Lab 4: Iterating Over Trees

Fall 2018

Contents

1 Objectives

By the end of this lab, you will:

- Understand and implement higher order functions on trees
- Learn about various traversal patterns over trees

2 Tree Traversal

When we "traverse" the nodes of a tree, we visit every node of that tree in a specified order. In this lab, we'll focus on three variants for traversing a tree:

- In-order traversal
- Pre-order traversal
- Post-order traversal

These traversals differ in the order in which they visit nodes in a tree.

2.1 In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

We can express this traversal algorithm with the following steps at every node:

1. Recursively traverse the left subtree.
2. Visit the root node.
3. Recursively traverse the right subtree.

Observe that, as we traverse a tree, we can put each value as we visit it into a `List`. This operation essentially allows us to transform a tree into a list. This isn't the first time we've seen something like this—we have many operations that transform a more complex data structure into a less complex one! We can use `reduce` or `fold` to transform lists into atomic data; tree traversals can turn trees into lists; graph traversals (breadth-first, depth-first, Dijkstra's algorithm, etc.) can turn graphs into trees.

Task: Consider a *binary search tree*. If we traverse a binary search tree in-order, and put each value as we visit it into a `List`, what property will the returned `List` have?

Before continuing, call over a TA to check that your answers are correct.

2.2 Pre-order Traversal

In this traversal method, the root node is visited first, then all of the subtrees of that node are traversed from left-to-right.

We can express this traversal algorithm with the following steps at every node:

1. Visit the root node.
2. Recursively traverse the subtrees from left-to-right.

Task: Assume the list `[list: "a", "b", "c", "d", "e", "f"]` was generated from a pre-order traversal of a binary tree. Draw a diagram of the tree this was created from. Is this the only possible tree?

2.3 Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

We can express this traversal algorithm with the following steps at every node:

1. Recursively traverse the subtrees from left-to-right.

2. Visit the root node.

Task: Assume the list `[list: 6, 5, 4, 3, 2, 1]` was generated from a post-order traversal of a binary tree. Draw a diagram of the tree this was created from. Is this the only possible tree?

Before continuing, call over a TA to check that your answers are correct.

3 Implementing Traversals

In the section on in-order traversal, we observed that we can use tree traversals to turn trees into lists. We'll now implement these traversal operations in Pyret.

3.1 Binary Tree Stencil

The stencil code for the binary tree sections of this lab can be found **at this link**.

3.2 Binary Tree Traversals

Consider the following data definition for a binary tree:

```
data BTree<T>:
  | mt
  | node(value :: T, left :: BTree<T>, right :: BTree<T>)
end
```

Task: Implement and write test cases for the following functions:

- `btree-in-order<A>(tree :: BTree<A>) -> List<A>`

  Returns a `List<A>` containing the elements in `tree` in the order they are visited by a in-order traversal.

- `btree-pre-order<A>(tree :: BTree<A>) -> List<A>`

  Returns a `List<A>` containing the elements in `tree` in the order they are visited by a pre-order traversal.

- `btree-post-order<A>(tree :: BTree<A>) -> List<A>`

  Returns a `List<A>` containing the elements in `tree` in the order they are visited by a post-order traversal.

Before continuing, call over a TA to check that your answers are correct.

4 Higher Order Functions on Binary Trees

In past assignments, we've explored using higher-order functions such as `map`, `filter`, and `fold` over lists. However, higher-order functions aren't just limited to lists.

In the second half of this lab, we'll extend those functions to binary trees and n-ary trees. Along the way, we'll use some of our previous work on tree traversals.

4.1 Setting Up the Functions

Consider the following data definition for a binary tree:

```
data BTree<T>:
  | mt
  | node(value :: T, left :: BTree<T>, right :: BTree<T>)
end
```

Task: With your partner, write the function signature for each of the following functions:

- `btree-map`—like `map` on `List`s, but for each value in a `BTree`s.
- `btree-filter`—like `filter` on `List`s, but for each value in a `BTree`s.
- `btree-fold`—like `fold` on `List`s, but for each value in a `BTree`s.

You should also be able to give a high-level description of what each function does. This can be a little tricky, since some of the above functions are purposely underspecified. You might find it helpful to work through a few examples for each function in order to determine whether or not there are any edge cases that it needs to handle in a certain way.

For example, for `btree-fold`, consider all of the possible `fold` directions on `List`s—how might this affect how we develop a `fold` operation for binary trees? (Your lab TAs will ask you about this!)

Before continuing, call over a TA to check that your answers are correct.

4.2 Implementation

You may have found that parts of the aforementioned functions were underspecified. Before continuing, let's solidify things a bit with the following specifications:

- `btree-map<A, B>(f :: (A -> B), tree :: BTree<A>) -> BTree<B>`

  Recursively applies `f` to the `value` of every `node` contained in `tree`.

- `btree-filter<A>(f :: (A -> Boolean), tree :: BTree<A>) -> BTree<A>`

  Recursively applies `f` to the `value` of every `node` and `leaf` in `tree`. If `f` returns `false` for a given `node` or `leaf`, that node and all of its children should be removed from the `tree`; otherwise, the returned `BTree<A>` should contain the given node.

- ```
  btree-fold<A, B>(
    f :: (A, B -> B),
    traversal :: (BTree<A> -> List<A>),
    base :: B,
    tree :: BTree<A>
  ) -> B
  ```

  Uses `traversal` to generate a `List<A>` of the values in `tree` in the order that they should be folded. Then, apply the function `f` over the generated `List<A>` from the left, starting with the initial value `base`.

  You should use the `btree-in-order`, `btree-pre-order`, and `btree-post-order` functions you previously defined when testing this function.

Task: Implement `btree-map`, `btree-filter`, and `btree-fold` in the binary tree stencil we provided you with in section 3.2. Make sure to develop a set of interesting and comprehensive test cases to check that your implementation is correct.

Call over a TA to check that your answers are correct.

⋆ Once a lab TA signs off on your work, you've finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you've just written.
  ```