Sample Exam; Solutions

Michael P. Fourman

February 2, 2010

1 Introduction

This document contains solutions to the sample questions given in Lecture Note 10

Short Question *5 marks*

Give the responses of the ML system to the following sequence of declarations

```
val a = 1;

val b = 2;

fun f a = a + b;

val b = 3;

f b;
```

the responses of the system are as follows:

```
val a = 1 : int
val b = 2 : int
val f = fn : int -> int
val b = 3 : int
val it = 5 : int
```

The point here is to check that you understand, and can apply, the scoping rules, applied to the bindings of `a` and `b`. The exam on Thursday may include `let` and `local` declarations, in a similar question.

1. **Long Question** *10 marks*

The following datatype can be used to represent trees whose nodes can have an arbitrary number of children.

```
datatype 'a Tree = Tree of 'a * 'a Tree list
```

(a) What tree does the following expression denote (i.e draw a picture):

Tree(1, [Tree(2, [ ]), Tree(3, [Tree(4,[ ])])])

(b) Define a function to calculate the number of leaves in such a tree.

```
fun sum []       = 0
  | sum (h :: t) = h + sum t

fun leaves (Tree(_,[])) = 1
  | leaves (Tree(_,ts)) = sum (map leaves ts)
```

(c) We can assign a level to each node in a tree as follows. The node at the root is at level 1. Its children are at level 2. Their children are at level 3 and so on.

Suppose we are interested in trees where an internal node at level n always has exactly n children. Define a function `check : 'a Tree ->bool` that checks whether a given tree has this property.

The recursion is not straightforward: to check the property for a tree, we must check a slightly *different* property for its subtrees. We therefor introduce an auxiliary function, `checkk`, with an extra parameter; `checkk k` checks that the appropriate property holds for a subtree rooted at level `k`:

```
fun length [] = 0
  | length (_::t) = 1 + length t

fun andl [] = true (* and over a list of booleans *)
  | andl (h :: t) = h andalso andl t

fun checkk k (Tree(_,[])) = true (* nothing to check for a leaf
  | checkk k (Tree(_,ts)) = ((length ts) = k)
```

```
                                          (* check there are k children *)
                                          andalso
                                          andl (map (checkk (k+1)) ts)
                                          (* subtrees are at level (k+1) *)

fun check  t           = checkk 1 t
```

2. **Long Question** *10 marks*

The `EQueue` signature is like the signature `Queue`, but is extended with an additional operation multiple enqueue, `menq:(Item  list * Queue) -> Queue`, intended to add a number of items (in an arbitrary order) to the queue in a single operation.

```
signature EQueue =
sig
    type Item
    type Queue

    val empty : Queue
    val enq : (Item * Queue) -> Queue
    val deq : Queue -> (Item * Queue)
    val menq: (Item list * Queue) -> Queue
end
```

An implementation of a **stack**, including this operation, uses the type declaration

```
type Queue = Item list list
```

the operations `empty` and `menq` are implemented as follows:

```
val empty = []

fun menq(items, q) = items :: q
```

(a) Complete the following declarations of the functions `enq` and `deq` for this implementation

```
fun enq(item,       []) = [[item]]
  | enq(item, (h :: t)) = (item :: h) :: t
    (* or, alternatively, [item] :: h :: t *)

fun deq((h :: t) :: r)  = (h, t :: r)
  | deq([] :: r)        = deq r
  | deq []              = raise Deq
```

The point here is to take care with the types. Since a stack is being represented as a list of lists, we need to make a list, `[[item]]`, whose only member is the singleton list, `[item]`, to represent a stack with one entry. When adding an item to a non-empty stack, we have a choice: we can either add the item to the list at the head of the list of lists, or we can form a new singleton list and add this to the list of lists.

(b) What is the complexity of the three operations

i. `enq`, $O(1)$

ii. `deq`, $O(1)$

iii. `menq` $O(1)$

for this implementation?

Notice that, for a conventional stack implementation we would have to implement `menq` using multiple calls of `enq`. The complexity would be $O(n)$, where n is the number of items being added in one go.

3. **Long Question** *10 marks*

An implementation of sets of integers is designed to represent a set by a list without repetitions, **kept in increasing order**. Here is the function `union : Set*Set -> Set` from this implementation

```
fun union(a, []) = a
  | union([], b) = b
  | union(ah :: at, bh :: bt) =
          if ah < bh then ah :: union(at, bh :: bt)
          else if ah = bh then ah :: union(at, bt)
          else bh :: union(ah :: at, bt)
```

(a) What is the complexity of this implementation of `union`?

$O(n)$, where n is the sum of the sizes of the sets; there is at most one recursive call for each of these elements.

(b) Give an implementation of the operation `insert : (int*Set) ->Set` compatible with this representation.

```
fun insert (x,[]) = [x]
  | insert (x,h :: t) = if x < h then x :: h :: t
                        else if h < x then h :: insert(x,t)
                        else (* x = h *) h :: t
```

This is book-work: a similar definition was given in the notes to implement a priority queue.

(c) Give an $O(n)$ implementation of the operation `intersect : Set*Set -> Set`, compatible with this representation.

```
fun intersect(a, []) = []
  | intersect([], b) = []
  | intersect(ah :: at, bh :: bt) =
          if ah < bh then intersect(at, bh :: bt)
          else if ah = bh then ah :: intersect(at, bt)
          else intersect(ah :: at, bt)
```

This follows the pattern given in the declaration of `union`.