5 Defining functions on lists

When a new data type is introduced by a **data** declaration, such as

```
data Bool = False | True
```

functions from that type are naturally defined by pattern matching using the constructors.

```
not :: Bool -> Bool
not False = True
not True  = False
```

Notice that the constructors are *constants*, and you cannot pattern match with any other expressions that happen to be equal to them.

More generally, pattern matching can cover a range of values and bind local variables to the values of components

```
data Either a b = Left a | Right b

either :: (a -> c) -> (b -> c) -> Either a b -> c
either left right (Left x)  = left x
either left right (Right y) = right y
```

The names of *left* and *right* are chosen because *either Left Right* is the identity on *Either a b*.

Similarly, functions from a recursive data type like

```
> data List a = Nil | Cons a (List a)
```

will be definable by pattern matching

```
f :: List a -> ...
f  Nil         = ...
f (Cons x xs) = ... x ... xs ...
```

though it would not be surprising were there a recursive call of f on xs.

The test for emptyness,

```
null :: [a] -> Bool
null    []  = True
null (x:xs) = False
```

is defined by pattern matching. Since x and xs are not used, indeed the type tells you that they are not used, the pattern matching non-null lists could have been `null (_:_)`. Note that *null* has to be strict, because the pattern matching has to determine which equation applies.

A more interesting example is *map*, which was introduced earlier as

```
map :: (a -> b) -> ([a] -> [b])
map f xs = [ f x | x <- xs ]
```

which you can show satisfies

$$\begin{aligned}
map\ f\ [\,] &= [f\ y \mid y \leftarrow [\,]] \\
&= [\,] \\
map\ f\ (x : xs) &= [f\ y \mid y \leftarrow (x : xs)] \\
&= [f\ x] +\!\!+ [f\ y \mid y \leftarrow xs] \\
&= f\ x{:}map\ f\ xs
\end{aligned}$$

These two equations serve as (and are the standard) definition of map. They identify the value of $map\ f$ on any finite list, and on infinite lists.

5.1 Partial functions

Some functions will be partial:

```
> head :: [a] -> a
> head (x:_) = x

> tail :: [a] -> [a]
> tail (_:xs) = xs
```

and can be defined without giving a second equation. Applying such a function to values which do not match is an error.

The other end of a non-empty list can be accessed by

```
> last :: [a] -> a
> last [x]    = x
> last (_:xs) = last xs
```

The $[x]$ notation is just an abbreviation for the pattern $(x : [\,])$ made only of constructors ($[\,]$ and $(:)$) and the variable x which matches the (last) element of a singleton. The second equation overlaps with the first, so the order of these equations matters. You might prefer

```
last (_:y:ys) = last (y:ys)
```

in which the pattern matches only lists of at least two elements, and so is disjoint from $[x]$. The disadvantage of this equation is that (when read as a rewriting rule) it takes $y : ys$ apart into its components, and then puts them together with a new $(:)$.

Catenation, $xs +\!\!+ ys = concat[xs, ys]$, also follows as similar scheme.

```
> (++) :: [a] -> [a] -> [a]
> []     ++ ys = ys
> (x:xs) ++ ys = x:(xs++ys)
```

The resulting function is strict in left argument, $\perp +\!\!\!+ [3,4] = \perp$ because of the pattern matching; but not strict in right $[1,2] +\!\!\!+ \perp = 1 : 2 : \perp \neq \perp$. (You can tell that $1 : 2 : \perp \neq \perp$, because $head\ (1 : 2 : \perp) = 1 \neq \perp = head\ \perp$.)

Notice that the cost of (the $(+\!\!\!+)$ in) $xs +\!\!\!+ ys$ is proportional to $length\ xs$.

```
> length :: [a] -> Int
> length    []  = 0
> length (_:xs) = 1 + length xs
```

The same pattern of recursion happens in map, $(+\!\!\!+ bs)$ and $length$.

5.2 A natural pattern

This same pattern also occurs in functions such as

```
> sum :: Num a => [a] -> a
> sum    []  = 0
> sum (x:xs) = x + sum xs
```

and $product$, in $filter\ p\ xs = [x \mid x \leftarrow xs, p\ x]$

```
> filter :: (a -> Bool) -> [a] -> [a]
> filter p    []  = []
> filter p (x:xs) | p x       = x:rest
>                 | otherwise = rest
>       where rest = filter p xs
```

and in $takeWhile\ p\ xs$ which returns a maximal initial segment of xs all of which satisfies p

```
> takeWhile :: (a -> Bool) -> [a] -> [a]
> takeWhile p [] = []
> takeWhile p (x:xs) | p x       = x:rest
>                    | otherwise = []
>       where rest = takeWhile p xs
```

and many others, including $concat\ xss = [x +\!\!\!+ xs \leftarrow xss, x \leftarrow xs]$

```
> concat :: [[a]] -> [a]
> concat      []  = []
> concat (xs:xss) = xs ++ concat xss
```

Abstracting this pattern leads to

```
> fold :: (a -> b -> b) -> b -> [a] -> b
> fold cons nil    []  = nil
> fold cons nil (x:xs) = cons x (fold cons nil xs)
```

This is (nearly) a standard function: it is essentially *foldr* and *foldr* is defined this way in most books. In more recent implementations of Haskell, *foldr* has a slightly more abstract type

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

The list type constructor is *Foldable*, so

```
fold = foldr :: (a -> b -> b) -> b -> [a] -> b
```

but you can (almost always) use *foldr* for *fold*.

It is often possible to spot the right *cons* and *nil* values to implement a given function. However if a function is a fold, it is always possible to compute them.

Suppose that $map\ f = fold\ cons\ nil$, then solve for

$$
\begin{aligned}
& nil \\
= \quad & \{\text{definition of } fold\} \\
& fold\ cons\ nil\ [\,] \\
= \quad & \{\text{assumption}\} \\
& map\ f\ [\,] \\
= \quad & \{\text{definition of } map\} \\
& [\,]
\end{aligned}
$$

Solving for *cons* is slightly harder:

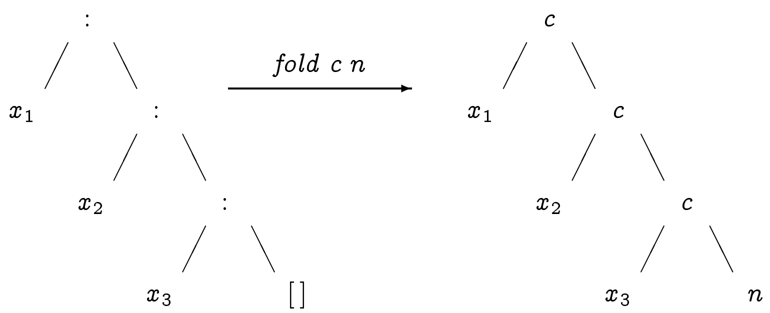
$$
\begin{aligned}
& cons\ x\ (fold\ cons\ nil\ xs) \\
= \quad & \{\text{definition of } fold\} \\
& fold\ cons\ nil\ (x : xs) \\
= \quad & \{\text{assumption}\} \\
& map\ f\ (x : xs) \\
= \quad & \{\text{definition of } map\} \\
& f\ x : map\ f\ xs \\
= \quad & \{\text{assumption, for a smaller argument}\} \\
& f\ x : fold\ cons\ nil\ xs
\end{aligned}
$$

and whilst this equation is not itself a definition of *cons*, it is certainly satisfied if $cons\ x\ ys = f\ x : ys$ for all ys.

$$\begin{aligned} sum &= fold\ (+)\ 0 \\ product &= fold\ (\times)\ 1 \\ (\mathbin{+\!\!+} bs) &= fold\ (:)\ bs \\ concat &= fold\ (\mathbin{+\!\!+})\ [\,] \end{aligned}$$

and so on.

The effect of *fold* is to substitute its arguments for the constructors, (:) and [ ], of a list; for example $fold\ c\ n$ applied to $[x_1, x_2, x_3]$



Notice that $fold\ (:)\ [\,] = id$.

In the same way *either left right* substitutes *left* and *right* for the constructors *Left* and *Right* of $Either\ a\ b$ and you can think of it as the fold for *Either* types, and $either\ Left\ Right = id$.

Given a data type definition, there is a natural fold function which substitutes its arguments for the constructors of the type, and which when applied to the constructors yields the identity function. The *fold* function is recursive where the type is recursive.

As a footnote: the fold is unique up to the order in which the arguments appear. The order of the alternatives in a **data** definition is almost immaterial, and it might have seemed more natural to have the arguments to *fold* in the same order as the constructors of the list type. However that proves confusing because of the history of *foldr*.