Foundations of Computer Science

Supervision 4 – Solutions

9. Sequences and laziness

9.1. Conceptual questions

1. Consider the list function `concat`, which concatenates a list of lists to form a single list. Can it be generalised to concatenate a sequence of sequences? What can go wrong?

```
let rec concat = function
  | []    -> []
  | l::ls -> l @ concat ls
```

> There are two problems that may arise with the obvious implementation below. If any of the sequences is infinite, the concatenation will never proceed past this sequence. There is no problem if there are an infinite number of nonempty but finite sequences, as they will be diligently consumed and streamed out one by one in an infinite lazy sequence. An error case however is an infinite sequence of `Nil`s, which will trigger the second pattern repeatedly and lead to nontermination (without a stack overflow, as the body is a tail call).
>
> ```
> let rec concat = function
>   | Nil                      -> Nil
>   | Cons (Nil, yf)           -> concat (yf ())
>   | Cons (Cons (x, xf), yf) -> Cons (x, fun () ->
>                                  concat (Cons (xf (), yf)))
> ```
>
> The first problem can be solved by a generalised "interleave" function that looks at the heads of every list one-by-one, so it will never get stuck on an infinite list. However, this would not work for an infinite sequence of finite lists, as the second elements of the sub-sequences will never be reached.

2. Why are lazy lists (sequences) useful and why are they not "natively" supported by OCaml? How do we simulate lazy lists in OCaml and why does that not have the issues you described above?

> Lazy lists are lists where the tail is evaluated lazily: it is not touched if the tail elements are not required, and can even be infinite. OCaml does not support lazy lists due to its eager evaluation strategy: it always evaluates the tail of the list and keeps all the elements in memory, even if we only need the first few elements. We simulate lazy lists using *sequences*, which have a tail function instead of the tail: the tail function is a value so it's not evaluated by default, but we can force evaluation by applying it to `()` (the only inhabitent of the type `unit`) and get the tail sequence (of which only the head is available). This trick enables us

> to put the tail behind a "wall", and only look at it (evaluate it) if we need to.

9.2. Exercises

3. Code an analogue of `map` for sequences.

> As before, the aim of the mapping is to lift a function `f : 'a -> 'b` to sequences: `map_seq f : 'a seq -> 'b seq`. The function is very similar in structure to the list version of `map`, though we have to do some extra work to handle the tail function.

```
let rec map_seq f = function
  | Nil          -> Nil
  | Cons (x, xf) -> Cons (f x, fun () -> map_seq f (xf()))
```

4. A *lazy binary tree* is either empty or is a branch containing a label and two lazy binary trees, possibly to infinite depth. Present an OCaml datatype to represent lazy binary trees, along with a function that accepts a lazy binary tree and produces a lazy list that contains all of the tree's labels. The order of the elements in the lazy list does not matter, as long as it contains all potential tree nodes.

> The datatype for lazy binary trees combines the structure of normal binary trees with the delayed evaluation trick of sequences: instead of subtrees, we have subtree functions.

```
type 'a ltree =
      Lf | Br of 'a * (unit -> 'a ltree) * (unit -> 'a ltree)
```

> The flattening function resembles inorder tree traversal, in that we flatten the (lazy) subtrees and combine them with the root. However, given that the subtrees may be infinite, we cannot use stream concatenation; instead, we use the `interleave` function to ensure that both subtrees get involved. The order of the elements will be somewhat unusual, but we were not required to adhere to any specific ordering.

```
let rec flatten = function
  | Lf -> Nil
  | Br (n, ltf, rtf) -> Cons (n, fun () ->
                interleave (flatten (ltf())) (flatten (rtf())))
```

9.3. Optional questions

5. Code a function to make change using lazy lists, delivering the sequence of all possible ways of making change. Using sequences allows us to compute solutions one at a time when there exists an astronomical number. Represent lists of coins using ordinary lists. (*Hint*: to benefit from laziness you may need to pass around the sequence of alternative solutions as a function of type `unit -> (int list) seq`.)

> The main difference between the list and sequence implementation is that `map` and append need to be replaced with sequence-specific variations. One simple way is to use an auxiliary function which performs the map and append at the same time, but mapping only over the first sequence, and taking the second sequence as a function, thereby delaying the evaluation.
>
> ```
> let rec mapapp f s yf = match s with
>   | Nil -> yf ()
>   | Cons (x, xf) -> Cons (f x, fun () -> mapapp f (xf()) yf)
>
> let rec change till amt = match till, amt with
>   | _, 0  -> Cons ([], fun () -> Nil)
>   | [], _ -> Nil
>   | c::till, amt ->
>      if amt < c then change till amt
>                 else mapapp (cons c) (change (c::till) (amt-c))
>                             (fun () -> change till amt)
> ```

6. Code the lazy list whose elements are all ordinary lists of zeroes and ones, namely `[]`, `[0]`, `[1]`, `[0;0]`, `[0;1]`, `[1;0]`, `[1;1]`, `[0;0;0]`, ...

> The idea is that given a list of length n, we create two new lists of length $n+1$ by attaching either `0` or `1`. Starting with `[]`, we get `[0]` and `[1]`, then doing the same on these two gives us `[0;0]`, `[0;1]`, `[1;0]` and `[1;1]`, which are all possible two-element lists. Continuing the same for each generation will give us a complete list of all possible lists of `0` and `1` (or binary numbers).
>
> The following function encapsulates this idea: given an initial list `xs`, it returns the (infinite) sequence starting with the given list, then the interleaved sequences of of the recursive calls on `0::xs` and `1::xs`. We need the `interleave` function because each of the recursive calls returns an infinite sequence.
>
> ```
> let rec binaryLists xs =
>     Cons (xs, fun () -> interleave (binaryLists (0::xs))
>                                    (binaryLists (1::xs)))
> let allBinaryLists = binaryLists []
> ```

7. (Continuing the previous exercise.) A palindrome is a list that equals its own reverse. Code the lazy list whose elements are all palindromes of 0s and 1s, namely `[]`, `[0]`, `[1]`, `[0;0]`, `[0;0;0]`, `[0;1;0]`, `[1;1]`, `[1;0;1]`, `[1;1;1]`, `[0;0;0;0]`, ... You can use the list reversal function `List.rev`.

> One simple, but rather inefficient solution is to filter the result of the previous exercise, `allBinaryLists`, removing every non-palindrome list. Of course, the longer the list, the "rarer" the palindromes, so the number of lists skipped increases exponentially. Instead, we can *generate* the palindromes from `allBinaryLists` in the following way: given a list `xs`, we can create the palindromes `xs @ rev xs`, `xs @ (0 :: rev xs)` and `xs @ (1 :: rev xs)`. That is, every binary list gives rise to three unique palindromes: one of even length and two of odd length, with either a `0` or a `1` in the middle. The following function accomplishes this, given an initial nonempty sequence:
>
> ```
> let rec genPalindromes (Cons(x, xf)) =
>                   Cons (x @ List.rev x,
>         fun () -> Cons (x @ (0 :: List.rev x),
>         fun () -> Cons (x @ (1 :: List.rev x),
>         fun () -> genPalindromes (xf()))))
> let allBinaryPalindromes = genPalindromes allBinaryLists
> ```

8. With some exceptions (such as appending or concatenation), we can adapt many common list operations to work on lazy lists. In addition, lazy lists can be used to generate infinite sequences without the risk of nontermination. This exercise explores some interesting examples using the `seq` type.

   a) Define the analogues of `filter` and `zipWith` for sequences. The `zipWith` list functional is similar to `zip` but it applies a binary function to the pair it constructs. For example, `zipWith (+) [1;2;3;4] [5;6;7;8] = [6;8;10;12]` where `(+)` is the function version of the `+` operator.

   ```
   val filterS  : ('a -> bool) -> 'a seq -> 'a seq = <fun>
   val zipWithS : ('a -> 'b -> 'c) -> 'a seq -> 'b seq -> 'c seq
                = <fun>
   ```

   > Straightforward translation of the list operations to sequences.
   >
   > ```
   > let rec filterS p = function
   >   | Nil -> Nil
   >   | Cons (x, xf) ->
   >           if p x then Cons (x, fun () -> filterS p (xf()))
   >                  else filterS p (xf())
   > let rec zipWithS opr xs ys = match xs, ys with
   >   | Nil, _ -> Nil
   >   | _, Nil -> Nil
   >   | (Cons (x, xf)), (Cons (y, yf)) ->
   > ```

```
      Cons (opr x y, fun () -> zipWithS opr (xf()) (yf()))
```

b) Consider the following two definitions. What do they represent and how do they work?

```
let s = let rec s_aux (Cons (x, xf)) =
                        Cons (x * x, fun () -> s_aux (xf()))
        in s_aux (from 0)

let rec f = Cons (0, fun () ->
            Cons (1, fun () -> zipWithS (+) f (tail f)))
```

> Both of these are infinite sequences generated from some initial value. The first one takes a sequence and returns a sequence of squared elements; when called on `from 0`, it gives the infinite sequence of square numbers. Note that the same can be achieved by using `map_seq`, applying the squaring function to every element of `from 0`.
>
> The second is more interesting, as it does not start from an initial value since it's not even a function: `f` is a *recursively defined value*. This is quite unusual, but the technique arises naturally when dealing with lazy infinite data structures, where the sub-structure is some modified version of the whole structure. The sequence starts off with `0` and `1`, then we perform pairwise addition (with `zipWithS (+)`) on the sequence itself, and its tail. Hence, the third value will be `0 + 1 = 1`, the fourth will be `1 + 1 = 2`, then `1 + 2 = 3`, `2 + 3 = 5`, `3 + 5 = 8` and so on, giving us the Fibonacci sequence. This definition exploits the property that pairwise adding the elements of the Fibonacci sequence with itself shifted by one, we also get the elements of the Fibonacci sequence:
>
> ```
>   0, 1, 1, 2, 3, 5,  8, 13
> +    0, 1, 1, 2, 3,  5,  8
> =    1, 2, 3, 5, 8, 13, 21
> ```
>
> The value `f` uses this property as a way to generate the elements of the sequence by initialising the first two elements, then performing pairwise additions. Even though the sequence is referencing itself as it is being constructed, laziness means that we only ask for elements that have already been created, so there is no inconsistency. This almost-one-liner is often used to show off the power of lazy computation.

c) Define the lazy list `sieve : int seq` that uses the Sieve of Eratosthenes to calculate the infinite sequence of prime integers. *Hint*: Define an auxiliary function `sieve_aux` such that `sieve = sieve_aux (from 2)`. You may want to use `filterS` for the "sieving".

In previous examples, the infinite stream was generated from some existing stream (e.g. square numbers), or the stream itself (Fibonacci sequence). Prime numbers are different in that they cannot be generated from a simple stream or defined recursively: they are irregular and there is no known algorithm for finding the k-th prime number. The best we can do is filtering – this is exactly what the Sieve of Eratosthenes does.

The inner function `sieve_aux` does most of the work. It takes an initial input stream, and returns a stream starting with the head of the input, but continuing by the recursively constructed tail filtered by removing everything that is divisible by the head. That is, we add the first element to the list of primes, and cross out everything divisible by this prime. This would of course only work if the first element is a prime, so we initialise the list with the sequence starting from 2.

```
let sieve =
  let rec sieve_aux (Cons (p, pf)) = Cons (p, fun () ->
          sieve_aux (filterS (fun x -> x mod p <> 0) (pf())))
  in sieve_aux (from 2)
```

It is worth noting that this is far from an efficient method of generating the prime numbers, but it is quite amazing how it's all done using a few simple lines of code.

10. Queues and search strategies

10.1. Conceptual questions

1. Suppose that we have an implementation of queues, based on binary trees, such that each operation takes logarithmic time in the worst case. Outline the advantages and drawbacks of such an implementation compared with one presented in the notes.

   Binary tree operations usually take logarithmic time due to the branching structure of trees. This is a constant cost: although each operation may take more or less time (depending, for example, on the location of the element that we are searching for), the worst-case time complexity for the operations is logarithmic. In contrast, the list implementation of queues is constant *amortised* cost: we have to perform linear-time normalisation in the worst case, but most of the time the constant-time consing and unconsing operations are sufficient. The difference is that the second implementation has two modes of operation, one of which is more expensive but occurs rarely; the first implementation has just normal binary tree lookup which might take longer or shorter depending not on the structure of the collection, but on the element we are looking for. The binary tree version is therefore logarithmic, while the list implementation is constant amortised: while we might need to perform expensive normalisation, this "evens out" over the lifetime of the collection. That said, if some performance guarantees are required (e.g. in real-time critical systems), it may be worthwhile to give up constant amortised cost for a more consistent runtime that

> doesn't spike at hard-to-predict intervals.

2. The traditional way to implement queues uses a fixed-length array. Two indices into the array indicate the start and end of the queue, which wraps around from the end of the array to the start. How appropriate is such a data structure for implementing breadth-first search?

> While arrays allow for constant-time access to any element (including the front and end of the queue), its length is fixed: we would have a limit on what size tree we can traverse, or need to initialise a very large array that might end up being unnecessary for our purposes.

3. Why is iterative deepening inappropriate if $b \approx 1$, where b is the branching factor? What search strategy would make more sense in this case?

> A tree with branching factor $b \approx 1$ is almost like a list with one element at every depth. Given that iterative deepening is appropriate when the number of nodes at each level increases exponentially, it would be wasteful for a linear structure such as a list. Normal depth-first or breadth-first search is suitable.

4. An interesting variation on the data structures seen in the lectures is a *deque*, or *double-ended queue* [1]. A deque supports efficient addition and removal of elements on both ends (either its front or back). Suggest a suitable implementation of deques, justifying your decision.

> Double-ended queues must support enqueuing and dequeuing operations on both ends, so we need to find an implementation that gives constant-time access to both the "front" and "back" of the deque. Conveniently, our double list implementation for queues would work nicely for deques as well, as we can cons to and uncons from both halves of the list. The only difference is the normalisation algorithm: as we can get elements from both ends of the deque, we cannot just move every element to the first list and leave the second one empty. The alternative is to evenly split the normalised list between the two halves, so we always have elements we can access in constant time on both ends.

10.2. Exercises

5. Mathematical sets can be treated as an abstract data type for an unordered collection of unique elements. One approach we may take is to represent a set as an *ordered list* without duplicates: $\{5, 3, 8, 1, 18, 9\}$ would become the OCaml list `[1;3;5;8;9;18]`. Code the set operations of membership test, subset test, union and intersection using this ordered-list representation. Remember that you can assume the ordering invariant for the inputs (and thereby make your functions more efficient), and your output should maintain this invariant.

> The membership test is similar to the usual `member` function for lists, except we can "short-circuit" if the first element is less than the head of the list. Given that the lists are ordered (here we assume an increasing order), the head of the list must be its smallest element; if we ask if an element smaller than the smallest element is in the list, we can say "no"

[1] Deque is usually pronounced "deck", which is convenient because a deck of cards is a good example of a double-ended queue – computer scientists are often way too clever with naming things.

right away. Hence we can add the `y < x` constraint as a conjunct to the recursive call, so we stop as soon as an element is found or if the element cannot be in the list.

```
let rec member x = function
  | []    -> false
  | y::ys -> (x = y) || (y < x && member x ys)
```

An empty set is a subset of anything, and nothing is a subset of the empty set. The general case checks the heads of the lists and makes the appropriate recursive call. We make use of the ordering constraint by comparing the heads: if `x < y`, the first list contains an element that cannot be in the second list, so we can short-circuit.

```
let rec subset xs ys = match xs, ys with
  | [],    _     -> true
  | _ ,    []    -> false
  | x::xs, y::ys -> (y <= x) && if x = y then subset xs ys
                                         else subset (x::xs) ys
```

Set union on ordered lists eliminates all the membership tests, as we can determine the elements contained by the union by simple comparison. The same can be done with set intersection as well.

```
let rec union xs ys = match xs, ys with
  | [],    ys    -> ys
  | xs,    []    -> xs
  | 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

let rec inter xs ys = match xs, ys with
  | x::xs, y::ys ->
          if x < y then inter xs (y::ys)
     else if x > y then inter (x::xs) ys
     else               x :: inter xs ys
  | _, _ -> []
```

10.3. Optional questions

6. Implement deques as an abstract data type in OCaml (including a type declaration and suitable operations). Estimate the amortised complexity of your solution.

As explained before, we can use the same double list implementation we gave for queues, with similar operations. The only function that needs changing is normalisation, as we want to evenly divide the elements between the two lists when either of the lists becomes empty. To accomplish this, we need to find the halfway point of the list where we can perform a split, which is a linear operation. One slight efficiency improvement we can make is a common trick when dealing with sized collections: instead of recalculating the length every time we need it, we keep track of it in the data structure itself as a separate field. Then, a `length` operation simply returns this known value, and we don't need to find the length of the list in the normalisation function. There might be some extra bookkeeping required in the definitions below, but the point of abstract data types is that the actual implementation is hidden from view.

We make use of the helper function `split` which splits a list into two at a given index. Here we give a definition from scratch, but the result should have the property that `split l i = (take l i, drop l i)`. In `norm` we have two cases which need to be handled: when either the first or second list is empty. In those cases, we split the other list in half, and reverse one half based on which end of the deque we are looking at. The other operations are all straightforward, though we need to increment or decrement the length if we enqueue or dequeue and element.

```
let rec split xs n = match xs, n with
  | [],    _ -> ([], [])
  | x::xs, 0 -> ([], x::xs)
  | x::xs, i -> match split xs (i-1) with
                  (ys, zs) -> (x::ys, zs)

type 'a deque = D of 'a list * 'a list * int

let norm = function
  | D ([], tls, l) -> (match split tls (l / 2) with
                       (h1, h2) -> D (List.rev h2, h1, l))
  | D (hds, [], l) -> (match split hds (l / 2) with
                       (h1, h2) -> D (h1, List.rev h2, l))
  | d -> d

let dnull = function (D ([],[],0)) -> true | _ -> false
let enqFront (D (hds, tls, l), x) = norm (D (x::hds, tls, l+1))
let enqEnd   (D (hds, tls, l), x) = norm (D (hds, x::tls, l+1))
let deqFront (D (x::hds, tls, l)) = norm (D (hds, tls, l-1))
let deqEnd   (D (hds, x::tls, l)) = norm (D (hds, tls, l-1))
let length   (D (_, _, l)) = l
```

11. Elements of procedural programming

11.1. Conceptual questions

1. What is the effect of `while (C1; B) do C2 done`? Where would such a formulation be useful?

> The sequencing operator `;` executes the commands left-to-right, and ignores the return value of all statements apart from the last one, which becomes the return value of the whole sequencing. Hence `C1` is executed on every iteration, but its value is replaced by the value of the Boolean condition `B`. The body of the while loop, `C2`, is executed on every iteration as well. In addition, `C1` is executed before the Boolean condition, so its side-effects are performed even if `B` evaluates to `false` right away. Therefore the expression is equivalent to `C1; while B do (C2; C1) done`. This formulation would be useful whenever we want to repeat instructions at least once; for example, making `C2` the empty instruction would result in a construct equivalent to a do-while loop, which checks the Boolean condition after the first iteration.

11.2. Exercises

2. Comment, with examples, on the differences between an `int ref list` and an `int list ref`. How would you convert between the two? *Hint*: `(!) : 'a ref -> 'a` is a function.

> The first one is a list of references, e.g. `[ref 1, ref 2, ref 3] : int ref list`. Each reference can be individually assigned, but the list itself is immutable. The second one is a reference to a list, e.g. `ref [1,2,3] : int list ref`. We cannot change the elements individually, but we can change the whole list.
>
> Converting from a list of references to a reference to a list would involve dereferencing the list elements, and creating a reference to the resulting list. Since `(!) : 'a ref -> 'a` is just a function, we can map it over elements of a list:
>
> ```
> let reflistToListref rs = ref (List.map (!) rs)
> ```
>
> The other way is similar, except we dereference first and map `ref` after:
>
> ```
> let listrefToReflist lr = List.map ref (!lr)
> ```
>
> Of course, working with state and mutability means that the behaviour of these functions may not always be predictable.

3. Write a version of the efficient `power` function (Page 10) using `while` instead of recursion.

> Instead of tail-recursion, we use imperative iteration with a `while`-loop. We initialise an accumulator reference `accr` and references for our two inputs to allow mutability. What was an argument to the recursive call before becomes a reference assignment.
>
> ```
> let power x n =
> ```

```
  let accr = ref 1.0
  and xr = ref x
  and nr = ref n
   in while !nr <> 1 do
        if !nr mod 2 = 0
        then (xr := !xr *. !xr; nr := !nr / 2)
        else (accr := !accr *. !xr;
              xr := !xr *. !xr; nr := !nr / 2)
      done;
      !accr *. !xr
```

4. Write a function to exchange the values of two references, `xr` and `yr`.

   While there are some type-specific hacks to perform the exchange "in-place", we generally need a temporary reference to avoid overwriting a value that we need to use later.

   ```
   let swap xr yr = let v = !xr in xr := !yr; yr := v
   ```

5. Arrays of multiple dimensions are represented in OCaml by arrays of arrays. Write functions to (a) create an $n \times n$ identity matrix, given n, and (b) to transpose an $m \times n$ matrix. Identity matrices have the following form:

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

   In both cases we make use of the `Array.init : int -> (int -> 'a) -> 'a array` function, which initialises an array with values calculated from the array index using a higher-order function argument. Matrices can be created by nesting these functions together, and using the current matrix index in the function argument of the inner `init` function. For example, an element of an identity matrix is 1 if its row and column index is equal, and 0 otherwise.

   ```
   let ident n = Array.init n (fun i ->
                 Array.init n (fun j ->
                   if i = j then 1 else 0))
   ```

   There are several approaches to transpose a matrix: we can use references to perform the transposition in-place, but `Array.init` allows us to use a more elegant, functional solution. Matrices are represented as an array of rows, so getting a single row of the matrix is just indexing into the outer matrix. Selecting an individual column c is harder, since we

need to return the c^{th} element of each of the rows. However, if we have a function that can do that, we can transpose the matrix by selecting the columns of the matrix and turning them into rows (by simply putting them in an array).

The helper function below uses `init` to return a given column of the input matrix. Given an $n \times m$ matrix, it returns an array of size n (which is just the length of the outer array), with its elements being the c^{th} element of the individual rows. "Looping" through the rows is done using `init` and `get`.

```
let getCol c arr = Array.(init (length arr)
                               (fun i -> get (get arr i) c))
```

The actual `transpose` function now uses `getCol` to create a new array of columns. The size of the outer array is the length of any of the inner arrays, and its r^{th} element is just the r^{th} column of the input matrix.

```
let transpose arr = Array.(init (length (get arr 0))
                                (fun r -> getCol r arr))
```