

CSL 765: Introduction to (Logic and Functional) Programming

I semester 2011-12

Sat 03 Sep 2011

11:00-12:00

Minor I

VI LT1

Max Marks 40

1. Please answer in the space provided on the question paper. The other sheets are only for rough work and will not be collected.
2. You may use any paper-based material including your class notes and any other text books.
3. You are not allowed to share reference material or rough pages during the exam.
4. You are not allowed to bring into the exam hall an electronic gadgets such as computers, mobile phones or calculators.
5. Please keep your identity card with you. You may be asked for it at any time for verification.

1. ((6+4) = 10 marks) Consider the following insertion sort program in Standard ML:

```
(* R is assumed to be a total ordering relation *)
fun insertSort R [] = []
  | insertSort R (h::t) =
    let fun insert R [] x = [x]
        | insert R (h::t) x = if R (x, h) then x::(h::t) else h::(insert R t x)
    in val rest = insertSort R t
      in insert R rest h
    end
```

Rewrite this sort program (without changing the algorithm) in a purely functional style in ML such that every recursive function you use is tail-recursive.

Solution.

```
(* Tail Recursive version *)
local (* insert a new element into a sorted list tail-recursively *)
  local
    fun ins_tr R x L M =
      let val Mr = rev M
      in (case L of
          nil => Mr@[x]
        | h::t => if R(x, h) then Mr@(x::L)
                  else ins_tr R x t (h::M)
        )
      end
    in fun insertT R x L = ins_tr R x L []
      end

    fun insertSortFun_tr R [] M = M
      | insertSortFun_tr R (h::t) M = insertSortFun_tr R t (insertT R h M)
  in fun insertSortFunT R L = insertSortFun_tr R L []
  end
```

(* Moral of the story: Tail recursion may not always be more efficient *)

2. ((4+5) + 6 = 15 marks)

- (a) Let 'a be any type and let $f, g : 'a \rightarrow 'a$ be any binary functions on the type 'a such that g has an identity element i . Assume given a $k \times l$ matrix X and another $m \times n$ matrix Y each of elements of type 'a. X is said to be *compatible* with Y if and only if $l = m$. If X is compatible with Y , their *convolution* is defined as a $k \times n$ matrix Z (denoted $Z = X \circledast Y$) such that each element $z_{r,c}$ in the matrix Z where $1 \leq r \leq k$ and $1 \leq c \leq n$ is defined as

$$z_{r,c} = g(\cdots g(g(f(x_{r,1}, y_{1,c}), i), f(x_{r,2}, y_{2,c})), f(x_{r,3}, y_{3,c})), \cdots, f(x_{r,m}, y_{m,c}))$$

$x_{r,j}$ and $y_{j,c}$ are elements from X and Y respectively.

Assume $X, Y, Z : 'a \text{ list list}$ are row-wise representations of the matrices.

- Define a *higher-order function* `compat` : `'a list list * 'a list list -> bool` which determines whether X is compatible with Y .
- Define a *higher-order function* `conv` : `((('a * 'a -> 'a) * ('a * 'a -> 'a) * 'a) -> ('a list list * 'a list list) -> 'a list list` which computes the convolution of two compatible matrices.

Solution.

```
exception emptyMatrix;
exception incompatible;
```

```
i. fun is2DMatrix []          = (true, 0, 0)
    | is2DMatrix [H]         = (true, 1, length H)
    | is2DMatrix (H::TL) =
      let val (b, r, c) = is2DMatrix TL
      in (b andalso (length H = c), r+1, c)
      end;
```

```
fun compatible ((X: 'a list list), (Y: 'a list list)) =
  let val (x, r, k) = is2DMatrix X
      and (y, m, c) = is2DMatrix Y
  in x andalso y andalso (k=m)
  end;
```

```
ii. local
  fun trans [] = []
    | trans ([::TL) = []
    | trans (LL as H::TL) = (map hd LL)::(trans (map tl LL));
```

```
(* Assuming sum is left associative with id as identity *)
fun lscalarprod (prod, sum, id:'a) (s:'a) [] [] = s
  | lscalarprod (prod, sum, id) s [] _ = raise incompatible
  | lscalarprod (prod, sum, id) s _ [] = raise incompatible
  | lscalarprod (prod, sum, id) s ((g:'a)::S) ((h:'a)::T) =
    lscalarprod (prod, sum, id) (sum(s, prod(g,h):'a)) S T
```

```
fun lsuperlscalarprod (prod, sum, id) ([], M) = []
  | lsuperlscalarprod (prod, sum, id) ((G::L), M) =
    (map (lscalarprod (prod, sum, id) id G) M)::
    (lsuperlscalarprod (prod, sum, id) (L, M))
```

```
in fun lconvolution (prod, sum, id) (X, Y) =
  if (null(X) orelse null(Y)) then raise emptyMatrix
  else if compatible (X, Y)
  then lsuperlscalarprod (prod, sum, id) (X, (trans Y))
  else raise incompatible
end;
```

```
val conv = lconvolution;
```

- (b) In the lectures we generated various progressions and ended up with the higher order functions

$$\begin{aligned}
 AGP4 \text{ bop } (a, d, n) &= \begin{cases} [] & \text{if } n \leq 0 \\ a :: (\text{map } (\text{curry2 } \text{bop } d)(AGP4 \text{ bop } (a, d, n - 1))) & \text{else} \end{cases} \\
 AP4 &= AGP4 \text{ op+} \\
 GP4 &= AGP4 \text{ op*} \\
 sumint &= foldl \text{ op } + \ 0 \\
 AS4(a, d, n) &= sumint(AP4(a, d, n)) \\
 GS4(a, d, n) &= sumint(GP4(a, d, n))
 \end{aligned}$$

where $AP4$ and $GP4$ generated the progressions. But if we are interested only in the sums of the various progressions and not the individual elements of the progressions, it is wasteful to first generate the sequences and then find the series sum. Instead it is better to compute the sum directly by a higher order function which generates each element and accumulates the sum. Define a higher-order function $AGS5 \text{ bop } (a, d, n)$ which directly computes the sum of the appropriate series of n elements starting from a with a common difference/ratio d directly.

Solution.

$$AGS5 \text{ bop } (a, d, n) = AGS5_tr \text{ bop}(a, d, n)0$$

where

$$AGS5_tr \text{ bop}(a, d, n)s = \begin{cases} s & \text{if } n = 0 \\ AGS5_tr \text{ bop}(a \text{ bop } d, d, n - 1)(s + a) & \text{else} \end{cases}$$

3. (4 + 4 = 8 marks)

Let $S \stackrel{df}{=} \lambda x \lambda y \lambda z [((x z) (y z))]$ and $K \stackrel{df}{=} \lambda x \lambda y [x]$. Prove that application is not associative by showing that $((S K) K) S$ and $(S (K K) S)$ yield distinct β -normal forms.

Solution.

$$\begin{array}{lcl}
 ((S K) K) S & & (S (K K) S) \\
 \equiv & & ((\lambda x y z [((x z) (y z))] (K K) S) \\
 \rightarrow_{\beta}^2 & & (\lambda z [((K z) (K z))] S) \\
 \equiv & & (\lambda z [((\lambda x y [x] z) (\lambda x y [x] z))] S) \\
 \rightarrow_{\beta}^2 & & (\lambda z [(\lambda y [z] \lambda y [z])] S) \\
 \rightarrow_{\beta} & & (\lambda z [z] S) \\
 \rightarrow_{\beta} & & S
 \end{array}
 \quad
 \begin{array}{lcl}
 & & \\
 \equiv & & ((\lambda x y z [((x z) (y z))] (K K) S) \\
 \rightarrow_{\beta} & & (\lambda y z [((K K) z) (y z))] S) \\
 \rightarrow_{\beta} & & \lambda z [((K K) z) (S z)] \\
 \equiv & & \lambda z [((\lambda x y [x] K) z) (S z)] \\
 \rightarrow_{\beta}^2 & & \lambda z [K (S z)] \\
 \rightarrow_{\beta}^2 & & \lambda z y [S z] \\
 \equiv_{\alpha} & & \lambda z y [(\lambda u v w [(u w) (v w)]) z] \\
 \rightarrow_{\beta} & & \lambda z y v w [(z w) (v w)]
 \end{array}$$

It is clear that the two terms yield distinct β -normal forms.

4. (6 + (2+2+2) = 12 marks) Consider the recursive datatype for booleans defined by

```
datatype boolean = t | f | ite of boolean * boolean * boolean
```

Intuitively **t** denotes *true*, **f** denotes *false* and **ite** is the *if-then-else* operator on booleans. Assuming that **t** and **f** are the normal forms of this data type,

- Define a complete set of rewrite rules which reduce any expression in the data type **bool** to its normal form.
- Define the functions **not**, **and** and **or** on elements of this datatype, so that they capture the intuitive meanings of these operators.

Solution.

- The following are the defining equations. For all $x, y \in \text{boolean}$

$$\begin{aligned}
 \text{ite}(\mathbf{t}, x, y) &= x \\
 \text{ite}(\mathbf{f}, x, y) &= y
 \end{aligned}$$

- For all $x, y \in \text{boolean}$,

$$\begin{aligned}
 \text{not } x &= \text{ite}(x, \mathbf{f}, \mathbf{t}) \\
 \text{and}(x, y) &= \text{ite}(x, y, \mathbf{f}) \\
 \text{or } \text{and}(x, y) &= \text{ite}(y, x, \mathbf{f}) \\
 \text{or}(x, y) &= \text{ite}(x, \mathbf{t}, y) \\
 \text{or } \text{or}(x, y) &= \text{ite}(y, \mathbf{t}, x)
 \end{aligned}$$