1 Foldl and Foldr

In class we discussed the \textit{foldl} function which is like \textit{foldr} but which associates to the left.

\begin{verbatim}
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op id [] = id
foldr op id (x:xs) = x `op` foldr op id xs
\end{verbatim}

\begin{verbatim}
fold :: (b -> a -> b) -> b -> [a] -> b
fold op acc [] = acc
foldl op acc (x:xs) = foldl op (acc `op` x) xs
\end{verbatim}

We have the following example computations.

\begin{verbatim}
foldr (+) 0 [1,2,3] 
⇝ 1 + (foldr (+) 0 [2,3])
⇝ 1 + (2 + (foldr (+) 0 [3]))
⇝ 1 + (2 + ( 3 + (foldr (+) 0 [])))
⇝ 1 + (2 + ( 3 + 0))
⇝ 1 + (2 + 3)
⇝ 1 + 5
⇝ 6
\end{verbatim}

\begin{verbatim}
foldl (+) 0 [1,2,3] 
⇝ foldl (+) (0 + 1) [2,3]
⇝ foldl (+) ((0 + 1)+ 2) [3]
⇝ foldl (+) (((0 + 1)+ 2) + 3) []
⇝ ((0 + 1)+ 2) + 3
⇝ (1 + 2) + 3
⇝ 3 + 3
⇝ 6
\end{verbatim}

We discussed that unlike \textit{foldr}, \textit{foldl} is tail-recursive, it does not require the entire list to be processed before it can start accumulating the result. Tail recursive functions are more efficient because they do not use up stack space storing the partial results waiting to be evaluated (as can be seen in the example with \textit{foldr} above.)
The function fold’ forces evaluation of the accumulated value before further unfolding the recursive call.

\[\text{foldl’ } (+) \ 0 \ [1,2,3] \Rightarrow \text{foldl’ } (+) \ (0 + 1) \ [2,3] \Rightarrow \text{foldl’ } (+) \ (1) \ [2,3] \Rightarrow \text{foldl’ } (+) \ (1 + 2) \ [3] \Rightarrow \text{foldl’ } (+) \ (3) \ [3] \Rightarrow \text{foldl’ } (+) \ (3 + 3) \ [] \Rightarrow \text{foldl’ } (+) \ (6) \ [] \Rightarrow 6\]

A recursive function is tail recursive if the final result of the recursive call is the final result of the function itself. If the result of the recursive call must be further processed (say, by adding 1 to it, or consing another element onto the beginning of it), it is not tail recursive. This definition may be a little opaque, but the main thing to notice is that the topmost function in the recursive call in the definition of foldl is to the function foldl itself. A nice discussion an more formal definition can be found at [http://www.haskell.org/haskellwiki/Tail_recursion](http://www.haskell.org/haskellwiki/Tail_recursion).

Most recursive functions on lists can be made tail recursive by using an accumulator.

For example, the following natural definition of the len function is not tail recursive because the recursive call to len is under the +.

\[
\begin{align*}
\text{len} :: [a] & \rightarrow \text{Int} \\
\text{len} \ [] & = 0 \\
\text{len} \ (x:xs) & = 1 + \text{len} \ xs
\end{align*}
\]

We can make a tail recursive version (which has a slightly different type) by adding an accumulator as follows:

\[
\begin{align*}
\text{len_acc} :: \text{Int} & \rightarrow [a] \rightarrow \text{Int} \\
\text{len_acc} \ k \ [] & = k \\
\text{len_acc} \ k \ (x:xs) & = \text{len_acc} \ (k + 1) \ xs
\end{align*}
\]

Then len can be defined in terms of len_acc by starting off with the accumulator value 0:

\[
\text{len} = \text{len_acc} \ 0
\]
We can define a tail recursive version of \(len\) hiding the \(len\_acc\) version using a where clause as follows:

\[
len :: [a] \rightarrow \text{Int} \\
len = \text{len\_acc} 0 \\
\text{where } \text{len\_acc} k [] = k \\
\text{len\_acc} k (x:xs) = \text{len\_acc} (k + 1) xs
\]

**Problem 1.1.** Write a tail recursive version of the \(filter\) function using an accumulator. The non-tail recursive version is as follows:

\[
filter :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
filter p [] = [] \\
filter p (x:xs) = \text{if } (p x) \text{ then } x : \text{filter } p \text{ } xs \text{ else } \text{filter } p \text{ } xs
\]

**Problem 1.2.** Write a tail recursive version of the \(rev\) function using an accumulator. The non-tail recursive version is as follows:

\[
rev :: [a] \rightarrow [a] \\
rev [] = [] \\
rev (x:xs) = \text{rev } xs ++ [x]
\]

Folds are rich - they can be used to implement other list functions. For example, we can implement reverse using \(\text{foldr}\) and \(\text{foldl}\) as follows\(^1\).

\[
\text{revr} = \text{foldr } (\lambda y \text{ } ys \rightarrow \text{ys } ++ [y]) [] \\
\text{revl} = \text{foldl } (\text{flip } (:)) []
\]

**Problem 1.3.** Use \(\text{foldl}\) (or \(\text{foldr}\)) to implement \(\text{map}\) and \(\text{filter}\).

---

\(^1\) I should have done another step of the last example in class ... because I was wrong! \(\text{foldl } (\text{flip } (:)) []\) reverses it, it is not the identity.