AFP - Assignment 2 - Fringes and Fibonacci Trees

Maximo Garcia Martinez

a) mirror

```
mirror :: Tree a -> Tree a
mirror E = E
mirror (T a left right) = T a (mirror right) (mirror left)
```

b) fringe_naive

```
fringe_naive :: Tree a -> [a]
fringe_naive E = []
fringe_naive (T a E E) = [a]
fringe_naive (T _ left right) = fringe_naive(left) ++ fringe_naive(right)
```

What is the complexity of your function (measured in terms of n, the number of leaves of a tree)?
In this function we are using ++ to append two lists. The complexity of ++ is O(m), m is the length of the first list. The complexity of appending items in the worst case would be:

$$O\left(\left(\frac{n}{2}\right)^{n/2}\right)$$

The function also iterates over the tree, the complexity of this is:

$$O(4n-2)$$

The complexity in total is:

$$O\left(\left(\frac{n}{2}\right)^{n/2} + O(4n-2)\right)$$

c) fringe

```
fringe :: Tree a -> [a]
fringe t = fringe_aux t []

fringe_aux :: Tree a -> [a] -> [a]
fringe_aux E _ = []
fringe_aux (T a E E) x = a:x
fringe_aux (T a l r) x = (fringe_aux l (fringe_aux r x))
```

What is the complexity of your function? The complexity of this function varies in how the items are being appended. Now, ':' is being used which has the complexity of O(1), so at the end, the complexity of appending items will be O(n). The complexity will be:

$$O(n+(4n-2))$$

d) same_fringe

```
same_fringe :: Eq a => Tree a -> Tree a -> Bool
same_fringe t1 t2 = (fringe t1) == (fringe t2)
```

What is the complexity of your function?

$$O(2 \times (n + (4n - 2)))$$

How does it behave if the two fringes differ? Haskell is lazy, it will only compute a value when it is needed. When we compare the two fringes, it won't calculate all the values in the list, only the first two elements of the lists and compare them, if they are the same it will check the second element in both lists and compare them. If at one point, the elements differ, then it will return False, otherwise it will return True.

e)

It would evaluate both fringes first and then compare them, because an eager language is a language with functions that need arguments already evaluated, so the == function will need both list already computed.

If we want to have the same function with the same complexity using a non-easer language, we would have to create a fringe function that iterates over both trees at the same time, if in one of them we reach a leave, then we would continue with the other tree until we reach the leave and then compare both leaves. If both leaves are equal we would keep the recursion for the rest of the leaves. If we find that two leaves differ, then we would return False.

g) fibtree_naive

```
fibtree_naive :: Int -> Tree Int
fibtree_naive n | n <= 1 = T n E E
                | otherwise = T (left + right) (T left l1 l2) (T right r1 r2)
                  where
                      T left l1 l2 = fibtree_naive (n-1)
                      T right r1 r2 = fibtree_naive (n-2)
```

How much space does tree Tn occupy in memory?

$$O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

f) fibtree

```
fibtree :: Int -> Tree Int
fibtree n | n <= 1 = T n E E
          | otherwise = fibtree_aux (n-2) ((T 1 E E, T 0 E E))

fibtree_aux :: Int -> (Tree Int, Tree Int) -> Tree Int
fibtree_aux n (T v1 t1 t2, T v2 t3 t4)
    | n < 0 = T v1 t1 t2
    | otherwise = fibtree_aux (n - 1) ((T (v1+v2) (T v1 t1 t2) (T v2 t3 t4), T v1 t1 t2))
```

h)

How much memory space is occupied by the tree that gets produced by the evaluation of expression mirror (fibtree n)?

First we define F(n) which approximates the number of nodes in the tree:

$$F(0) = 1$$

$$F(1) = 1$$

$$F(n) = \varphi^n$$

The memory used by the algorithm is:

$$M(n) = (F(n) + M(n-1) + M(n-2)) \times S$$

where S is the size of a node.

j)

Why, as n increases, the first line is shown immediately, but the second line is delayed more and more?

As it is mentioned before, Haskell is a lazy language. That means that it won't evaluate all the tree and then calculate the fringe. It will do that only if it is needed. In this case we are calculating a Fibonacci tree and then its mirror. As we expect the fringe of both trees are different, so it won't evaluate all the tree, because it will find that the fringe of both trees differ early in the process. On the other hand, in the second step, the program is calculating another Fibonacci tree with n+1. The fringes of a Fibonacci tree with n and n+1 have the same elements at the beginning, so the bigger n gets, the more time it will need to evaluate both trees and compare the fringes.