UNIVERSITY OF EDINBURGH

FACULTY OF SCIENCE AND ENGINEERING

COMPUTER SCIENCE THREE

FUNCTIONAL PROGRAMMING AND SPECIFICATION

Wednesday 4 September 2002

09:30 to 11:00

Examiners:

Prof. K. Turner (External)

Prof. C. Stirling (Chairman)

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.

All questions carry equal weight.

1. Consider the following datatype of "fat lists":

```
datatype 'a fatlist = nil
                    | cons of 'a * 'a fatlist
                    | append of 'a fatlist * 'a fatlist
```

(a) Define variants of the list functions `length`, `reverse` and `map` for fat lists. Define a function `null : 'a fatlist -> bool` which tells if a fat list is empty (contains no elements) or not. [*7 marks*]

(b) Define a function `fold : ('a * 'b -> 'b) -> 'b -> 'a fatlist -> 'b` that is analogous to `foldr` on lists. Give a function `f` and a value `v` such that `fold f v l` is `null l` for all fat lists `l`. Give a function `g` and a value `w` such that `fold g w l` is `length l` for all fat lists `l`. [*6 marks*]

(c) Define a function `slim : 'a fatlist -> 'a list` which converts a fat list to an ordinary list of its elements in the same order. Give a function `f` (depending on `g`) and a value `v` such that `slim(fold f v l) = slim(map g l)` for all fat lists `l`. [*6 marks*]

(d) Is there a function `f` (depending on `g`) and a value `v` such that `fold f v l = map g l` for all fat lists `l`? Give a brief justification of your answer. [*6 marks*]

2. Consider the following structure representing *bags* (also called *multisets*). A bag is a collection of elements where the order in which elements are inserted is unimportant; only the number of times that each element has been inserted is significant.

```
structure Bag =
  struct
    type 'a bag = 'a -> int
    fun emptybag x = 0
    fun addbag(x,b) = fn y => if x=y then b x + 1 else b y
    fun count(x,b) = b x
  end
```

(a) In the absence of a programmer-defined signature, Standard ML will calculate the *principal signature* for a structure. What is the principal signature for the `Bag` structure? [*5 marks*]

(b) For encapsulation purposes a Standard ML programmer usually provides a signature for every structure. Provide such a signature for the `Bag` structure. [*5 marks*]

(c) Provide an alternative `Bag` structure which matches your signature from (b) above but uses lists, rather than functions, to represent bags. [*5 marks*]

(d) The `bagunion` function unites two bags to produce a bag with the following property:

```
forall (x,b1,b2) => count(x,bagunion(b1,b2)) = count(x,b1) + count(x,b2)
```

Extend either version of the `Bag` structure with the `bagunion` function. [*5 marks*]

(e) The `baglistunion` function unites a list of bags. Extend either version of the `Bag` structure with the `baglistunion` function. [*5 marks*]

3. The following defines an expression as either a variable or the application of an operator (represented as a string) to a list of expressions, with constants regarded as 0-ary operators.

```
datatype exp = app of string * exp list | var of string
```

Now consider the following program for matching one expression against another, producing a substitution. (The intention is that, for example,

```
match( app("f",[var "x",     app("g",[]), var "x"]),
       app("f",[app("h",[]), app("g",[]), app("h",[])]) )
```

is the substitution taking `"x"` to `app("h",[])`.)

```
fun matchexp(app(c,l), app(c',l'), s) =
        if c=c' then matchl(l,l',s) else none
  | matchexp(var x, exp, s) =
        if member(x,s) then if lookup(x,s)=exp then s else none
        else add(x,exp,s)
  | matchexp(app(_,_), var _, _) = none
and matchl(nil,nil,s) = s
  | matchl(a::l,a'::l',s) = matchl(l,l',matchexp(a,a',s))
  | matchl(_,_,s) = none        (* lists of different lengths *)
fun match(e,e') = matchexp(e,e',empty)
```

This assumes a type subst of substitutions with

```
empty : subst
member : string * subst -> bool
lookup : string * subst -> exp
add : string * exp * subst -> subst
none : subst
```

where `add(x,e,none)=none` and `lookup(x,none)` raises an exception.

(a) Write Extended ML axioms specifying as abstractly as possible the behaviour of match. You may wish to use the following auxiliary function that replaces the variables in an expression according to a substitution:

```
fun apply(s,var x) = lookup(x,s)
  | apply(s,app(c,l)) = app(c,map (fn exp => apply(s,exp)) l)
```

[10 marks]

QUESTION CONTINUES ON NEXT PAGE

(b) Consider the following implementation of substitutions:

```
datatype subst = some of (string * exp) list | none
val empty = some []
fun member(x,none) = false
  | member(x,some []) = false
  | member(x,some((y,_)::l)) = (x=y) orelse member(x,some l)
exception Fail
fun lookup(x,none) = raise Fail
  | lookup(x,some []) = var x
  | lookup(x,some((y,exp)::l)) =
        if x=y then exp else lookup(x,some l)
fun add(x,exp,none) = none
  | add(x,exp,some l) = some((x,exp)::l)
```

Show that

```
forall (l,l',x) =>
        lookup(x,some(l@l')) == apply(some l',lookup(x,some l))
```

holds if `l` associates variables with variable-free expressions. The proof need not be completely formal. *[15 marks]*