Chetan Mistry 38523

Foldr Report

Foldr is a Haskell function defined in the Prelude which is defined as:

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

What this means is that it takes a function, which takes something of type a and something of type b and produces something of type b, as well as taking something of type b and a list of a's to produce a single thing of type b. A reason why foldr is a useful function is because it enables Haskell programmers to use recursion since iterative loops do not exist in Haskell.
In Haskell, a list is a data structure that can store multiple elements and each element has to be of the same type. I.E. a list can store any number of characters or any number of integers, however, a single list cannot store both some characters and some integers at the same time.
How foldr works is by applying a function to each item in a list and then accumulating it in some value.
One way of thinking about this is to take a list of a's and break that down into it's most basic form by using its individual elements and cons (:)
I.e. the list:

```
[1,2,3,4,5,6]
```

Is the same as writing:

```
1 : 2 : 3 : 4 : 5 : 6 : []
```

What foldr does, is to replace each cons with some function f, and replace the empty list with some value k, eg.

```
1 : 2 : 3 : 4 : 5 : 6 : []
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
1 f 2 f 3 f 4 f 5 f 6 f k
```

So by using the function sum as an example of a fold, which will take in a list of numbers and calculate the total sum of the list, it shows how the function f can be replaced with the mathematical operator (+) resulting in the following:

```
1 f 2 f 3 f 4 f 5 f 6 f k
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
1 + 2 + 3 + 4 + 5 + 6 + k
```

To complete the definition of sum as a fold, k needs to have some value; since the sum of the numbers between 1 to 6 is already accounted for, it leaves that k is equal to 0. The code therefore which defines sum as a fold is as follows:

```
sum :: [Integer] -> Integer
sum xs = foldr (f) k xs where
     f = (+)
     k = 0
```

The above function has an alternative definition which makes use of Leibniz's Law and is as follows:

```
sum :: [Integer] -> Integer
sum = foldr (f) k where
     f = (+)
     k = 0
```

These demonstrate how the function sum can be written as a fold, but foldr can be described in an even more general way like so:

```
1 : 2 : 3 : 4 : 5 : ... : n : []

f1 (f2 (f3 (f4 (f5 (...(fn k)))))
```

By taking the general template of the foldr function above, it becomes much easier to break down seemingly complicated function, and then write them down as a fold by replacing the function f and base value k, accordingly.

There is an alternative definition of foldr which is:

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

This definition includes the Type Class Foldable which means that the value(s) supplied to foldr which is of type a, must also be Foldable otherwise foldr won't work.