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 ($\oplus$), 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 fold ($\oplus$) $n$ $xs$

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

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

\[ e_1 + e_2 + e_3 = \text{foldr} (+) 0 \]
Example: product

\[
\text{product} = \text{foldr} (\ast) 1
\]
Example: length

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

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

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

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

\[
\begin{align*}
\text{sum} & \quad = \, \text{foldr} \ (+) \ 0 \\
\text{product} & \quad = \, \text{foldr} \ (*) \ 1 \\
\text{length} & \quad = \, \text{foldr} \ (\lambda x \ ys \to 1 + ys) \ 0 \\
\text{map} \ f & \quad = \, \text{foldr} \ (\lambda x \ ys \to f \ x : ys) \ [] \\
\text{filter} \ p & \quad = \, \text{foldr} \ c \ [] \\
& \quad \text{where} \ c \ x \ ys = \, \text{if} \ p \ x \ \text{then} \ x : ys \ \text{else} \ ys \\
\text{xs ++ ys} & \quad = \, \text{foldr} \ (:) \ ys \ xs \\
\text{concat} & \quad = \, \text{foldr} \ (++) \ [] \\
\text{and} & \quad = \, \text{foldr} \ (\&\&) \ True \\
\text{or} & \quad = \, \text{foldr} \ (||) \ False
\end{align*}
\]
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 $(\oplus)$ is an associative operator with unit $n$, then
  \[
  \text{foldr} (\oplus) n \; \text{xs} \oplus \text{foldr} (\oplus) n \; \text{ys} = \text{foldr} (\oplus) n \; (\text{xs} ++ \text{ys})
  \]

- $(x_1 \oplus \ldots \oplus x_k \oplus n) \oplus (y_1 \oplus \ldots \oplus y_j \oplus n) = (x_1 \oplus \ldots \oplus x_k \oplus y_1 \oplus \ldots \oplus y_j \oplus n)$

- All of the following laws are special cases:
  - $\text{sum} \; \text{xs} + \; \text{sum} \; \text{ys} = \text{sum} \; (\text{xs} ++ \text{ys})$
  - $\text{product} \; \text{xs} \ast \; \text{product} \; \text{ys} = \text{product} \; (\text{xs} ++ \text{ys})$
  - $\text{concat} \; \text{xss} ++ \; \text{concat} \; \text{yss} = \text{concat} \; (\text{xss} ++ \text{yss})$
  - $\text{and} \; \text{xs} && \; \text{and} \; \text{ys} = \text{and} \; (\text{xss} ++ \text{yss})$
  - $\text{or} \; \text{xs} || \; \text{or} \; \text{ys} = \text{or} \; (\text{xss} ++ \text{yss})$
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 [e₁, e₂, e₃]
= s (s (s n e₁) e₂) e₃
= ((n `s` e₁)`s` e₂)`s` e₃
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 \texttt{foldr} and \texttt{foldl} that work on non-empty lists:

\begin{align*}
\text{foldr1} & \quad \text{:: (a -> a -> a) -> [a] -> a} \\
\text{foldr1} f \ [x] & = x \\
\text{foldr1} f \ (x:xs) & = f \ x \ (\text{foldr1} f \ xs)
\end{align*}

\begin{align*}
\text{foldl1} & \quad \text{:: (a -> a -> a) -> [a] -> a} \\
\text{foldl1} f \ (x:xs) & = \text{foldl} f \ x \ xs
\end{align*}

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)

"abcdefg"

["abcdefg", "def", "g"]

["abc", "def", "g", ",", ",", ",", ...

["abc"; "def", "g", ",", ",", ",", ..., ...

["abcdefg", "defg", "g", ",", ",", ",", ...

"abcdefg"
Example: Adding Commas

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

"1,234,567"

"765,432,1"

["765","432","1"]

"7654321"

"1234567"
Example: transpose

\[
\text{transpose} \quad :: \quad [[a]] \rightarrow [[a]]
\]

\[
\text{transpose} \quad [\quad] \quad = \quad []
\]

\[
\text{transpose} \quad ([\quad] : \text{xss}) \quad = \quad \text{transpose} \quad \text{xss}
\]

\[
\text{transpose} \quad ((x : \text{x}) : \text{xss})
\]

\[
= \quad (x : [h \mid (h : t) \leftarrow \text{xss}])
\]

\[
: \quad \text{transpose} \quad (\text{x} : [t \mid (h : t) \leftarrow \text{xss}])
\]

Example:

\[
\text{transpose} \quad [[[1,2,3],[4,5,6]]] \quad = \quad [[1,4],[2,5],[3,6]]
\]
Example: say

Say> putStrLn (say "hello")

<table>
<thead>
<tr>
<th>H</th>
<th>H</th>
<th>EEEE</th>
<th>L</th>
<th>L</th>
<th>OOO</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>H</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>O</td>
</tr>
<tr>
<td>HHHHH</td>
<td>EEEE</td>
<td>L</td>
<td>L</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>E</td>
<td>L</td>
<td>L</td>
<td>O</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>EEEE</td>
<td>LLLLL</td>
<td>LLLLL</td>
<td>OOO</td>
</tr>
</tbody>
</table>

Say>
... continued:

```haskell
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"
```

```plaintext
  A
  A  A
   AAAAA
  A  A
  A  A

  A  A  A
  A  A  A
   AAAAA
  A  A  A
  A  A

  A  A  A  A  A
  A  A  A  A  A
   AAAAA  AAAAA
  A  A  A  A  A
  A  A  A

  A  A  A  A  A  A  A
  A  A  A  A  A  A  A
   AAAAA  AAAAA  AAAAA
  A  A  A  A  A  A  A
  A  A  A

  A  A  A  A  A  A  A
  A  A  A  A  A  A  A
   AAAAA  AAAAA  AAAAA
  A  A  A  A  A  A  A
  A  A

  A  A  A  A  A
  A  A  A
   AAAAA
  A  A
  A  A

  A  A  A
  A  A
   AAAAA
  A  A
  A  A

  A  A
  A  A
   AAAAA
  A  A
  A  A

  A  A
  A  A
   AAAAA
  A  A
  A  A

  A  A
  A  A
   AAAAA
  A  A
  A  A

  A
  A
   AAAAA
  A
  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 …)