Homework 3: Higher-Types / More on Trees

CIS 352: Programming Languages

2018-02-02

Administrivia

- Part I of the homework consists of a few simple problems about high-type functions. Part II builds on the binary search tree example in chapter 8 in LYAH.
- ***No teams of size larger than two!***
- If you pick up an idea from someone outside of your team or an internet site or a book, *document it* in your coversheet file.
- In your Blackboard submission, include: *(i)* your coversheet, *(ii)* your modified version of `trees.hs`, *(iii)* your modified version of `prop.hs`, *(iv)* the transcripts of test runs.

Grading Criteria

➤ The homework is out of 100 points.

➤ Each problem is, roughly, 70% correctness and 30% testing.

➤ Omitting your name(s) in the source code looses you 5 points.

Part I: Some higher-type exercises

Create a fresh file for the following definitions.

❖ ***Problem 1 (6 points)*** ❖

Using `filter`, write a function

```
rmChar :: Char -> String -> String
```

such that `(rmChar c str)` returns the result of removing all occurrences of `c` from the string `str`. **Do not use** the library functions `delete`, `elem`, \\, ... in you definition.

Testing: For `rmChar` devise your own tests.

Examples:

```
(rmChar 'x' "foo") ⇝ "foo"
(rmChar 'x' "fox") ⇝ "fo"
(rmChar 'x' "xfoxx") ⇝ "fo"
```

❖ ***Problem 2 (12 points)*** ❖

Write two functions

```
rmCharsRec :: String -> String -> String
rmCharsFold :: String -> String -> String
```

where both `(rmCharsRec s1 s2)` and `(rmCharsFold s1 s2)` remove all occurrences of characters from `s1` from `s2`. For `rmCharsRec` use a recursive definition and for `rmCharsFold` use a fold.

Testing: For `rmCharsRec` and `rmCharsFold` devise your own tests.

Examples:

```
(rmCharsRec "wx" "foo") ⇝ "foo"
(rmCharRec "ox" "fox") ⇝ "f"
(rmCharRec "ox" "qxfoxo") ⇝ "qf"
```

❖ ***Problem 3 (12 points)*** ❖

Write two functions

```
andRec :: [Bool] -> Bool
andFold :: [Bool] -> Bool
```

Examples:

```
(andRec []) ⇝ True
(andRec [True,True]) ⇝ True
(andRec [True,False]) ⇝ False
```

where both `(andRec bs)` and `(andFold bs)` checks whether every item in a list is `True`. For `andRec` use a recursive definition and for `andFold` use a fold. **Do not use** the library function `and` in you definition.

Testing for `rmCharsRec` and `rmCharsFold`: Devise your own tests.

❖ ***Problem 4 (12 points)*** ❖

Write a function

```
same :: [Int] -> Bool
```

that returns `True` if and only if all of the elements of the list are equal. Your definition should be of the form

```
same xs = and (zipWith ____ xs (tail xs))
```

To get an idea of what would be in the above blank, compute

```
zip [1..5] (tail [1..5])
```

Testing: For `same` devise your own tests.

Examples:

```
(same []) ⇝ True
(same [22]) ⇝ True
(same [5,5,5]) ⇝ True
(same [5,5,4]) ⇝ False
```

Part II: Binary search trees with indexing

Background. We consider binary search trees that have characters as their key values and which each node keeps tract of the number of elements in its *left* subtree. We shall call these things *ITrees*, for *indexed binary search trees*. *Example:* See Figure 1.

The *depth* of a node N in an ITree is the length of the path from the ITree's root node to N, e.g., in the ITree of Figure 1, the j-node is of depth 2.

The *index* of a node N in an ITree t is the number of nodes to the left of N in t. *Example:* For t_0 of Figure 1:

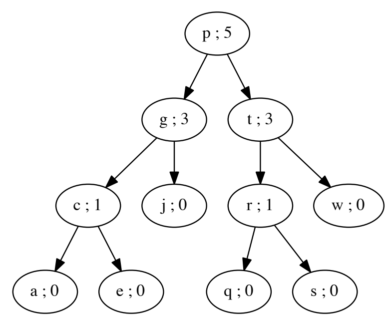

Figure 1: The ITree t_0.

node with character	a	c	e	g	j	p	q	r	s	t	w
the t_0-index of that node	0	1	2	3	4	5	6	7	8	9	10

Note/Hint: If t_1 is the t_0-subtree rooted at the `g`-node then, for each t_1 node N, the t_1-index of N = the t_0-index of N; whereas, if t_2 is the t_0-subtree rooted at the `t`-node, then, for each t_2 node N:

$$\text{the } t_2\text{-index of } N \;=\; (\text{the } t_0\text{-index of } N) - (5+1).$$

PROBLEMS. Use the starter file `ITree.hs` for the following.

The `ITree` data type is roughly based on the binary search tree example in chapter 8 of LYAH.

❖ ***Problem 5 (10 points)*** ❖
Write a function

```
treeInsert :: ITree -> Char -> ITree
```

such that `(treeInsert t c)` *(i)* if c is already in t, `treeInsert` reports an error; *(ii)* otherwise, `treeInsert` returns the result of adding c to t.[1] Your function should run in $O(h)$ time, where h is the height of t.

Testing: Use the QuickCheck properties `prop_treeInsert1` and `prop_treeInsert2`.[2]

[1] Additionally, the left subtree counts are updated as needed.

[2] I.e.: Run
```
quickCheck prop_treeInsert1
quickCheck prop_treeInsert2
```

❖ ***Problem 6 (16 points)*** ❖
Write a function

```
index :: ITree -> Char -> Int
```

such that `(index t c)` returns the index of c in t.[3] Your function should run in $O(d)$ time where d is the depth c's node in t.

Testing: Use the QuickCheck property `prop_index`.

[3] If c does not occur in t, an error should be reported.

❖ ***Problem 7 (16 points)*** ❖
Write a function

```
fetch :: ITree -> Int -> Char
```

such that `(fetch t n)` returns the character at index n in t.[4] Your function should run in $O(d)$ time where d is the depth in the tree of the node with index i.

Testing: Use the QuickCheck property `prop_fetch`.

[4] If there is no character at index n in t, then an error is reported.

❖ ***Problem 8 (16 points)*** ❖
Write a function

```
reroot :: ITree -> c -> ITree
```

such that `(reroot t c)` returns the result of altering t to make c's node the root.[5] Your function should run in $O(d)$ time where d is the depth of c's node in t. Moreover, your function should correctly update the left subtree element counts as needed. *Hint:* Tree rotations may be helpful, see Figure 2 and https://en.wikipedia.org/wiki/Tree_rotation.

Testing: Use the QuickCheck property `prop_reroot`.

[5] If c does not occur in t, then an error is reported.

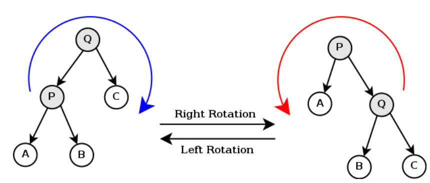

Figure 2: Left and right rotations.