Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Dr. Peter Lammich

SS 2018
4. 5. 2017

Functional Data Structures

Exercise Sheet 4

Exercise 4.1 List Elements in Interval

Write a function to in-order list all elements of a BST in a given interval. I.e., *in_range t u v* shall list all elements x with $u \le x \le v$. Write a recursive function that does not descend into nodes that definitely contain no elements in the given range.

fun *in_range* :: "$'a$::*linorder tree* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ *list*"

Show that you list the right set of elements

lemma "*bst t* $\Longrightarrow$ *set* (*in_range t u v*) = $\{x \in$ *set_tree t*. $u \le x \wedge x \le v\}$"

Show that your list is actually in-order

lemma "*bst t* $\Longrightarrow$ *in_range t u v* = *filter* ($\lambda x.\ u \le x \wedge x \le v$) (*inorder t*)"

Exercise 4.2 Pretty Printing of Binary Trees

Define a function that checks whether two binary trees have the same structure. The values at the nodes may differ.

fun *bin_tree2* :: "$'a$ *tree* $\Rightarrow$ $'b$ *tree* $\Rightarrow$ *bool*"

While this function itself is not very useful, the induction rule generated by the function package is! It allows simultaneous induction over two trees:

print_statement *bin_tree2.induct*

Binary trees can be uniquely pretty-printed by emitting a symbol L for a leaf, and a symbol N for a node. Each N is followed by the pretty-prints of the left and right tree. No additional brackets are required!

datatype $'a$ *tchar* = *L* | *N* $'a$

fun *pretty* :: "$'a$ *tree* $\Rightarrow$ $'a$ *tchar list*"

Show that pretty-printing is actually unique, i.e., no two different trees are pretty-printed the same way. Hint: Auxiliary lemma. Simultaneous induction over both trees.

lemma *pretty_unique*: *"pretty t = pretty t' ⟹ t=t'"*

Exercise 4.3 Enumeration of Trees

Write a function that generates the set of all trees up to a given height. Show that only trees up to the specified height are contained.

(The other direction, i.e., that all trees are contained, requires an advanced case split, which has not yet been introduced in the lecture, so it is omitted here)

fun *enum* :: *"nat ⇒ unit tree set"* **where**
lemma *enum_sound*: *"t ∈ enum n ⟹ height t ≤ n"*

Homework 4 Rank Annotated Trees

Submission until Friday, May 11, 11:59am.

In this homework, we will develop a binary search tree that additionally stores the rank (= number of nodes) of the left subtree in each node.

With this auxiliary information, it is easy to implement a rank query, i.e., to return the position of a given element in the inorder traversal.

datatype *'a rtree = Leaf | Node "'a rtree" nat 'a "'a rtree"*

Define a function to count the number of nodes in a tree

fun *num_nodes* :: *"'a rtree ⇒ nat"* **where**

Define a function to check for the invariant: search tree property and the correct rank annotation (number of nodes in left subtree)

fun *rbst* :: *"'a::linorder rtree ⇒ bool"* **where**

Define the insert function. You may assume that the value to be inserted is not contained in the tree. Note: Double-check to correctly update the rank annotation.

fun *rins* :: *"'a::linorder ⇒ 'a rtree ⇒ 'a rtree"* **where**

Show that *rins* actually inserts, and preserves the invariant. Hint: Auxiliary lemma on number of nodes.

lemma *rins_set*: *"set_rtree (rins x t) = insert x (set_rtree t)"*
lemma *"x∉set_rtree t ⟹ rbst t ⟹ rbst (rins x t)"*

Define the membership query function and show it correct.

fun *risin* :: *"'a::linorder ⇒ 'a rtree ⇒ bool"* **where**

lemma *"rbst t ⟹ risin x t ⟷ x∈set_rtree t"*

Define the inorder traversal

fun *inorder* :: *"'a rtree ⇒ 'a list"* **where**

Define a function that returns the rank of an element. Use the rank annotation to avoid unnecessary descents into the tree.

Note: You may assume that the element is contained in the tree.

fun *rank* :: *"'a::linorder ⇒ _"* **where**

The operator *op* !::*'a list ⇒ nat ⇒ 'a* indexes a list, i.e., $l!n$ is the nth element of list l, or *undefined*, if the index is out of bounds. The following predicate states that index i into list l contains element x

definition *"at_index i l x ≡ i<length l ∧ l!i=x"*

Show your rank function correct. Hint: Auxiliary lemma relating *num_nodes* and *inorder*.

lemma *"rbst t ⟹ x∈set_rtree t ⟹ at_index (rank x t) (inorder t) x"*

Define a select function, that returns the ith element of the inorder traversal, and prove it correct.

Only recurse over the tree once, following a single path. In particular, *inorder t ! i* is not the desired solution, as it would enumerate all nodes of the tree in a list first, and not exploit the rank annotations at all.

fun *select* :: *"nat ⇒ 'a::linorder rtree ⇒ 'a"* **where**
lemma *select_correct*: *"rbst t ⟹ i<length (inorder t) ⟹ select i t = inorder t ! i"*

For 3 **bonus** points, remove the assumption that the inserted element is not yet contained in the tree. Only recurse over the tree once, i.e., do not simply use *if risin x t then t else rins x t*!

Hint: Add an additional return value to the insert function that indicates whether the element was in the tree or not, in order to correctly update the rank annotation. At the end, you must provide a function *rins'* that satisfies the following specification (and only recurses over the tree once, following a single path):

definition *rins'* :: *"'a::linorder ⇒ 'a rtree ⇒ 'a rtree"*
lemma *rins'_set*: *"rbst t ⟹ set_rtree (rins' x t) = {x} ∪ set_rtree t"*
lemma *rins'_bst*: *"rbst t ⟹ rbst (rins' x t)"*