F28PL - Programming Languages

Question G(Programming)

Tommy Lamb

H00217505

The plain text file which accompanies this PDF contains a listing of all functions, with some formatting to aid readability. It can be opened and executed in Poly/ML from the Terminal by navigating to the save location of the file, opening Poly/ML, and using the command : ' use "F28PL - Tommy Lamb - Question G - Code.txt"; '. This method will cause Poly/ML to ignore the document formatting and only display system output (not input). To maintain formatting, all text can be copied and pasted into a running instance of Poly/ML. For best results it should be copied/pasted in small blocks, around 1 question in size.

Where answers comprise of more than one function, each is listed and commented on separately.

1. Represent a binary number as a list of booleans, where the first boolean is the *least* significant digit. Thus, `[false,true]` represents two and `[false,true,false,true]` represents ten. Recall the school algorithm for addition, involving carrying ones over to more significant digits as appropriate. Implement this binary addition as a function of type `bool list -> bool list -> bool list`.

   A.
   ```
   fun xor (x, y) = (x orelse y) andalso (not (x andalso y));
   infix xor;
   ```
   ```
   (bool * bool) -> bool
   ```
   A function implementing the logic of XOR (exclusive or) using the inbuilt methods of logical AND, OR, and NOT. Also given infix status in order to improve readability of this function when in use; `"A" xor "B"` makes more sense than `xor "A" "B"`, and matches standard usage of logical connectives.

   B.
   ```
   fun bitadder x y carry = ((( x xor y) xor carry),
       (x andalso y) orelse (carry andalso (x xor y)));
   ```
   ```
   bool -> bool -> bool -> (bool * bool)
   ```
   The function which handles adding two single-bit numbers and factors in a carry bit. Using the previously defined xor function it calculates the resulting bit value and carry bit from the addition and returns both in a tuple. The exact logic used here was researched and constructed with particular reference to this Wikipedia GIF image as a refresher of knowledge.

C.
```
fun adder [] [] carryin result = result@[carryin]
  | adder [] (head::tail) carryin result =
        let
              val (sum, carryout) = bitadder false head carryin
        in
              adder [] tail carryout (result@[sum])
        end
  | adder (head::tail) [] carryin result =
        let
              val (sum, carryout) = bitadder head false carryin
        in
              adder tail [] carryout (result@[sum])
        end
  | adder (headx::tailx) (heady::taily) carryin result =
        let
              val (sum, carryout) = bitadder headx heady carryin
        in
              adder tailx taily carryout (result@[sum])
        end;
```

```
bool list -> bool list -> bool -> bool list -> bool list
```

This is the main computational function of this program, as it recursively applies the single-bit addition function `bitadder` to pairs of elements from both input lists. Alongside the two lists it takes an accumulator variable `result` to construct the resulting list, and a variable `carryin` as the carry bit from the preceding addition operation.

Pattern matching is used here to handle cases where the length of input lists differs, as well as providing the base case for the recursion. In this base case where both lists are empty, the carry bit is simply tacked on the end of the result list. This is of course how addition works, though it does not discriminate between 1 and 0, thus a 0 (false) is often inserted at the end of the result - semantically the equivalent of writing 0200 rather than 200. In each step case where one list is empty and the other not, addition is carried out as normal, substituting 0 (false) for the empty list. This has the effect of padding out either list with zeroes such that each list is the same length. The final step case just carries out the addition using `bitadder` on the heads of both lists and the `carryin` carry bit value.

The local declarations are used here to pattern match on the tuple returned by `bitadder`, and make it useable in the function. In each step case, the `sum` value is concatenated to the `result` accumulator and the `carryout` value (the carry bit resulting from the addition) is passed to the succeeding call alongside the rest of the lists (where applicable) and the `result` accumulator.

D.
```
fun binaryaddition listx listy = adder listx listy false [];
```

```
bool list -> bool list -> bool list
```

Nothing more than a wrapper function for adder to hide away the arguments used to enable recursion, and to provide a function of the type specified in the question.

2. Using an accumulator variable or otherwise, write a function max of type `int list -> int` that returns the greatest element of its input.

   A.
   ```
   fun calcmax [] maxvalue = maxvalue
     | calcmax (head::tail) maxvalue =
         if head>maxvalue then
             calcmax tail head
         else
             calcmax tail maxvalue;
   ```

   ```
   int list -> int -> int
   ```

   This function quite simply iterates through a provided list, checking each element against the `maxvalue` variable, and changing the value depending on which of the two values is larger via an if statement. Pattern matching on the empty list returns the `maxvalue` variable in the base case - where the list has been fully traversed. By virtue of the operations being performed and accumulator variable, this algorithm is entirely tail recursive.

   B.
   ```
   fun max [] = ~0
     | max (head::tail) = calcmax tail head;
   ```

   ```
   int list -> int
   ```

   A wrapper function used to cope with the edge case of the empty list and to initialise the value of `maxvalue` to the first element before recursively checking the rest. Also has the type required by the question. Initialising `maxvalue` so allows the program to operate with negative numbers as well as positive, as there is no assumption that 0 is smaller than everything in the list. The empty list edge case attempts to return minus zero (~0) as a number which no function should ever reasonably have as a normal mathematical result; both 0 and minus 1 are reasonable results for successful execution. Unfortunately Poly/ML automatically converts this to positive 0, which does not seem a reasonable error return value. As well as also being returned in a normal application, the implication that the maximum value in the empty list is 0 does not sit well. The empty list by its very virtue has no elements; none larger, smaller, or equal to 0 - none.

3. Using accumulator variables or otherwise, write a function avg of type real list -> real that returns the average of a list.

   A.
   ```
   fun calcavg [] total length = total / length
     | calcavg (head::tail) total length = calcavg tail (total+head) (length+1.0);
   ```

   ```
   real list -> real -> real -> real
   ```

   Much like in Q2, this function iterates through a `real list` and adjusts the accumulator values `total` and `length`. Once the list has been exhausted (is empty) the average is calculated from these values and returned up the stack.

   B.
   ```
   fun avg [] = 0.0
     | avg list = calcavg list 0.0 0.0;
   ```

   ```
   real list -> real
   ```

   As previously, a wrapper class is used to satisfy the function type in the question and to initialise values for the accumulator variables, namely to 0. Having learned from Q2 this function simply returns `0.0` and does not attempt to make it negative, though the uncomfortable assumption remains.

4. (Harder) Using accumulator variables or otherwise, write a function modes of type `int list -> (int list*int)` that returns the list of those integers that occur most often in the input, along with the number of times they occur (see mean, median, and mode).

   A.
```
fun countoccurances value count [] = (value, count)
  | countoccurances value count (head::tail) =
   if head = value then
          countoccurances value (count+1) tail
   else
          countoccurances value count tail;
```

```
''a -> int -> ''a list -> ''a * int
```

   Given a `value` variable and a list, this function returns the number of occurrences (frequency) of that value in the list using recursion and the accumulator variable `count`. The function returns this information as a *key-value* pair of the form `(value, count)`. The reason for this is that it better fits with the usage in later defined functions.

   B.
```
fun insertmap (value, frequency) [] = [(value, frequency)]
  | insertmap (value, frequency) (head::tail) =
   if (value = (#1 head)) orelse (frequency < (#2 head)) then
          head::tail
   else if value < (#1 head) then
          (value, frequency)::head::tail
   else
          head::(insertmap (value,frequency) (tail));
```

```
int * int -> (int * int) list -> (int * int) list
```

   Taking inspiration from Question F4 (Strings and Chars exercise), this function inserts a *key-value* pair into a provided map-like structure, sorted by *key* and filtered somewhat by *value*. Iterating recursively through the provided list of *key-value* pairs, if the key is found or the frequency value is smaller than the element here then the list is just returned. This means that duplicates are not allowed, and that values whose frequency is too low to be considered a potential mode are filtered somewhat. The filtering is not perfect, but it can limit the size of the final frequency map (as it's later referred to) and make the program somewhat more efficient - at the expense of reusability. Sorting on the value just makes it more efficient and complete in finding when the values are already present.

   Unfortunately as a result of using the "<" comparator with the `value`, `frequency`, and both elements of the head tuple, these variables are cast to `int`, losing all abstraction that could otherwise be achieved. This would be a prime candidate for using Java-style generics.

   C.
```
fun calcfrequency freqmap [] = freqmap
  | calcfrequency freqmap (head::tail) =
   calcfrequency (insertmap (countoccurances head 0 (head::tail)) freqmap) tail;
```

```
(int * int) list -> int list -> (int * int) list
```

   Using the accumulator variable `freqmap` this function recursively constructs a complete frequency map for any given list, utilising the previously defined `countoccurances` and `insertmap` functions. For each value in the list, it first counts the frequency of it, before attempting to insert this result into the map. This is the reason that a map structure was used: duplicate entries will be generated, as each value will be counted as many times as it appears in the list, not just once. Given the partial filtering in `insertmap` the frequency map is not a complete representation of all entries in the provided list, but it does represent a list of potential modal values and the frequency of each within the list.

```
D. fun separatefrequencies freqlist [] = freqlist
     | separatefrequencies freqlist ((head:('a*'b))::tail) =
            separatefrequencies ((#2 head)::freqlist) tail;
```

```
'a list -> ('b * 'a) list -> 'a list
```

Using the `freqlist` accumulator, this function recursively strips the frequencies from a frequency map into a separate list. Actually, it will work on any list of 2-value tuples given the deliberately abstract type casting employed. The casting itself is necessary to declare to the compiler that head should be a tuple of two values. Without this, the compiler complains that head is treated as a tuple without any evidence that's what it is.

Note that this method reverses the order of frequencies in the map. However not for any purpose.

```
E. fun maxfrequency freqmap = max (separatefrequencies [] freqmap);
```

```
('a * int) list -> int
```

Utilising the `max` function from Question 2 and previously defined function `separatefrequencies`, this function returns the maximum frequency value found in the frequency map. That is to say, the frequency of all modal values contained in the map, and thus the list from which it was calculated.

```
F. fun calcmode modelistin maxfrequencyin [] = (modelistin, maxfrequencyin)
     | calcmode modelistin maxfrequencyin ((head:'a*''b)::tail) =
       let
              val (modelist, maxfrequency) = calcmode modelistin maxfrequencyin tail
       in
              if (#2 head) = maxfrequency then
                     (((#1 head)::modelist), maxfrequency)
              else
                     (modelist, maxfrequency)
       end;
```

```
'a list -> ''b -> ('a * ''b) list -> 'a list * ''b
```

This function uses an accumulator variable `modelistin` and `maxfrequencyin` to construct the list of modal values and return it as a tuple, as specified by the question. This is achieved by recursively checking the frequency value of each value in the frequency map against the provided maximum frequency value. Each value whose frequency matches is concatenated to the mode list accumulator, and this accumulator is returned once the frequency map is exhausted.

The local declaration is used to manage the tuple returned by each recursive call, parsing the values out to allow formatting for return further up the stack, and to allow addition of modal values where necessary.

G.
```
fun modes list =
   let
         val freqmap = calcfrequency [] list
   in
         calcmode [] (maxfrequency freqmap) freqmap
   end;
```

```
int list -> int list * int
```

The final function of this program which pulls together all of those previously defined into a single cohesive operation of the type defined in the question. It first uses a local declaration to hold the frequency table of the argument list, calculated using the `calcfrequency` function (C). This is done so that the frequency map need only be calculated once; due to the extremely high growth rate of the operation it was decided that this was necessary. Using this frequency table, the maximum frequency value is calculated using `maxfrequency` (E), which is then passed as argument to `calcmode` (F) which ultimately constructs the tuple containing the list of modal values, and their frequency within the list.

There is a large question over just how good this algorithm, this program, is for completing its task. While it achieves the right answer, the code seems unwieldy and unclear, and the growth rate is terrifying. In order to calculate mode one must calculate the frequency of each value, that is unavoidable. This gives any algorithm a starting point of $O(n^2)$: one iteration to read in each number, and another to count its frequency. Using a proper hash-map would have an access time approximating $O(1)$, however this program uses nothing of the sort. In fact, the method used here has a worst case of $O(n)$ if all elements are inserted, though the average case is suspected to be much better (thanks to the ordering and filtering). On top of this is the fact that calculating the maximum frequency value is also $O(n^2)$. One iteration through the frequency map to strip out the frequencies into a list, and another to find the maximum. All told, a worst case approximately $O(n^3 + n^2) = O(n^2)$

Of course (with hindsight) one should either calculate the maximum just by iterating through the frequency map, not splitting it out and then iterating again. Even better would be to calculate it alongside the frequency map, eliminating all iterating uniquely required to find `maxfrequency`. This latter method would require a not insubstantial refactoring and reworking of the code, as it stands each function is heavily dependent on the return type of other lower functions which would change dramatically with such a change in methodology. This was realised too late to be implemented and tested fully. It was also revealed later that there was an inbuilt method for splitting a list of tuples into two lists of values.

On the plus side, this code has rather high cohesion and considerable reusability as a result of the high number of functions. It would take little work to adapt some of the underlying functions to similar tasks, like displaying the entire frequency map.