FUNCTIONAL PROGRAMMING MT2020

Sheet 3

5.1 The predefined functions

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
```

divide a list into an initial segment and the rest, so that $take\ n\ xs +\!\!+ drop\ n\ xs = xs$ and $take\ n\ xs$ is of length n or $length\ xs$, whichever is less.

Write your own definitions for these functions and check that they give the same answer as the predefined functions for some representative arguments. Is $take\ n\ xs$ strict in n? Is it strict in xs? Can it be strict in neither?

5.2 Is map strict? Is $map\ f$ strict?

5.3 Define a function $evens :: [a] \to [a]$ which returns a list of the elements of its input that are in even numbered locations:

```
*Main> evens ['a'..'z']
"acegikmoqsuwy"
```

and a function *odds* of the same type which returns the remaining elements. (Hint: you might use the one function in defining the other...)

Suppose you need both $evens\ xs$ and $odds\ xs$ for the same xs. Find an alternative definition for

```
> alts :: [a] -> ([a],[a])
> alts xs = (evens xs, odds xs)
```

which calculates the result in a single pass along the list.

Ideally, you should derive the definition showing that it is right.

6.1 In the lecture, *zip* was defined by two equations whose left-hand side patterns overlapped. The order of these two equations matters: what happens if they are switched? Find a set of equations defining *zip* whose order does not matter.

6.2 The predefined function

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

clearly, from its type, is related to *zip*. Give a definition of *zipWith* in terms of *zip* and other standard functions.

In practice, *zipWith* is defined directly and *zip* is then defined in terms of *zipWith*. Write a recursive definition of *zipWith* and use it to define *zip*.

6.3 Write (perhaps using *unfold* to do so) a function

```
> splits :: [a] -> [(a,[a])]
```

for which *splits xs* is a list of all the $(x, as +\!\!+ bs)$ that satisfy $as +\!\!+ [x] +\!\!+ bs = xs$, so that you can define

```
> permutations [] = [[]]
> permutations xs =
>     [ x:zs | (x,ys) <- splits xs, zs <- permutations ys ]
```

6.4 The function *permutations'*

```
> permutations' :: [a] -> [[a]]
> permutations' [] = [[]]
> permutations' (x:xs) =
>       [ zs | ys <- permutations' xs, zs <- include x ys ]
```

has the form of a fold, as (almost) does *include*

```
> include :: a -> [a] -> [[a]]
> include x    []  = [[x]]
> include x (y:ys) = (x:y:ys) : map (y:) (include x ys)
```

Rewrite them to use *fold* (or *foldr*) and no explicit recursion.

6.5 If it were defined by

```
> unfold :: (a->Bool) -> (a->b) -> (a->a) -> a -> [b]
> unfold null head tail =
>           map head . takeWhile (not.null) . iterate tail
```

a call of *unfold* would traverse the result list three times: once to generate the result of *iterate*, once to check for elements satisfying *null*, and once to apply *head* to each element. Find a recursive definition of *unfold* which, by doing all three of these at once, reduces the overhead of doing this.

Geraint Jones, 2020