Lab 7: Testing, HOF on Composite Data Structures, Policy abstraction, continuation passing

DUE to Moodle Sunday, August 5, 2018 at 20:00 at the latest.

Table of Contents

Prompt

You might go on to build software for large companies with a team of test engineers that will write all the tests for you. You might want to actually be one of those test engineers. You might want to work at a smaller company where you are both the developer and the tester. Perhaps you actually don't want to go into the software industry at all. Regardless, knowing how to test your work is a very valuable skill to have. For this reason, among others, I will not be providing you with an autograder for this lab.

We ended up not doing Lab5 so that we could spend more time on HOFs and Continuation passing. Accordingly, this lab will explore testing your work for problems that continue to challenge your understanding of HOFs and Continuation passing.

Source material

I don't know how to set up a Scala project in intelliJ and it didn't seem to be the best use of my time in setting up this lab. Instead to get us started I have created a few scripts for you work on. Below are recommendations on how to use the provided files.

- L7Q1.scala : Use this file as your development space for coding question 1. Test and check your work here. To encourage TDD I have included a structure for testing your work and the test function before each item that you need to implement.
- L7Q2.scala : Use this file as your development space for coding question 2. Test and check your work here. To encourage testing I have provided some test structure.
- L7Q3.scala : Use this file as your development space for coding question 3. Test and check your work here. To encourage you to write your own testing structure I have not provided testing support here.
- last_first_src.scala: Put your final solutions in this file. Rename this file to have your actual first and last name. (e.g. my name is Spencer Wilson, I will submit a file names `wilson_spencer_src.scala`)

What to expect

- You can do this at any pace you want… BUT
- I'll have lab goals on Piazza
- I'll post some kind of interactive coding exercises, lab hints, readings and videos for you to do every night.
    - I might get this together over the weekend and release it to you as a Chapter 7 for the GistOfPL
    - This material will include many of the test scripts and the test framework that I will be using to test your work
    - Watch Piazza for news on this matter.
- You still have nightly Moodle assignments.

Grading

- 50% code
- 20% writeup
- 30% written interview Tuesday, August 7, 2018 5:45 – 6:15 in our classroom.

Submission

This assignment is Due to Moodle by Sunday, August 5, 2018 at 20:00. No exceptions.

When you are done with the lab you will submit a <last_first>_writeup.pdf file and a <last_first>_src.scala file to Moodle. You will submit these as separate files, NOT as a zip file. Your name needs to be spelt in the way it is in the gradebook. Further details bellow.

<last_first>_writeup.pdf shall contain solutions to all problems listed as writeup, this includes questions:

- 1.c.i.
- 2.b.
- 3.b.
- 4

<last_first>_src.scala shall contain solutions to all coding questions. It shall not call any functions in the main scope of the script. It shall only define the functions. In addition to having these functions you are allowed to have any helper functions that you would like. To receive full credit the functions must also be implemented correctly according to their specification in this lab AND you must provide the definition of all helper functions used. To receive any credit, you must have all of the following functions declared correctly according to the type definitions provided or specified:

- (not so much a function, please include the provided definition of :) BT[A] and BinT[A]
- Any HOF for BT[A] used in BTLLfoldLeft and BTLLmap
- BTLLfoldLeft
- BTLLmap
- sumBinnedTree
- isBinnedSearchTree
- getByPolicy
- getEvens
- getNonNegs
- getConsonants
- findByPolicyIff
- getFirstFibSeqSTLenGtN **(this one is rather challenging)**

Questions

Q1. Composite data structures

1. In L7Q1.scala I have provided you with a definition for a Binary Tree (BT) and a Binned Binary Tree (BinT). The Binned Binary Tree is a Binary structure in which each data point in the structure itself is some a linear structure containing the true data points of the binned binary tree. In this assignment our Binned Binary Tree is a BT[List[A]], so it uses a homebrewed binary tree (a binary structure) alongside the Scala native List structure (a linear structure). Complete the following tasks.
   - **a.** <u>**CODE**</u>**:** In L7Q1.scala I have declared **BTLLfoldLeft** and **BTLLmap** which should solve fold left and map over a binned binary tree respectively. There are so many ways to solve these problems but… Please solve these problems using Higher Order Functions of the substructures of a binned binary tree and list. (e.g. your solution to BTLLfoldLeft should use one of the provided HOFs over our BT[A] structure, it should also use of the Scala HOF methods for List[A])
   - b. <u>**CODE**</u>: In L7Q1.scala implement a function named **sumBinnedTree.** This function shall take as input a binned binary tree of integers. This function shall return as output an Int. This function shall return the sum of all integers found in the input. You shalll implement this function using one of the HOFs defined in question 1.a. of this handout.
   - **c.** *valid ascending ordered binned binary <u>search</u> tree*. Not all binned binary trees are search trees. But, in practice one use of the binned binary tree is to create a search tree where duplicate values are important. Below is an explanation of the structure. You will need to implement questions 1.c.i. and 1.c.ii.

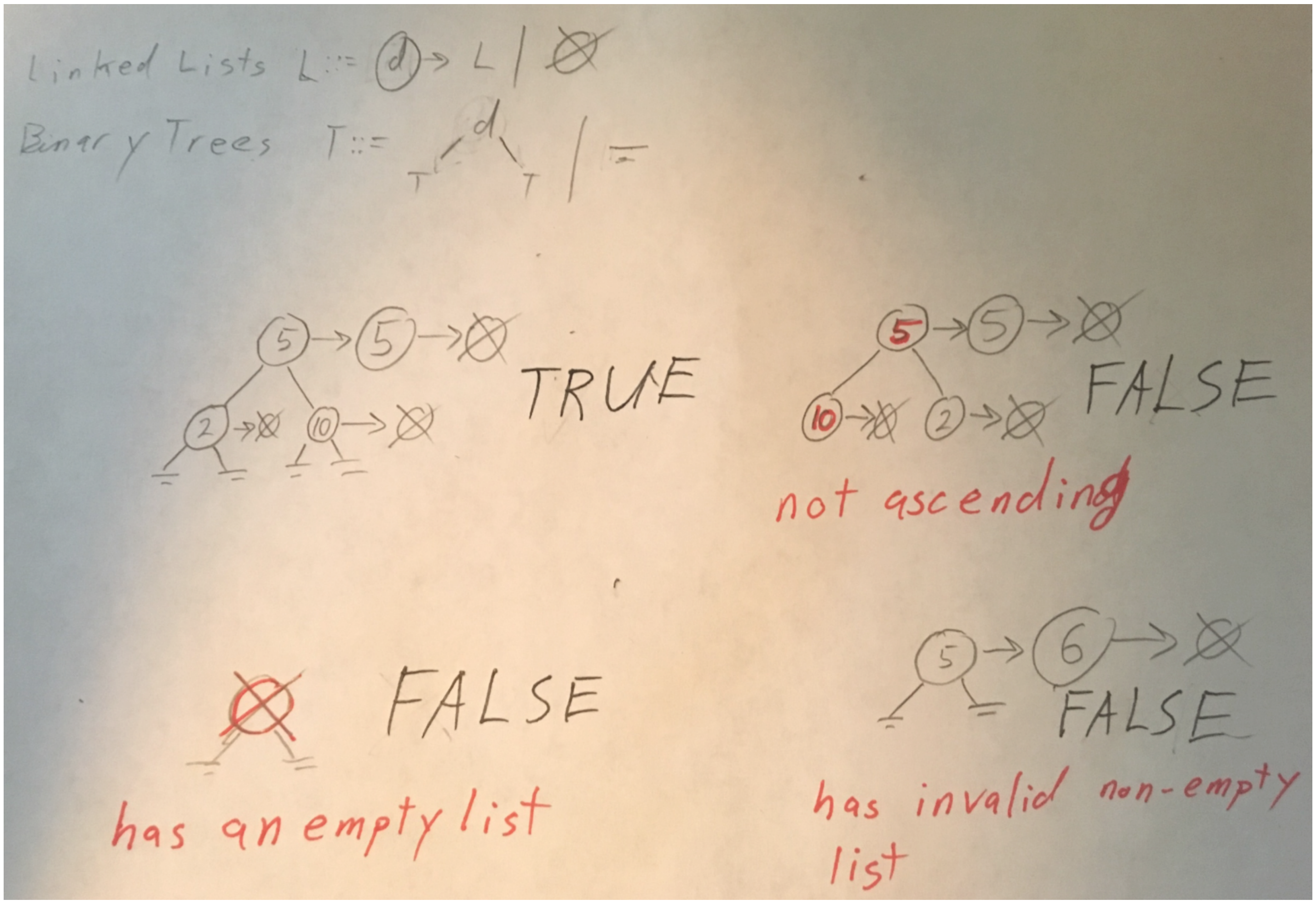
- Let t denote a tree of lists of integers
- Let li denote an observation of some single list of integers contained in t
- Let ni denote the head element of some valid li

- For t of ordinance (~size~) k to be valid (further details bellow)
  - all li must be valid
  - all ni in t must be strictly increasing

- For t to be valid all li must be valid
    - If t contains no li, then t is considered to be valid
    - For an li to be valid, the following must hold true
        - li must not be empty
        - let ni denote the head element of li
            - valid li can be comprised only of ni
        - examples of valid li include
            - List(1,1,1,1)
            - List(50, 50)
            - List(6)
        - Examples of invalid li
            - List() because "li must not be empty"
            - List(1,50,6) because ni = 1 but there are elements of the list that are not equivalent to ni

- For t of ordinance (~size~) k to be valid, all ni in t must be strictly increasing
    - Let k denote the count of lists li contained in t
    - If t is comprised only of valid li, then performing an in order traversal of the tree we will observe in series: $\{ n_0, n_1, n_2, \dots , n_i, \dots, n_{k-1} \}$ such that n_i is the is the head element of l_i for all i in range [0 , k)
    - For t to be valid $n_i < n_{i+1}$ for all i in range [0 , k)

Here is a picture showing examples of valid and invalid binned serch trees


Linked Lists L ::= d → L | ∅
Binary Trees T ::= d T T | ⊥
TRUE
FALSE
not ascending
FALSE
has an empty list
FALSE
has invalid non-empty list

i. **<u>WRITEUP</u>:** Write a set of inference rules over the above provided logic for a *valid ascending ordered binned binary <u>search</u> tree*. You will need to define your own grammar for lists, binary trees, and perhaps binned binary trees. You will need to define your own operational semantics. If the operational semantic is non-obvious, please clearly state what it attempts to accomplish. You are welcomed and encouraged to write inference rules for any subproblems that you observe.

ii. **<u>CODE</u>**: In L7Q1.scala I have provided the stub for a function **isBinnedSearchTree** implement the logic for determining if a *binned binary <u>search</u> tree* is valid provided in question 1.a. of this handout. (I also provided the stub of a useful helper function that you might consider implementing, namely **nonEmptyAndSame**)

Q2. Policy abstraction

2. Not sure how technical of a term this is, but I call it "policy abstraction". Consider that often times we solve problems that are nearly identical but which use some slightly different block of code in the middle of the work being performed. We'll call that block of a code a 'policy'. We can write a function that takes in the original inputs plus a policy and solves the problem being asked. Complete the following tasks
   a. CODE:
      i. Implement a function named **getByPolicy**. This function shall work with a type object, let's call it A. This function shall take as input a list of type List[A] and also a policy of type (A) => Boolean. The inputs to getByPolicy shall be curried with the list before the policy. This function shall return a List[A]. This function shall return the in-ordered sub-list of items in the input list such that policy returns true for that item in the input list. For an added challenge our solution shall be implemented using the 'flatMap' method for List[A]
      ii. Implement the following functions according their respective provided specifications. Each function shall take as Input a List[A] and shall return a List[A]. Each function shall be implemented using a single call to getSubListByPolicy.
         1. **getEvens**
            a. 'A' shall be an Int (i.e. the input and output is List[Int])
            b. return the sub-list of all even values in the input list in the order they were observed
         2. **getNonNegs**
            a. 'A' shall be an Int
            b. return the sub-list of all non-nagative values in the input list in the order they were observed
         3. **getConsonants**
            a. 'A' shall be shall an Char
            b. You may assume that the input list contains only characters in range [a, z]. I will only test on these inputs but you might think about how you'd code this if we couldn't make such an assumption.
            c. return the sub-list of all lower case consonants found in the input list in the order they were observed

b. **WRITEUP**: Best effort. Write at least one paragraph about what you learned when solving this problem. Not sure what to write? Looking for inspiration? This might* include something about flatMap, TDD, the value of relaxing requirements during the engineering process, or the value of policy abstraction. It might* also include the drawbacks to any of those items.

* you can write whatever you want here… it's an opinion piece

Q3. Continuation passing

3. Continuation passing modes can be convenient for writing code that solve certain types of pure problems faster. They are also strong for languages that don't particularly like head recursion but rather, have a preference toward tail recursion.
   a. **CODE:** Solve each of the following problems using continuation passing modes in your implementation
      i. **findByPolicyIff** : shall allow operations on 2 type objects, let's call them A and B. It shall take as input a list of type List[A]. It shall take a decision transpose policy of type (A) => Option[B]. The inputs shall be curried with the list before the policy. It shall return the option of a list of B (i.e. Option[List[B]]). Let \`b\` objects be defined as items of type B that are found by applying the input policy to an element of our input list and finding Some(b:B) rather than a None. This function shall use continuation passing to either return None without constructing any lists if no \`b\` objects are observed. This function shall return Some(list of \`b\` objects found in the order observed) if and only if there is at lease 1 \`b\` object observed.
      ii. getFirstFibSeqSTLenGeN (this one is rather challenging)
      (get_first_Fibonacci_sequenc_such_that_length_greater-that-or-equal-to_n)
      In L7Q3.scala I have provided the declaration of this function. It shall use a success and fail continuation in it's implementation. It shall require that a call to the function have an \`n\` input greater or equal to 2. Let 'outputList' be defined as the first contiguous sublist of integers from the input list s.t. outputList has length greater than or equal to n and the elements of the outputList are a fibbinochi sequence starting with values 1, 1,…. The function shall return outputList.

b. **WRITEUP**: Write at least one paragraph about what you learned when solving this problem. It can be positive and/or negative, this is an opinion piece.

Q4. Discussion

4. **WRITEUP**: Please discuss the content of this lab with your peers. Discussions can be very useful in helping us discover what we do and do not understand about a topic. Discussion can also help us develop a vocabulary to further discuss these materials in the future. Afterwards, please write at least one paragraph explaining what you found silly/helpful/tedious about this portion of the assignment.