[CONFIDENTIAL] Covariant Interview: FoldR

Rules

- You may write your solutions in any of the available languages:
    - C++
    - C#
    - Java
    - Javascript
    - Python

  Due to email system restrictions, the ".txt" extension is added to each source code file. Please remove the extra extension to recover the original files.

  Make sure to submit your source code as well as any instructions to build / run your code.
- You should write your code in the blocks enclosed by `[BEGIN] YOUR CODE HERE` and `[END] YOUR CODE HERE`. You should NOT edit any other code.
- If you get stuck, you may ask for a hint by contacting your interviewer.
- Your solution will be evaluated not only by correctness (including robustness to handle edge cases), but also by quality of the code.

Introduction

We start with a constructor for a linked list, `cons`, which takes a required argument `head` and an optional argument `tail`. It returns a linked list representation where the first element is `head` and the rest of the elements are contained in the `tail` linked list. Empty lists are represented by the null value in the corresponding language:

- `nullptr` in C++;
- `null` in C#;
- `null` in Java;
- `undefined` in Javascript;
- `None` in Python.

For example, `cons(1, cons(2, cons(3, cons(4))))` constructs a linked list with 4 elements, `1 2 3 4`.

To make it easy to inspect the content of a linked list, the `listToString` method is defined to convert the list to a string, where the string representations of each element are joined by a single space in between.

The myMap method takes a unary function fn and a linked list list. It calls fn in order over each element in list and returns a linked list of the return values of fn.

The myReduce method calls a reducer function fn from the beginning of the list to the end and returns the result. For example, if the list is cons(1, cons(2, cons(3))), myReduce(fn, accm, list) should return the result of evaluating fn(fn(fn(accm, 1), 2), 3).

All three methods above are implemented using recursion, leveraging the recursive structure of the linked list.

Part 1: Implementing myReduceRight

Implement the myReduceRight method. This is similar to myReduce, with the difference that it calls the reducer function fn from the end of the list to the beginning. For example, if the list is cons(1, cons(2, cons(3))), myReduceRight(fn, accm, list) should return the result of evaluating fn(1, fn(2, fn(3, accm))).

Requirements:

- You SHOULD implement your solution using recursion, instead of any explicit for / while loops.
- You MUST NOT use any of the previously defined listToString, myMap, myReduce methods in your implementation.
- You MUST NOT mutate the original list.

To check your implementation, verify that:

- myReduceRight(xTimesTwoPlusY, 0, exampleList) should evaluate to 20.
- myReduceRight(unfoldCalculation, "accm", exampleList) should evaluate to fn(1, fn(2, fn(3, fn(4, accm)))).
- myReduceRight(printXAndReturnY, 0, exampleList) should print out the content of the list in the reverse order.

Part 2: Implementing myMap2

Implement the myMap2 method using myReduceRight. This should be functionally equivalent to myMap.

Basic Requirements for your solution:

- You MUST NOT use any of the previously defined listToString, myMap, myReduce methods in your implementation.
- You MUST NOT modify any of the function signatures, including that of myReduceRight.

- You MUST NOT hack the implementation of `myReduceRight` for the purpose of this part, for example passing a hidden flag to `myReduceRight` to signal special handling.
- You MUST NOT use any language-native special data structures (e.g. `std::vector` in C++, `ArrayList` in Java, `list` in Python).

You will get "bonus points" for your solution, if it satisfies as many of the following requirements as possible:

- Do not use any explicit recursion. In particular, avoid calling `myMap2` within your implementation.
- Do not use any explicit for / while loops in your implementation. Therefore you need to figure out a way to make clever use of `myReduceRight`.
- Do not mutate the original list.

Feel free to include multiple solutions in this part (you can name those differently). You will be graded on both the correctness and the cleanness of your code.

Here are a few directions that you can follow:

- List reversal.
- Mutate the list within the reducer method.
- Consider clever use of closures and lambda functions to change the order of evaluation. In particular, consider the use of delayed evaluations, e.g. (() -> doSomething)().

To check your implementation, verify that:

- `listToString(myMap2(plusOne, exampleList))` should evaluate to `2 3 4 5`.
- `myMap2(printAndReturn, exampleList)` should print the list in the correct order (`1 2 3 4` each on a separate line instead of `4 3 2 1`).