Functional Programming and Specification
Practical 3

This is an assessed practical exercise, to be completed by 4pm on Monday 21st March. In contrast to previous exercises, the mark will contribute to the overall mark for the course, and what you submit must be your own work.

You may submit your solution electronically using the DICE submit command, like so:

For UG3 students: submit cs3 fps-3 3 myfile1 myfile2 ... myfilen
For UG4/MSc students: submit msc fps-4 3 myfile1 myfile2 ... myfilen

or on paper to the ITO on level 4 of Appleton Tower. You may want to use Extended Moscow ML to check your solution to Exercise 1 for syntax and type errors before submitting it, but that is not required. If you do this, please submit at least this part using submit.

Exercise 1: Specification in EML

Give an EML signature specifying first-in first-out queues with the following constant and functions:

- empty: the empty queue
- isempty: tests whether or not a queue is empty, returning a boolean result
- insert: add an element to the back of the queue, returning the new queue
- front: return the element at the front of the queue
- remove: remove an element from the front of the queue, returning the new queue
- join: join two queues, the second behind the first, returning the resulting queue

Make the specification as abstract and high-level as possible; for instance, it should not specify a particular representation of queues in terms of (say) lists. Mention any choices made in the interpretation of the above informal requirements. It is not necessary to specify the behaviour of front and remove when applied to the empty queue.

Exercise 2: Proof in EML

Consider the following EML functor:

signature PO =
sig
type t
val le : t * t -> bool
axiom Forall x => le(x,x)
axiom Forall (x,y,z) => le(x,y) andalso le(y,z) implies le(x,z)
axiom Forall (x,y) => le(x,y) andalso le(y,x) implies x==y
axiom Forall (x,y) => le(x,y) orelse le(y,x)
end

signature SORT =
sig
structure OBJ : PO
local
val cmp: OBJ.t * OBJ.t -> int
axiom For all a => cmp(a,a) = 1
axiom For all (a,b) => a=/=b implies cmp(a,b) = 0
val count : 'a * 'a list -> int
axiom For all a => count(a,nil) = 0
axiom For all (a,b,1) => count(a,b::1) = cmp(a,b) + count(a,1)
val permutation : 'a list * 'a list -> bool
axiom For all (l,l') =>
permuation(l,l') = (Forall x => count(x,l) = count(x,l'))
val ordered : OBJ.t list -> bool
axiom ... ... ...
in
val sort : OBJ.t list -> OBJ.t list
axiom Forall l => permutation(l,sort l)
axiom Forall l => ordered(sort l)
end
end

functor Sort(structure X : PO) :> SORT where type OBJ.t=X.t =
struct
structure OBJ = X
datatype heap = empty | node of heap * OBJ.t * heap
fun insert(x,empty) = node(empty,x,empty)
  | insert(x,node(h,y,h')) =
    if OBJ.le(x,y) then node(insert(y,h'),x,h)
    else node(insert(x,h'),y,h)
fun createheap nil = empty
  | createheap(a::l) = insert(a,createheap l)
fun merge(l,nil) = l
  | merge(nil,l') = l'
  | merge(a::l,a'::l') = if OBJ.le(a,a') then a::merge(l,a'::l')
    else a'::merge(a::l,l')
fun extractheap empty = nil
  | extractheap(node(h,x,h')) = x::merge(extractheap h, extractheap h')
fun sort l = extractheap(createheap l)
end

This sorting algorithm uses an intermediate data structure called a heap, which is a labelled binary tree in which the label of each node is the smallest value in the subtree having that node as root. Inserting a value into a heap preserves this property. To sort a list, we successively insert the values into a heap (initially empty) and then we extract a sorted list from the resulting heap. The insert function uses a simple trick to ensure that the tree remains balanced.

Give a proof that the code for sort satisfies the axiom which requires its output to be a permutation of its input. (Hint: you will need to prove lemmas involving merge, extractheap, createheap and insert.) Your proof should be convincing but need not give all details of every case.

⋆

If you want to learn more:

- Write EML signatures specifying some of the Moscow ML library modules.
- Fill in the specification of ordered in SORT. Then give a proof that the code for sort satisfies the axiom which requires its output to be ordered.
- Find out about a computer-assisted theorem proving system (examples: HOL, PVS, Lego). Translate the sorting example into the syntax of that system, then repeat your proofs in the system.