COMPUTER SCIENCE TRIPOS Part IA – 2014 – Paper 1

2 Foundations of Computer Science (LCP)

This question has been translated from Standard ML to OCaml

lists, queues, complexity

(*a*) Write brief notes on the queue data structure and how it can be implemented efficiently in OCaml. In a precise sense, what is the cost of the main queue operations? (It is not required to present OCaml code.) [6 marks]

Answer: A queue represents a sequence, allowing elements to be taken from the head and added to the tail. Lists can implement queues, but append is a poor means of adding elements to the tail. The solution is to represent a queue by a pair of lists, where

$$([x_1; x_2, \ldots, x_m], [y_1, y_2, \ldots, y_n])$$

represents the queue $x_1 x_2 \ldots x_m y_n \ldots y_1$.

The front part of the queue is stored in order, and the rear part is stored in reverse order. We add elements to the rear part using cons, since this list is kept reversed; this takes constant time. To remove an element, we look at the front part, which normally takes constant time, since this list is stored in order. When the last element of the front part is removed, we reverse the rear part, which becomes the new front part.

Queue operations take $O(1)$ time when *amortized*: averaged over the lifetime of a queue. Even for the worst possible execution, the average cost per operation is constant.

---

lists, exceptions, programming

(*b*) Run-length encoding is a way of compressing a list in which certain elements are repeated many times in a row. For example, a list of the form $[a; a; a; b; a; a]$ is encoded as $[(3, a); (1, b); (2, a)]$. Write a polymorphic function `rl_encode` to perform this encoding. What is the type of `rl_encode`? [6 marks]

Answer:

```
let rec rl_encode = function
| [] -> []
| x::xs ->
    let rec code n = function
    | [] -> [(n, x)]
    | y::ys ->
        if x = y then
          code (n + 1) ys
        else
          (n, x) :: rl_encode (y::ys)
    in
      code 1 xs
```

The type is `'a list -> (int * 'a) list`. The `code` function can also be expressed with guard clauses:

```
    let rec code n = function
    | [] -> [(n, x)]
    | y::ys when x = y -> code (n + 1) ys
    | ys -> (n, x) :: rl_encode ys
```

---

lists, programming

(*c*) The simple task of testing whether two lists are equal can be generalised to allow

a certain number of errors. We consider three forms of error:

- *element mismatch*, as in [1; 2; 3] versus [1; 9; 3] or [1; 2; 3] versus [0; 2; 3]
- *left deletion*, as in [1; 3] versus [1; 2; 3] or [1; 2] versus [1; 2; 3]
- *right deletion*, as in [1; 2; 3] versus [1; 3] or [1; 2; 3] versus [1; 2]

Write a function `genEquals n xs ys` that returns `true` if the two lists `xs` and `ys` are equal with no more than `n` errors, and otherwise `false`. You may assume that n is a non-negative integer. [8 marks]

Answer:

```
let rec genEquals n xs ys =
  match xs, ys with
  | ([], []) -> true
  | ([], y::ys) -> n > 0 && genEquals (n - 1) [] ys
  | (x::xs, []) -> n > 0 && genEquals (n - 1) xs []
  | ((x::xs), (y::ys)) ->
      if x = y then genEquals n xs ys
      else n > 0 && (genEquals (n - 1) xs ys
                     || genEquals (n - 1) (x::xs) ys
                     || genEquals (n - 1) xs (y::ys))
```

All OCaml code must be explained clearly and should be free of needless complexity.