CS 457/557: Functional Languages

Folds
Today’s topics:

• 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 …
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 $\text{fold} (\oplus) n \; xs$
- Many useful functions on lists can be described in this way.
Graphically:

\[
\begin{array}{c}
e_1 : e_2 : e_3 \quad \text{foldr} (\oplus) \quad n \\
\end{array}
\]
Example: sum

\[
\text{sum} = \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`

\[
\begin{align*}
\text{cons } x \text{ ys} &= f x : \text{ys} \\
\text{map } f &= \text{foldr } (\lambda x \text{ ys} \to f x : \text{ys}) \text{ []}
\end{align*}
\]
Example: filter

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

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

\[ \text{foldr \hspace{0.5cm} cons \hspace{0.5cm} nil \hspace{0.5cm} } [] \hspace{0.5cm} = \hspace{0.5cm} \text{nil} \]

\[ \text{foldr \hspace{0.5cm} cons \hspace{0.5cm} nil \hspace{0.5cm} } (x : xs) \hspace{0.5cm} = \hspace{0.5cm} \text{cons \hspace{0.5cm} } x \hspace{0.5cm} (\text{foldr \hspace{0.5cm} cons \hspace{0.5cm} nil \hspace{0.5cm} } \hspace{0.5cm} xs) \]
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 $(\oplus)$ 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:
  \[
  \begin{align*}
  \text{sum} \; \text{xs} + \text{sum} \; \text{ys} &= \text{sum} \; (\text{xs} ++ \text{ys}) \\
  \text{product} \; \text{xs} \ast \text{product} \; \text{ys} &= \text{product} \; (\text{xs} ++ \text{ys}) \\
  \text{concat} \; \text{xss} ++ \text{concat} \; \text{yss} &= \text{concat} \; (\text{xss} ++ \text{yss}) \\
  \text{and} \; \text{xs} \;&\& \text{and} \; \text{ys} &= \text{and} \; (\text{xs} ++ \text{ys}) \\
  \text{or} \; \text{xs} \; || \; \text{or} \; \text{ys} &= \text{or} \; (\text{xs} ++ \text{ys})
  \end{align*}
  \]
foldl:

• There is a companion function to foldr called foldl:

\[
\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\text{foldl}\ s\ n\ []\ =\ n \\
\text{foldl}\ s\ n\ (x:xs)\ =\ \text{foldl}\ s\ (s\ n\ x)\ xs
\]

• For example:

\[
\text{foldl}\ s\ n\ [e_1, e_2, e_3] \\
\quad =\ s\ (s\ (s\ n\ e_1)\ e_2)\ e_3 \\
\quad =\ ((n\ `s`\ e_1)\ `s`\ e_2)\ `s`\ e_3
\]
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? ... (to be continued)
foldr1 and foldl1:

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

  \[
  \text{foldr1} :: (a \to a \to a) \to [a] \to a \\
  \text{foldr1 } f \ [x] = x \\
  \text{foldr1 } f \ (x:xs) = f \ x \ (\text{foldr1 } f \ xs)
  \]

  \[
  \text{foldl1} :: (a \to a \to a) \to [a] \to a \\
  \text{foldl1 } f \ (x:xs) = \text{foldl } f \ x \ xs
  \]

- 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: Folds on Trees

foldTree :: t -> (t -> Int -> t -> t) -> Tree -> t
foldTree leaf fork Leaf = leaf
foldTree leaf fork (Fork l n r)
    = fork (foldTree leaf fork l) n (foldTree leaf fork r)

sumTree :: Tree -> Int
sumTree = foldTree 0 (\l n r -> l + n + r)

catTree :: Tree -> [Int]
catTree = foldTree [] (\l n r -> l ++ [n] ++ r)

treeSort :: [Int] -> [Int]
treeSort = catTree . foldr insert Leaf