Foundations of Computer Science

Supervision 4

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
```

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?

9.2. Exercises

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

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.

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`.)

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]`, ...

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`.

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>
```

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)))
```

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".

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.
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?
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?
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.

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:

[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.

$\{5, 3, 8, 1, 18, 9\}$ 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.

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.

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?

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.
3. Write a version of the efficient `power` function (Page 10) using `while` instead of recursion.
4. Write a function to exchange the values of two references, `xr` and `yr`.
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:

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