Folds

> Folds are all powerful. All hail the fold. To understand the true power of folds checkout http://www.origami-fun.com :)

Spot the Difference!

There are three differences between the two functions below:

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

function' :: Num a => [a] -> a
function' []       = 1
function' (x:xs)   = x * function' xs
```

The more astute of you may have noticed that the first function is just `sum`, and the second is `product`. Notice how these functions are EXTREMELY similar? The way that they have a base case for the empty list, and how an operation is sandwiched between the first element of the list and the `function` of the rest of the list.

Introduction

Folds are a generic way of crushing a data structure into a single representative value. The fancy maths word for them is "catamorphisms". Meow.

To make it clear what I mean by crush, consider these examples: * product crushes a list into a number that is all the elements of the list multiplied together * sum crushes a list into a number that is the sum of the elements of the list * length crushes a list into a number that represents the length of the list * then if you imagine you had a tree data type, a function that counted the number of leaves a value of type tree has would also be fold, crushing the tree into just the number of leaves that it has, forgetting all about the branches and values (such as apples) it might hold.

foldr

The simplest example of a fold is the one that works on our favourite Haskell data structure: lists. `foldr` is a function included in Haskell's prelude:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k []       =  k
foldr f k (x:xs)   =  x `f` foldr f k xs
```

It is extremely useful because most functions that involve reducing a list can be defined using `foldr`. Folds take advantage of similarity between functions like `sum` or `product`. Looking at the definition of `foldr` you can see how the similarities are taken advantage of since `k` is a label for the base case and `f` is the operation (aka the two obvious differences in our little game of Spot the Difference). They can also do much more exciting things that pattern matchers only dream of.

foldr - Type

`foldr` is a function that manipulates lists. Unlike with pattern matching (where the function is told what to do case by case) the base case doesn't have to be specified since `foldr` deals with that. It does this by breaking a list down into its parts:

```
[1,2,3,4] -> 1 : 2 : 3 : 4 : []
```

The `cons (:)` is then replaced with a function of type `(a -> b -> b)`. Notice that this is a generalisation of the type of `cons (a -> [a] -> [a])`. The empty list is also exchanged. It is replaced by something of type b, the same type as the result. For example, to sum a list of integers the cons would be replaced by (+) and the empty list would be replaced by 0:

```
foldr (+) 0 [1,2,3,4] = 1 + 2 + 3 + 4 + 0 = 10
```

foldr - Definition

The first line of the definition deals with the base case: what foldr outputs if given the empty list, or when only the empty list remains. When given the empty list foldr outputs the k value. It is because of this line that when a function is defined using foldr the base case doesn't have to be specified since just giving it a value for k does this.

The second line is the recursive step. It takes the first value of the list and sandwiches the function f between it and the rest of the folded list, which is created by calling the foldr function within itself (recursion). This changes the separator of the elements in the list from cons to whatever f is given. So, if foldr was called with cons and the empty list it would output the same list, making it the identity function.

Conclusion

Folds are a very powerful tool in Haskell, providing a fantastic alternative to pattern matching. Defining a function as a fold instead of with pattern matching usually makes a more concise function, leading to tidier code. However, it is more difficult to see what a fold is doing. It is like creating a sculpture with origami instead of papier-mâché: much cleaner provided one knows what they are doing.

BONUS Content

Fun Folds

The result of a fold can be anything from the product of the list to the length of the list, or even all the permutations of the list. For example, foldr can be used to create a function that takes in two lists and outputs the second list removing any elements that were in the first list:

```
remove :: Eq a => [a] -> [a] -> [a]
remove a b = foldr (consif (not . flip elem a)) [] b

consif :: (a -> Bool) -> a -> [a] -> [a]
consif f x (xs)
   | f x         =  x :(xs)
   | otherwise   =  (xs)
```

```
remove "DON'T" "i DON'T like folds" = "i  like folds"
```

consif is a function that will add an element to a list if the condition provided is met. Since this function wants to output the second list removing any elements that are in the first list, the condition given to consif is whether the item is not in the first list.

Alternatively, foldr could be used to define a function that removes adjacent duplicates from a list. Especially handy if one has a friend that tyyypess likkeee thisssss.

```
remdups :: Eq a => [a] -> [a]
remdups = foldr f []
       where
           f x []            = [x]
           f x (y:ys)
               | x == y      =  y:ys
               | otherwise   =  x:y:ys
```

```
remdups "tyyypess likkeee thisssss" = "types like this"
```

When defining folds the `f` function can either be written in place (like the first example), or it can be declared as a sub function using a `where` clause. The `f` in the first example is a composition of other smaller functions.

Folding Trees

Folding is not just exclusive to lists. The `foldr` function can be adapted to fold other data structures, such as Trees. Trees are a data structure consisting of `nodes` and `tips`. Nodes store values and link to left and right subtrees. Subtrees can either be more `nodes` or they can be `tips`, which store nothing and mark the end of a 'branch'. To convert `foldr` into a folding function for trees, say `foldTree`, the constructors for list and trees can be compared:

```haskell
data Tree a  =  Tip
             |  Node (Tree a) a (Tree a)

data [] a  =  []
           |  a : [a]
```

When put side by side it is easier to see that a tree's version of the empty list is a `Tip`, and the equivalent of the `cons` constructor is a `Node`. This makes it easier to find out what the type for `foldTree` is.

The parameters `foldr` takes in are something to replace the `[]` with, something to replace `(:)` with, and the list to operate on; the parameters for `foldTree` will be something to replace the `Tip` with `(b)`, something to replace the `Node` constructor with `(b -> a -> b -> b)`, and the tree that is to be folded `(Tree a)`.

```haskell
foldTree :: b -> (b -> a -> b -> b) -> Tree a -> b
```

Just like the `f` that replaces the `cons` in `foldr`, the type of the `f` that replaces the `Node` has a type that is a generalisation of the type of the `Node`. `Node` has type: give me a left-hand subtree `(b)`, a value to place at the `Node (a)`, and right-hand subtree `(b)` and I'll produce you a new tree `(b)`, hence the function that replaces the `Node` will have type `(b -> a -> b -> b)`. `foldtree` is constructed just like `foldr`:

```haskell
foldTree t n Tip            =  t
foldTree t n (Node l x r)   =  n (foldTree t n l) x (foldTree t n r)
```

When presented with a `Tip` `foldTree` will output the value to replace a `Tip` with `t`. When presented with a `Node` it will replace the `Node` with the function intended to replace `Node`s with `n`, which is expecting three parameters, these will be the result of folding the left subtree, the value in the `Node`, and the result of folding the right subtree. Using this knowledge `foldr` functions can be converted into `foldTree` functions:

Summing/finding the product of a tree:

```haskell
sumTree :: Tree Int -> Int
sumTree = foldTree t n
        where
               t        =  0
               n l x r  =  l + x + r

productTree :: Tree Int -> Int
productTree = foldTree t n
            where
               t        =  1
               n l x r  =  l * x * r

Counting the nodes/tips:

```haskell
nodes ::  Tree a -> Int
nodes = foldTree tip node
   where
```
```

```haskell
      tip         =  0
      node l x r  =  l + r + 1

tips ::  Tree a -> Int
tips = foldTree tip node
   where
      tip         =  1
      node l x r  =  l+r
```

Algebras

Don't worry not school algebra. A much better one!

"Algebra" is the name we give to the collection of functions that tell you what to do in each case of the fold. So a list algebra is the `f` and the `k`, and a tree algebra consists of `tip` and `node`.

Since we have a collective name for these functions, looking back at the type of `foldTree` is actually starting to annoy me because I have to pass it two things: something to replace a `Tip` with, and something to replace a `Node` with. With folds over data types you need to pass in a parameter per constructor. How laborious! I mean I *suppose* that it is not that bad with lists or trees since there are only two constructors but what if I had a bigger data type with many constructors. Wouldn't it be much better if I could just bundle these things up into one thing!

Well my friend this is what *algebras* are for! An `alg` is a data structure that is specifically made as a big holdall for your constructor replacements. Here is a `TreeAlg` for our `Tree`:

```haskell
data TreeAlg a b = TreeAlg b (b -> a -> b -> b)
```

If you compare this to the first two parameters of `foldTree` you can see that this data type is one with a single constructor followed by all the parts that need to be given to the fold. You can even rewrite it with new lines and comments so that you don't forget what each section is for:

```haskell
data TreeAlg a b = TreeAlg
                         b                    --  tip
                      (  b -> a -> b -> b)    --  node
```

Now if we go and redefine `foldTree` the type looks much neater:

```haskell
foldTree :: TreeAlg a b -> Tree a -> b
```

Basically it is now saying "Give me instructions on on how to deal with a `Tree` and an actual `Tree a` and I will be AWEsome and present you with a pretty `b` thing".

```haskell
foldTree (TreeAlg tip node) Tip                  =  tip
foldTree alg@(TreeAlg tip node) (Node l x r)  =  node (foldTree alg l) x (foldTree alg r)
```

You may be wondering what the mysterious `@` is doing there, well that just gives me a way to refer to the `TreeAlg` as a whole meaning that for the recursive step I don't have to type the big long `TreeAlg tip node`.

Let's make a simple example to see how you would use an `alg`. The following function counts the number of `Tips` a `Tree a` has:

```haskell
noTip :: Num b => Tree a -> b
noTip = foldTree alg
          where
              alg         =  TreeAlg tip node
              tip         =  1
              node l x r  =  l + r
```

When using `algs` all the information to do with what the fold will do is in the `where` clause. This is where you create the `TreeAlg` and decide what `tip` and `node` will do. You do not need to specify `tip` and `node` on lines all to themselves but it does make it very clear and you don't have to use lambdas.

We can do better though. Notice how in `foldTree` how you have to pattern match on all the parts of the `treeAlg` so that you can access the fields you want on the RHS of the equals sign? Well, record syntax allows you to make a better `treeAlg` in which you can name the fields :0!

```
data TreeAlgBetter a b  =  TreeAlgBetter  {  tip   ::     b
                                          ,  node  ::  (  b   -> a -> b -> b)
                                          }

foldTree alg  Tip         =  tip  alg
foldTree alg (Node l x r) = (node alg) (foldTree alg l) x (foldTree alg r)
```

Much easier! One thing to note is the type of `tip` or `node`:

```
< tip   ::  TreeAlgBetter a b -> b
< node  ::  TreeAlgBetter a b -> b -> a -> b -> b
```

The types of these are not quite what you would expect. They do not match the type that you state in `TreeAlgBetter`, they have an additional parameter on the front, this is because you need to give these functions a `TreeAlgBetter` so that they know what to do. This also explains why the RHS of the equals in `foldTree` says `tip alg` or `node alg`. See the record syntax note for more :-)