Supervision work

Tudor Avram
tma33@cam.ac.uk

3rd of November 2015

1 Exercise 9.1

```
fun map f Nil = Nil
    | map f (Cons(x,xs)) = Cons(f x, fn() =>map f (xs()));
```

2 Excercise 9.2

The function *concat* can be generalised to sequences, just like any other function that acts on lists. Even though this can be achieved, the actual function would never terminate, because it will never be able to reach the end of an infinite list.

3 Exercise 9.3

Datatype declaration :

```
datatype 'a SEQ = Nil1
     | Cons1 of 'a list * (unit -> 'a SEQ);
```

Function :

```
fun seqChange (Coins, CoinVals, 0, ANS) = Cons1(Coins, ANS)
| seqChange (Conins, [ ], amt, ANS) = ANS()
| seqChange (Coins, C::CoinVals, amt, ANS) =
     if (amt < 0) then ANS()
 else seqChange(C::Coins, C::CoinVals, amt-C,
     fn() => seqChange(Coins, CoinVals, amt, ANS));
```

4 Excercise 9.4

Datatype delaration :

```
datatype 'a Ltree = TNill
      | Br of 'a * (unit ->'a Ltree) * (unit ->'a Ltree);
```

Function :

```
fun Store TNill = [ ]
      | Store (Br(X,Ls,Rs)) = X :: ( Store(Rs()) @ Store(Ls()));
```

5 Exercise 10.1

A queue implemented such as any operation takes logarithmic time, it would have a general complexity of O($\log N$), whereas the simple queue can have an amortised complexity of O(1). Therefore, the binary tree-based implementation of queues would be slower. On the other hand, the new queue implementation would be easier to use for any programmer.

6 Exercise 10.2

The implementation of queues using fixed-length arrays would not be appropriate for implementing the breadth-first search, because at any point the number of the nodes we need to store in the queue can exceed the length of the array and this would raise an error. Therefore, queues using fixed-sized arrays are not a good option for the breadth-first search.

7 Excercise 10.3

Function :

```
fun BFS (Q([],[])) = []
    | BFS q =
    let
          fun breadth Lf = BFS (qerase q)
          | breadth (Br(v,t,u)) =
               BFS(qadd(qadd(qerase q, t),u))
     in
        breadth (qhd q)
    end;
```

8 Exercise 10.4

Let f(b) = $\frac{b}{b-1}$ be the number of breadth-first searches needed to reach depth d. Then the limit $\lim_{b\to 1} f(b) = \infty$, so the Iterative Deepening would be an inappropriate algorithm to use if the branching factor b ≈ 1.
In my opinion, the appropriate algorithm to use in this case would be the breadth-first search, because when b ≈ 1, it would use almost no memory and the time complexity would be O(d) to reach depth d.

9 Exercise 10.5

If the function `fun next n = [2*n, 2*n+1]` represents a tree where every subtree is computed from the current label, next 1 would represent a binary tree whose labels are the numbers from 2^{d-1} to 2^{d} - 1, for every given depth d.

10 Exercise 11.1

Membership test :

```
fun member a [] = false
    | member a (x::xs) =
        if (a¿x) then member a (xs)
            else if (a¡x) then false
                else true;
```

Subset test :

```
fun subset [] xs = true
    | subset (n::[]) xs = member n xs
    | subset (n::ns) xs =
        if (member n xs) then subset ns xs
            else false;
```

Intersection :

```
fun inter [] [] = []
    | inter [] ys = []
    | inter (x::xs) ys =
        if (member x ys) then x :: (inter xs ys)
            else inter xs ys;
```

Union :

```
fun union [] [] = []
  | union Xs [] = Xs
  | union [] Ys = Ys
  | union (x::Xs) (y::Ys) =
    if (x<y) then x :: union Xs (y::Ys)
      else if (x>y) then y :: union (x::Xs) Ys
        else x :: union Xs Ys;
```

11 Excercise 11.2

Let two polynomials be [(n, a_n), (n-1, a_{n-1}), ..., (0, a_0)] and [(m, b_m), (m-1, b_{m-1}), ..., (0, b_0)].
Polysum is based on the merge sort algorithm, so the two resulting list of tuples will also be in descending order. Because all the terms in the initial polynomials are non-zero and the algorithm treats the case if (a+b=0.0), there will be no zero terms in the result.
Polyprod also uses an algorithm similar to merge sort, so the tuples in the result will be sorted in descending order. Also, because there are no "0"-terms in the two initial polynomials and because it uses the function *polysum* to find the final answer, there are not going to be any "0"-terms in the result.

12 Excercise 11.3

Let a = min(n,m) and b = max(n,m). In the initial part of the recursion, the function will consumate the polynomial with less terms. So, it will require *O(a)* operations to do this part.

13 Excercise 11.4

The first version of *polyprod* will go through all the elements from the first polynomial and will call the *polysum* function, which in this case takes O(n*m) complexity. Therefore, the complexity of the summations will be O(n^2 * m). The mapping will also be done n times, so it has a complexity of O(n*m). In conclusion, the final complexity of this algorithm will be O(n*m*(n+1)) = O(n*m*n) = (O(n^2 * m));

The second version of *polyprod* uses the Divide et Impera technique to reduce the number of recursive calls to log(n) and the *polysum* function will only add polynomials of the same length, so it has an amortised complexity of O(p), where p is the length of every polynomial, which, in this case, can be at most $\frac{n*m}{2}$. Therefore, the complexity of the summations is O(log(n)*n*m) . But we still need to take into accound the mapping that is done when we and up with only one element left, which takes O(m) complexity and it is done n times, so O(n*m). In conclusion, the final complexity of this version of *polysum* is O(n*m*(log(n)+1)) = O(n*m*log(n)).