldr cons nil } [] = \text{nil}
\]

\[
\text{foldr cons nil } (x:xs) = \text{cons } x \ (\text{foldr cons nil xs})
\]

Applications:

\[
\text{sum } = \text{foldr } (+) \ 0
\]

\[
\text{product } = \text{foldr } (*) \ 1
\]

\[
\text{length } = \text{foldr } (\lambda x \ ys \rightarrow 1 + ys) \ 0
\]

\[
\text{map } f = \text{foldr } (\lambda x \ ys \rightarrow f \ x : ys) \ []
\]

\[
\text{filter } p = \text{foldr } (\lambda x \ ys \rightarrow \text{if } p x \text{ then } x:ys \text{ else } ys) \ []
\]

\[
\text{where } c \times ys = \text{if } p x \text{ then } x:ys \text{ else } ys
\]

\[
\text{xs }++\text{ ys } = \text{foldr } (++) \text{ xs } \text{ys}
\]

\[
\text{concat } xss ++ \text{ yss } = \text{concat } (xss ++ yss)
\]

\[
\text{and } \text{xs } \&\& \text{ and } \text{ys } = \text{and } (\text{xs }++\text{ ys})
\]

\[
\text{or } \text{xs } || \text{ or } \text{ys } = \text{or } (\text{xs }++\text{ ys})
\]

Patterns of Computation:

- \text{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 \text{foldr} can be used to reason about many other functions.

A law about \text{foldr}:

- If \((\oplus)\) is an associative operator with unit \(n\), then

\[
\text{foldr } (\oplus) \ n \ xs \oplus \text{foldr } (\oplus) \ n \ ys
\]

\[
= \text{foldr } (\oplus) \ n \ (xs ++ ys)
\]

\[
(x_1 \oplus \ldots \oplus x_k \oplus n) \oplus (y_1 \oplus \ldots \oplus y_j \oplus n)
\]

\[
= (x_1 \oplus \ldots \oplus x_k \oplus n) \oplus (y_1 \oplus \ldots \oplus y_j \oplus n)
\]

- All of the following laws are special cases:

\[
\text{sum } xs + \text{ sum } ys = \text{sum } (xs ++ ys)
\]

\[
\text{product } xs * \text{ product } ys = \text{product } (xs ++ ys)
\]

\[
\text{concat } xss ++ \text{ concat } yss = \text{concat } (xss ++ yss)
\]

\[
\text{and } xs \&\& \text{ and } ys = \text{and } (xs ++ ys)
\]

\[
\text{or } xs || \text{ or } ys = \text{or } (xs ++ ys)
\]
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:
  foldl s n [e_1, e_2, e_3] = s (s (s n e_1) e_2) e_3
  = ((n `s` e_1) `s` e_2) `s` e_3

foldr vs 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

```haskell
  group n = takeWhile (not.null) . map (take n) . iterate (drop n)
  ["abc", "def", "g"]
  ["abc", "def", "g", ",", ",", ",", ...]
  ["abcdefg", "defg", "g", ",", ",", ",", ...]
  "abcdefg"
```
Example: Adding Commas

\[
\text{group } n = \text{reverse} \\
\quad . \ \text{foldr1} (\langle x y s \rightarrow x s++","y s \rangle) \\
\quad . \ \text{group } 3 \\
\quad . \ \text{reverse} \\
\quad "1,234,567" \\
\quad "765,432,1" \\
\quad ["765", "432", "1"] \\
\quad "7654321" \\
\quad "1234567"
\]

Example: transpose

\[
\text{transpose} :: [[a]] \rightarrow [[a]] \\
\text{transpose } [] = [] \\
\text{transpose } ([] : xss) = \text{transpose } xss \\
\text{transpose } ([x : xs] : xss) \\
\quad = (x : [h | (h : t) \leftarrow xss]) \\
\quad : \text{transpose } (xs : [t | (h : t) \leftarrow xss])
\]

Example:

\[
\text{transpose } [[1,2,3],[4,5,6]] = [[1,4],[2,5],[3,6]]
\]

Example: say

\[
\text{Say}> \ \text{putStr} \ (\text{say} \ "\text{hello}\")
\]

\[
\begin{array}{cccc}
H & H & EEEE & L & L & 000 \\
H & H & E & L & L & 0 & 0 \\
HHHH & EEEE & L & L & 0 & 0 \\
H & H & EEEE & LLLLL & LLLLL & 000 \\
\end{array}
\]

\[
\text{Say}>
\]

... continued:

\[
\text{say} = (\'\n\':) \\
\quad \text{. unlines} \\
\quad \text{. map } (\text{foldr1 } (\langle x y s \rightarrow x s++","y s \rangle)) \\
\quad \text{. transpose} \\
\quad \text{. map } \text{picChar}
\]

where

\[
\text{picChar } \text{ 'A' } = [ \ " A \", \ " A A \", \ "AAAAA\", \ "A A", \ "A A" ]
\]

edc...

Composition and Reuse:

\[
\text{Composition and Reuse:}
\]

\[
\text{Composition and Reuse:}
\]

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 ...