Solutions

1. Types, Polymorphism [14 pts] (parts a–b)

   (a) [9 pts] For each of the 3 functions below, pick an appropriate type for it from the list below.

   i. `fun f x y = (y, x)`

   (B) `'a -> 'b -> ('b * 'a)`.

   ii. `fun f (x, y) z = z y`

   (D) `('a * 'b) -> ('b -> 'c) -> 'c`.

   iii. `fun f x y = fn y => (y, x)`

   (G) `'a -> 'b -> 'c -> ('c * 'a)`

   A. `('a * 'b) -> 'c -> 'b`
   B. `'a -> 'b -> ('b * 'a)`
   C. `('a * 'b) -> ('b * 'a)`
   D. `('a * 'b) -> ('b -> 'c) -> 'c`
   E. `'a -> 'b -> 'c -> ('b * 'a)`
   F. `('a * 'b) -> 'c -> 'c * 'b`
   G. `'a -> 'b -> 'c -> ('c * 'a)`

   (b) [5 pts] Write a datatype declaration that describes a list structure with elements of alternating types. That is, odd elements have some type `'a`, and even elements have some type `'b`.

   Answer:

   ```
   datatype ('a,'b) list = Nil | Cons of 'a * ('b,'a) list
   ```

2. Pattern matching [18 pts] (parts a–b)

   (a) [8 pts] The following datatype describes syntax trees for arithmetic expressions:

```
datatype expr = Const of int
              | Plus of expr * expr
              | Times of expr * expr
```

Use pattern matching to implement a function `trans: expr -> expr` that transforms an expression e into an equivalent expression by distributing multiplication over addition for all subexpression of e. Make sure you identify all cases where this transformation is applicable.

Answer:

```
fun trans(e: expr): expr =
  case e of
    Times(e1, Plus(e2,e3)) => Plus(Times(trans e1, trans e2),
                                   Times(trans e1, trans e3))
  | Times(Plus(e1,e2), e3) => Plus(Times(trans e1, trans e3),
                                   Times(trans e2, trans e3))
  | _  => e
```

(b) [10 pts] Write a function `zip : int list * int list -> int list` combines two lists a and b by inserting the i-th element of a before the $2i$-th element of b. If list b is too short, excess elements are appended. For example:

```
zip([1, 2, 3, 4], [11, 12, 13]) = [1, 11, 12, 2, 13, 3, 4]
```

Answer:

```
fun zip(a: int list, b: int list): int list =
  case (a, b) of
    (x::tx, y::y'::ty) => x::y::y'::zip(tx, ty)
  | (x::tx, [y]) => x::y::tx
  | (_, []) => a
  | ([], _) => b
```

3. Using fold functions [20 pts] (parts a–c)

Absolute frequency distributions are obtained by dividing a data set into several classes and then counting the number of elements in each class. For example, an absolute frequency distribution for student grades in this exam can be constructed by considering the grade ranges (0-10), .., (80-90), (90-100), and counting the number of students in each range. We'll represent frequency distributions as lists of integers.

A histogram is a graphical representation of a frequency distribution graphically, as a sequence of vertical bars. The height of each bar indicates the frequency at that point. We want to write a function `histogram` that prints out a histogram in text form. For instance, `histogram([3,1,2]` must produce the following output:

```
*
* *
***
```

Below is a sketch of the code for this function. The function assumes that each element in the input list is non-negative.

```
fun histogram(dist: int list): unit =
let
    val num: int = ...
    fun line(i: int): string = ...
    fun show(i: int): string = if i = 0 then ""
                               else (line i) ^ "\n" ^ (show(i-1))
in
    print (show num)
end
```

The code for `num` and `line` can be implemented concisely using list folding operations. Recall the following definition:

```
fun foldl f v l = case l of nil  => v
                          | h::t => foldl f (f(h,v)) t
```

(a) [6 pts] Write the appropriate code for `num` using `foldl`.

Answer:

```
val num: int = foldl Int.max 0 dist
```

(b) [6 pts] Write function `line` using list folding.

Answer:

```
fun line(i: int): string =
   foldl (fn (x,a) => a^(if x < i then " " else "*")) "" dist
```

(c) [8 pts] A cumulative distribution is similar to an absolute frequency distribution, but for each range it counts the elements in that range, plus all elements in the smaller ranges.

Write a function `cumulative` that converts an absolute frequency distribution into a cumulative distribution. For instance: `cumulative([3,1,2])=[3,4,6]`. Implement this function using `foldl`. Make sure that elements in your result are properly ordered.

Answer:

```
fun cumulative(dist: int list): int list =
  rev( #2( foldl (fn(x,(y,z)) => (x+y,x+y::z)) (0,[]) dist ) )
```

4. Data Abstraction [18 pts] (parts a–b)

The following signature models sets of integers:

```
signature INTSET = sig
  (* A "set" is a set of integer values *)
  type set

  (* The empty set *)
  val empty: set

  (* insert(n, s) adds integer n to set s *)
  val insert: int * set -> set

  (* remove(n, s) deletes integer n from set s *)
  val remove: int * set -> set

  (* equal(s,t) is true if sets s and t contain the same values *)
  val equal: set * set -> bool
end
```

For sets that contain many consecutive numbers, it is more efficient to use value ranges, rather than enumerating all values. It is appropriate to represent such sets as lists of pairs, where each pair denotes an integer range:

```
type set = (int * int) list
```

In addition, we want to implement set equality in a straightforward manner:

```
val equal(s: set, t: set): bool = s = t
```

(a) [6 pts] Define an appropriate representation invariant that would make the above implementation of `equal` correct.

Answer: *A list* $[(a_0, a_1), (a_2, a_3), .., (a_{2n}, a_{2n+1})]$ *must be such that:*

- *All ranges are valid:*

$$a_{2i} \leq a_{2i+1}, \text{ for all } i$$

- *All ranges are disjoint, not adjacent, and in increasing order:*

$$a_{2i+1} \leq a_{2i+2} - 2 \text{ for all } i$$

The representation invariant ensures that each set has a unique implementation. Therefore, set equality via list equality is correct.

(b) [12 pts] Finish the implementation below for function `remove`, by filling in the inner case statement. Do not use other `if` or `case` statements in the remaining of the code.

You may find it helpful to use the function `Int.compare: int * int -> order` to compare two integers and obtain their ordering: `LESS`, `EQUAL`, or `GREATER`. You must ensure that your code maintains the representation invariant.

Answer:

```
fun remove(n: int, s: set): set =
    case s of
       nil => nil
    | (a,b)::t => case (Int.compare(n,a), Int.compare(n,b)) of
                    (LESS, _) => s
                  | (EQUAL, EQUAL) => t
                  | (EQUAL, _) => (a+1,b)::t
                  | (_, EQUAL) => (a,b-1)::t
                  | (_,LESS) => (a,n-1)::(n+1,b)::t
                  | _ => (a,b)::remove(n,t)
```

5. Complexity [20 pts] (parts a–c)

Consider the following implementation of list reversal:

```
fun reverse(l: int list): int list =
  case l of nil => nil
          | [x] => [x]
          | _ => let
                   val n = List.length(l) div 2
                   val x = List.take(l, n)
                   val y = List.drop(l, n)
                 in
                   (reverse y) @ (reverse x)
                 end
```

The complexity of an append operation `a @ b` is $O(n)$ in the length of the first list `a`. Function `List.length` is $O(n)$ where n in the length of its input list.

Function `List.take(l,n)` returns the first `n` elements of list `l`, and `List.drop(l,n)` returns the remaining elements. Both of these functions are $O(n)$ where n is their integer argument.

(a) [5 pts] Write a recurrence for the running time of `reverse`.

Answer:

$$\begin{aligned} T(0) &= c_1 \\ T(1) &= c_2 \\ T(n) &= 2T(n/2) + c_3 n + c_4, \text{ for } n > 1 \end{aligned}$$

(b) [10 pts] What is the complexity of `reverse`? Prove your answer.

Answer:

The complexity is $O(n \log n)$.

Take $c_1 = c_2 = c_3 = 1$. Ignore c_4 because it is subsumed by $c_3\, n$. Therefore:

$$\begin{aligned} T(0) &= T(1) = 1 \\ T(n) &= 2T(n/2) + n, \text{ for } n > 1 \end{aligned}$$

We show that $T(n) \leq n + n \log n$, for all $n >= 1$, using strong induction on n.

Induction statement. $P(n) = T(n) \leq n + n \log n$.

Base case $n = 1$. $T(1) = 1 = 1 + 1log1$.

Induction Hypothesis. Assume that $T(k) \leq k + k \log k$, for all $k = 0, .., n$ for some $n > 1$.

Induction Step. Prove that $T(n+1)leq(n+1) + (n+1)\ log(n+1)$. We have :

$$\begin{aligned} T(n+1) &= 2T((n+1)/2) + (n+1) \text{ (because n > 1)} \\ &\leq 2((n+1)/2\ log((n+1)/2) + (n+1)/2) + (n+1) \\ &\qquad \text{(by IH, because } (n+1)/2 \leq n) \\ &= (n+1)\ log((n+1)/2) + 2(n+1) \\ &= (n+1)\ log(n+1) \end{aligned}$$

This completes the proof. Hence $T(n) \leq n + n \log n$, so $T(n)$ is $O(n \log n)$.

(c) [5 pts] Write a list reversal function `reverse2` that runs faster than `reverse` for large lists. Needless to say, `List.rev` is not accepted as an answer.

Answer:

```
val reverse2 l = foldl List.:: nil l
```

6. Evaluation [10 pts]

Consider the following function:

```
fun foo (n: int) (f: int->int): int =
   if n < 3 then f(n)
            else foo (n-1) (fn m => f(n * m))
```

(a) [7 pts] Write the evaluation of expression `foo 3 (fn x=>x+1)`.

Write your solution using reductions of the form `e -> e'`. You may find it useful to name function values and use their names during evaluation. Here is an example of an evaluation using reductions:

```
(fn x => fn y => y x) 2 (fn x => x)
    ->  (fn y => y 2) id
        [where id  =  fn x => x]
    ->  id 2
    ->  2
```

Answer:

```
 foo 3 (fn x => x)
    -> (fn f => if 3 < 3 then f(3)
                else foo (3-1) (fn m => f(3 * m))) f1
[where f1 = fn x => x+1]
    -> if 3 < 3 then f1(3) else foo (3-1) (fn m => f1(3 * m))
    -> if false then f1(3) else foo (3-1) f2
[where f2 = fn m => f1(3 * m)]
    -> foo (3-1) f2
    -> foo 2 f2
    -> (fn f => if 2 < 3 then f(2)
                else foo (2-1) (fn m => f(2 * m))) f2
    -> if 2 < 3 then f2(2) else foo (2-1) (fn m => f2(2 * m))
    -> if true  then f2(2) else foo (2-1) f3
 [where f3 = fn m => f2(2 * m)]
    -> f2(2)
    -> f1(3 * 2)
    -> f1(6)
    -> 7
```

(b) [3 pts] Given a number `n > 0` and and arbitrary function `f`, what does the expression "`foo n f`" compute?

Answer: *It computes* `f(n!)`