UNIVERSITY OF EDINBURGH

FACULTY OF SCIENCE AND ENGINEERING

MASTER OF SCIENCE IN COMPUTER SCIENCE

APPLICATIVE PROGRAMMING AND SPECIFICATION

Wednesday 26 April 2000

2:00 to 3:30

Examiners:

I. Watson (External)

R.N. Ibbett (Chairman)

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.

All questions carry equal weight.

1. Sets of integers may be represented using lists of "intervals": the interval $[a, b]$ for $a \leq b$ represents the set of integers between a and b inclusive, where $[a, a]$ represents the set $\{a\}$. A list containing several intervals represents the union of the sets represented by the intervals. If the intervals do not overlap or "touch" then this representation is space-efficient; if they are kept in ascending order then manipulation of sets can be made time-efficient. We call a list of intervals *valid* if it satisfies these conditions.

   Here is an example of sets represented this way, and four non-examples. (We represent the interval $[a, b]$ in ML as the pair `(`a`,`b`):int*int`.)

   `[(1,3),(7,7),(10,11)]` is valid and represents $\{1, 2, 3, 7, 10, 11\}$.
   `[(2,1),(5,6)]` is invalid: $[2, 1]$ isn't a valid interval.
   `[(1,4),(3,6)]` is invalid: intervals overlap.
   `[(1,4),(5,6)]` is invalid: intervals "touch".
   `[(3,4),(1,1)]` is invalid: intervals aren't in ascending order.

   (a) Implement the following using this representation. You may assume that sets provided as input are valid in the sense explained above, and the sets that you produce as results must be valid.

      i. Implement *set membership*. [*4 marks*]

      ii. Implement *insertion* of an integer into a set. [*10 marks*]

      iii. Implement *deletion* of an integer from a set. [*7 marks*]

   (b) The list-of-intervals representation of sets may be used for sets over domains other than integers. Suppose you were asked to produce an SML functor that implements sets over any given linearly-ordered element type, containing suitably modified versions of your functions in (a) above. What would the input signature of this functor be? (Concentrate on discrete domains, disregarding domains such as the real numbers.) [*4 marks*]

2. (a) Consider the following two implementations of queues.

```
structure Q1 =
  struct
    exception Emptyq
    type 'a queue = 'a list
    val emptyq = []
    fun enq(x,q) = x::q
    fun front [] = raise Emptyq
      | front [x] = x
      | front(x::q) = front q
  end

structure Q2 =
  struct
    exception Emptyq
    type 'a queue = 'a list
    val emptyq = []
    fun enq(x,q) = q@[x]
    fun front [] = raise Emptyq
      | front(x::q) = x
  end
```

Describe the response of the ML system to the following:

```
Q1.front(Q2.enq(1,Q2.enq(2,Q1.emptyq)))
```

What is the problem with this result? How might one reformulate the definitions of `Q1` and/or `Q2` to ensure that this kind of problem does not arise? Give the new version of `Q1` and/or `Q2` and describe the response of the ML system to the above line in that case. [*10 marks*]

(b) Consider the following:

```
signature QUEUESIG =
  sig
    exception Emptyq
    type queue
    type item
    val enq : item * queue -> queue
    val emptyq : queue
    val isempty : queue -> bool
    val front : queue -> item
    val rest : queue -> queue
  end

signature SETSIG =
  sig
    type elem
    type set
    val singleton : elem -> set
    val emptyset : set
    val union : set * set -> set
    val choose : set -> elem
    val member : elem * set -> bool
  end

functor F(structure Q : QUEUESIG and S : SETSIG) =
  struct
    structure Q = Q
    structure S = S
    fun setify q = if Q.isempty q then S.emptyset
                   else S.union(S.singleton(Q.front q),
                                setify(Q.rest q))
  end
```

How would the ML system respond to this code? Use a concrete instantiation of `F` to illustrates what would go wrong if this were permitted. How would you correct the problem? [*15 marks*]

3. A *priority queue* is a data structure which allows elements to be added in arbitrary order, but the smallest element is always removed first.

   Consider the following two simple implementations of priority queues represented using lists, with a function *decant* that initializes the queue to contain all the elements in a given list:

   Elements unordered, remove searches for the minimum

```
fun add(x,q) = x::q
fun remove(x::nil) = x
  | remove(x::y::l) = min(x,remove(y::l))
fun decant l = l
```

   Elements kept in order, remove takes the first

```
fun add(x,nil) = [x]
  | add(x,y::l) = if x<y then x::y::l
                  else y::add(x,l)
fun remove(x::l) = x
fun decant nil = nil
  | decant(x::l) = add(x,decant l)
```

   (a) Specify *remove* and *add*. The properties you give should be abstract enough to be satisfied by both of the implementations above. [*5 marks*]

   (b) Pick one of the above implementations and prove that it satisfies this specification. [*10 marks*]

   (c) Prove that

   $$l \neq \texttt{nil} \Rightarrow \texttt{remove(decant } l\texttt{) = remove'(decant' } l\texttt{)}$$

   where `remove` and `decant` refers to the first implementation and `remove'` and `decant'` refers to the second. [*10 marks*]