F28PL - Programming Languages

Question C (Lists)

Tommy Lamb

H00217505

The plain text file which accompanies this PDF contains a listing of all functions alongside test function calls, 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 C - 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.

Note:
Certain functions utilise other functions defined in earlier questions. Any such prerequisite code is noted in blue for each applicable question.

1. Write a function to drop the first n elements of a list l:
   `drop n l : int -> 'a list -> 'a list`
   `drop 3 [1,2,3,4,5] ==> [4,5]`
   - To drop 0 elements from a list return the list.
   - To drop n elements from the empty list, return the empty list.
   - To drop n elements from a list, drop n-1 elements from the tail.

   - ```
     fun drop 0 l = l |
     drop n [] = [] |
     drop n (head::tail) = drop (n-1) tail;

     type: int -> 'a list -> 'a list
     ```

     Pattern matching provides the two base cases, where $n = 0$ and when there are no elements that can be removed (the empty list). Otherwise it recursively calls itself, decrementing n and dropping the head of the list each time. When $n_0 < 0$ the code will never hit the $n = 0$ base case, and only stops when the empty list is provided. Thus the end result is that, regardless of the size of list, every element is dropped and the empty list returned. 'Cons' are used to separate the head from the rest of the list argument.
     ```

2. Write a function to take the first n elements of a list l:

```
take n l : int -> 'a list -> 'a
take 3 ["a","b","c","d","e"] ==> ["a","b","c"]
```

   - To take 0 elements from a list return the empty list.
   - To take n elements from the empty list, return the empty list.
   - To take elements from a list, put the head of the list onto the result of taking n-1 elements from the tail.

- ```
  fun take 0 l = [] |
  take n [] = [] |
  take n (head::tail) = [head]@(take (n-1) tail);

  type: int -> 'a list -> 'a
  ```

  This function acts similarly to Q1, with a few differences. In the $n = 0$ base case, the empty list is returned instead (taking nothing as opposed to dropping nothing) and rather than dropping the head, it is concatenated to the result of the recursive calls. By decrementing n eventually a base case will be reached where the empty list is returned, to which all head elements from previous calls are added. The recursive result is:

  $$head_0 + head_1 + head_2 \ldots + head_{n-1} + []$$

  or

  $$head_0 \ldots + head_{y-1} + []$$

  where $y = length\ of\ list\ L$

  Much like with Q1, when $n < 0$ the entire list - regardless of length - is returned.
  ```

3. Write a function to check if list l1 starts list l2:

```
starts l1 l2 : ''a list -> ''a list -> bool
starts [1,2,3] [1,2,3,4] ==> true
```

   - An empty list starts a list.
   - A list does not start an empty list.
   - A list starts a second list if they have same head and the tail of the first starts the tail of the second.

- ```
  fun starts [] l2 = true |
  starts l1 [] = false |
  starts (head1::tail1) (head2::tail2) =
     if head1 = head2 then
           starts tail1 tail2
     else
           false;

  type: ''a list -> ''a list -> bool
  ```

  First checks if either list is empty and acts according to the specification, which also acts as the base case. If neither is empty, the first two elements are evaluated. If they do equate, then the remainder of both lists are recursively checked, otherwise false is returned. Hitting the second base case (`l1 [] = false`) returns false because if L1 is longer than L2, L2 cannot possibly begin with L1. The first base case (`[] l2 = true`) returns true because if we have exhausted L1 then it must be true, otherwise false would have been returned by the if statement before reaching this state.
  ```

4. Write a function to check if list l1 is contained in list l2:
   `contains l1 l2 : ''a list -> ''a list -> bool`
   `contains ["d","e","f"] ["a","b","c","d","e","f","g","h"] ==> true`
   - A list is not contained in an empty list.
   - A list is contained in a second list if it starts the second list.
   - Otherwise, a list is contained in a second list if it is contained in the tail of the second list.

- ```
  fun contains l1 [] = false |
  contains [] l2 = true |
  contains (head1::tail1) (head2::tail2) =
     if head1 = head2 then
           starts tail1 tail2
     else
           contains (head1::tail1) tail2;

  type: ''a list -> ''a list -> bool
  ```

  Prerequisite - Q3: "fun starts ..."

  Recursively checks if the head of list 1 equals that of list 2, iterating through list 2 (but not list 1) until the condition is met (or the base case `l1 [] = false` is reached). If the base case is reached, no element in list 2 equates to the head of list 1, and so false is returned. Otherwise calls `starts` to check if the next n **concurrent** elements of the two lists are the same, where `n = length(List 1)`. Using a recursive call to `contains` would not suffice here, as it would not check concurrent elements - which is required by the semantics of "starts" as defined in Q3. This method, if somewhat counter-intuitively, does work for single element lists. In such cases the empty list is passed to `starts` where the base cases for that function return the correct result.
  The second pattern matching of `contains [] l2 = true` is required, else a match exception would be raised in such a case as a result of non-exhaustive checking. It is true given that "(Q3) An empty list starts a list." and "(Q4) A list is contained in a second list if it starts the second list."

  There was great internal debate whether the implementation of the `if` statement above was better (or worse) than that alluded to by the question:

  ```
  if starts l1 (head2::tail2) then
     true
  else
     contains l1 tail2
  ```

  The conclusion reached was that while the latter may be somewhat more understandable, the former was *marginally* more efficient. Since on average the former would make less calls to `starts` it will avoid more of the (likely negligible) overhead of calling another function. Ultimately no sufficient justification could be found to change the answer from its first incarnation.
  ```

5. Write a function to delete a list from another list:
   `delete l1 l2 : ''a list -> ''a list -> ''a list`
   `delete [3,4,5] [1,2,3,4,5,6] ==> [1,2,6]`
   - To delete a list from the empty list, return the empty list.
   - To delete a list from a second list, if the first list starts the second list then drop the length of the first list from the second list.
   - Otherwise, put the head of the second list onto the result of deleting the first list from the tail of the second.

   - ```
     fun delete l1 [] = [] |
     delete l1 (head2::tail2) =
        if starts l1 (head2::tail2) then
              drop (length l1) (head2::tail2)
        else
              ([head2]@ (delete l1 tail2));

     type: ''a list -> ''a list -> ''a list
     ```

     Prerequisite - Q3: `"fun starts ..."`
     Prerequisite - Q1: `"fun drop ..."`

     Much like Q4 this function recursively iterates through the second list, until the first list is found contained within the second (or reaches the first base case `l2 = []`). When list 1 is found (`starts l1 (head2::tail2) = true`), the length of that list is dropped from the second, returning whatever is left of list 2 afterwards. Each time list 1 is not found a recursive call is made on the tail of list 2, however the current head of list 2 is concatenated to the result of the call. The result is that the leading values of list 2 are preserved in the call stack, to which the base case return list is appended. The first base case is met when list 1 is not contained within list2, and thus cannot be dropped. In this case the empty list is returned up the stack and every single value of list 2 is concatenated to it. This also results in `length(l2)` recursive calls.
     The second base case is when list 1 is found within list 2, and subsequently dropped.
     ```

6. Write a function to delete every occurrence of a list from another list:
   `deleteAll l1 l2 : ''a list ->''a list -> ''a list`
   `deleteAll [1,2,3] [3,2,1,2,3,2,1,2,3] ==> [3,2,2]`
   - All occurrences have been deleted from an empty list.
   - If the first list starts the second list, delete it and then delete all occurrences in the remaining list.
   - Otherwise, put the head of the second list on the front of deleting all occurrences of the first list in the tail.

   - ```
     fun deleteAll l1 [] = [] |
     deleteAll [] l2 = l2 |
     deleteAll l1 (head2::tail2) =
        if starts l1 (head2::tail2) then
              deleteAll l1 (drop (length l1) (head2::tail2))
        else
              [head2]@(deleteAll l1 (tail2));

     type: ''a list -> ''a list -> ''a list
     ```

     Prerequisite - Q3: "fun starts ..."
     Prerequisite - Q1: "fun drop ..."

     As with Questions 4 & 5 above this function iterates recursively through list 2 until list 1 is found or list 2 is exhausted. The difference here is that whether or not list 1 is found a recursive call is made. If it is found (`if starts l1 (head2::tail2)`) then it is dropped from list 2, and the remainder of list 2 is searched for more occurrences of list 1. If it is not found at the current location in list 2 then the current head is concatenated to the result of searching the remainder of list 2, like in Q5. Regardless of how often (or not) list 1 occurs in list 2, a recursive call is made where the second list passed is empty. So eventually the empty list is retuned up the stack for each non-dropped element of list 2 to be concatenated to the front of.
     Unfortunately even in the absolute best case (ignoring base case) this has growth rate O(N), as every single element of list 2 has to be checked against the head of list 1, or the entirety of list 2 must be dropped (if list1 = list2).
     The second case (`[] l2 = l2`) is not strictly necessary as the function should never reach it through execution, though it is helpful in preventing overzealous edge-case-obsessed software testers from crashing their computers as the result of a non-halting recursion (as I did).
     ```