Functions as Values: Difference Lists

In this section we discuss an example where our primitive values are themselves functions. We will see Difference Lists, which are an approach to efficiently appending on the right.

We will also discuss the concept of function composition in this context, and the "point-free programming style" that it allows.

Reading

- Section 7.5

Difference Lists

We have already seen the list append operation:

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

Looking at this a bit further it becomes clear that appending requires the copying of the entire first list, and therefore is affected by the length of the first list, but it is not at all affected by the length of the second list.

> In general, lists are set up so that it is easy/efficient to add elements to their left, but not to their right. But adding to the right of a list has a cost proportional to the length of the list.

In many practical applications we want to in fact add to the right of a list. Imagine reading through a configuration file and creating a resulting web page as you go. You will be constantly adding new entries to the end of the string, which corresponds to the right side of a list. As the length of the string grows, adding one more character to the end becomes expensive as all the earlier parts get copied each time. We saw the same problem when we tried to reverse a list in a naive way.

A solution to this problem is presented by difference lists. The idea of a difference list is that it does not try and do the appending right away, but it simply sets up something that will happen in the future. These steps are then performed in reverse order. So instead of continuously appending to the right of the previously constructed list, you first construct all the parts that go to the right, and only then append the list to their left.

> Difference lists delay the appends and perform them in reverse order, growing the list towards the left than towards the right.

For instance suppose that there are 4 strings to be added in total in this process, A, B, C, D. Instead of saying:

```
Start with A
Append B to the end of A
Append C to the end of (A++B)
Append D to the end of ((A++B) ++ C) ++ D
Return the result
```

We say:

```
Remember that you will put A to the left of the remaining result.
Remember that you will put B to the left of the remaining result, before adding the A.
Remember that you will put C to the left of the remaining result, before adding the B.
Remember that you will put D to the left of the remaining result, before adding the C.
Since you have no more strings to add, start with an empty string [] and work backwards:
Add the D to the left of []
Add the C to the left of (D ++ [])
Add the B to the left of (C ++ (D ++ []))
Add the A to the left of (B ++ (C ++ (D ++ [])))
Return the result.
```

So the key difference here is that the list appends end up happening from right to left, which is more efficient in the long run. But in order to achieve that, we had to delay the appending of the strings as we encountered them, until we reached the end.

This leads us to the structure of *difference lists*. A difference list is effectively a promise to append a list later:

```
type dList a = [a] -> [a]
```

It is meant to say "You give me the rest of the list, and I'll add the extra bits to its left". The actual value is a function, that is given a list representing the rest of the result and adjusts it, hopefully by adding its bit to the front.

This is a key mental shift. Our "values" are really functions. We want to think of them as a value, but at the same time they are functions that we need to feed with inputs at some point in time. But before we do that we need to be able to for instance append them. So we will need to find a way to define how to "append" one of these functions to another one of these functions, and produce yet another one of these functions as the result.

We should start with a way to obtain the actual list from a difference list, This is simple you just need to start with the empty list, then the difference list will produce its result (conceptually we are thinking of it as appending to the left of an empty list):

```
toList :: dList a -> [a]
toList f = f []
```

We can now write two functions, to form a difference list from a list, or from a single element. The type for such a function would be:

```
fromList :: [a] -> dList a
```

If we unravel what this means, keeping in mind that values of type `dList a` are actually functions `[a] -> [a]`, we get:

```
fromList :: [a] -> ([a] -> [a])
fromList xs  = \ys -> xs ++ ys
-- Can also write as:
fromList xs ys = xs ++ ys
-- Or even
fromList = (++)
```

In this setting we are thinking of a list as a difference list by saying "I will prepend this list to any list you give me."

Similarly for single elements we would have:

```
fromElem :: a -> dList a
fromElem x = \ys -> x : ys
-- Can also write as:
fromElem x ys = x : ys
-- Or even
fromElem = (:)
```

So we think of a single element as a difference list by saying "I will prepend this element to any list you give me."

We have encountered an important idea. These methods are our familiar ++ and : operators, but now considered in a new light because of the new types we have.

But still, we have not defined our most important operation, namely that of appending two difference lists together. So let us do this now. Remember that they are both functions. And we must also return a function:

```
append :: dList a -> dList a -> dList a
-- If we "unalias":
append :: ([a] -> [a]) -> ([a] -> [a]) -> ([a] -> [a])
append f1 f2 = \ys -> ...
```

This is probably the most complicated function we have written yet, in terms of types. Let us think of what it needs to do: It takes as input two difference lists, and it is supposed to produce a new difference list which amounts to appending f2 at the end of f1. What we really mean by it, since we think of both lists instead as their effects on a list to their right, is that we should first do what f2 asks us to do, then use f1 on the result of that. So the code ends up looking like this:

```
append :: dList a -> dList a -> dList a
append f1 f2 = \ys -> f1 (f2 ys)
-- Can also write:
append f1 f2 ys = f1 (f2 ys)
```

Let us see it in action in the example where both dLists came from lists. In order to think of their combination, we apply the dList to a list zs.

```
append (fromList xs) (fromList ys) zs
(fromList xs) ((fromList ys) zs)
(fromList xs) ((\ws -> ys ++ ws) zs)
(fromList xs) (ys ++ zs)
(\ws -> xs ++ ws) (ys ++ zs)
xs ++ (ys ++ zs)
```

So, when the time finally comes to compute the list concatenations, they end up being in the proper order!

Function Composition

The operation append earlier actually ended up corresponding to a very common operation, that of **function composition**. In function composition you have two functions, and you build a new function from them by applying one function first, then applying the second function to the result of the first. This is so common that it has a simple notation, namely that of a single dot:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
-- Can also write as
f . g x = f (g x)
```

With this definition in mind, we could have defined append earlier as simply append=(.).

As a trivial example of function composition, consider a linear function:

```
f x = 3 * x + 1
```

We can think of this function as a composition of two functions: We first multiply by 3, then we add 1. We can think of these two as sections. So a rather convoluted way of writing the above would have been:

```
f = (+ 1) . (3 *)
```

This doesn't really show the significance of this function, but it will hopefully help show a bit of the mechanics.

As another example, suppose we wanted to do the following: Write a function that given a list of numbers squares them then adds the results. We could write this in a number of different ways:

```
sumSquares :: Num t => [t] -> t
-- List comprehension
sumSquares xs = sum [x^2 | x <- xs]
-- Using map
sumSquares xs = sum (map (^2) xs)
-- "Point-free" using function composition
sumSquares = sum . map (^2)
```

This type of programming has a certain elegance to it: We define a function via composition of known functions, without ever having to mention the function's actual parameter. This is often called **point-free style**, some times also called **tacit programming**.

It is also at times hard to read, so use it with caution.

Practice

1. Give a point-free definition of a function that is given a list of numbers finds the maximum of the first 3 elements.
2. Give a point-free definition of a function that given a list of lists returns a list containing the heads of those lists (you do not need function composition here, this is simply a curried function with only some parameters provided).

3. Consider the following "string scrambling" process: For each character, convert it to an integer via the method `ord :: Char -> Int`, then double that integer, finally convert the corresponding integer back to a character via `chr :: Int -> Char`. Do this for each character in the string. Write this function of type `String -> String` that performs this combined task. You can do this as a partially applied `map`, where the provided function is point-free and composes the functions `ord, chr` together with a section for the multiplication.