Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

SS 2022
13. 5. 2022

Functional Data Structures

Exercise Sheet 3

Exercise 3.1 Membership Test with Less Comparisons

In the worst case, the *isin* function performs two comparisons per node. In this exercise, we want to reduce this to one comparison per node. The idea is that we never test for >, but always goes right if not <. However, one remembers the value where one should have tested for =, and performs the comparison when a leaf is reached.

fun *isin2* :: "('a::linorder) tree ⇒ 'a option ⇒ 'a ⇒ bool"
— The second parameter stores the value for the deferred comparison

Show that your function is correct.
Hint: Auxiliary lemma for *isin2 t (Some y) x* !

lemma *isin2_None*:
"bst t ⟹ isin2 t None x = isin t x"

Exercise 3.2 Height-Preserving In-Order Join

Write a function that joins two binary trees such that

- The in-order traversal of the new tree is the concatenation of the in-order traversals of the original trees
- The new tree is at most one higher than the highest original tree

  Hint: Once you got the function right, proofs are easy!

fun *join* :: "'a tree ⇒ 'a tree ⇒ 'a tree"

lemma *join_inorder*[*simp*]: "inorder(join t1 t2) = inorder t1 @ inorder t2"

lemma "height(join t1 t2) ≤ max (height t1) (height t2) + 1"

Exercise 3.3 Implement Delete

Implement delete using the *join* function from last exercise.

Note: At this point, we are not interested in the implementation details of join any more, but just in its properties, i.e. what it does to trees. Thus, as first step, we declare its equations to not being automatically unfolded.

declare *join.simps*[*simp del*]

Both *set_tree* and *bst* can be expressed by the inorder traversal over trees:

thm *set_inorder*[*symmetric*] *bst_iff_sorted_wrt_less*

Note that *set_inorder* is declared as simp. Be careful not to have both directions of the lemma in the simpset at the same time, otherwise the simplifier is likely to loop.

You can use *simp del*: *set_inorder add*: *set_inorder*[*symmetric*] to temporarily remove the first direction of the lemma from the simpset.

Alternatively, you can write *declare set_inorder*[*simp del*] to remove it once and forall.

For *bst*, you might want to delete the *bst_wrt* simps, and use the append lemma:

thm *bst_wrt.simps*
thm *sorted_wrt_append*

Show that join preserves the set of entries

lemma *join_set*[*simp*]: *"set_tree (join t1 t2) = set_tree t1 ∪ set_tree t2"*

Show that joining the left and right child of a BST is again a BST:

lemma *bst_pres*[*simp*]: *"bst (Node l (x::_::linorder) r) ⟹ bst (join l r)"*

Implement a delete function using the idea contained in the lemmas above.

fun *delete* :: *"'a::linorder ⇒ 'a tree ⇒ 'a tree"*

Prove it correct! Note: You'll need the first lemma to prove the second one!

lemma *bst_set_delete*[*simp*]: *"bst t ⟹ set_tree (delete x t) = (set_tree t) − {x}"*

lemma *bst_del_pres*: *"bst t ⟹ bst (delete x t)"*

Homework 3.1 Tree Addressing

Submission until Thursday, May 19, 23:59pm.

A position in a tree can be given as a list of navigation instructions from the root, i.e. whether to go to the left or right subtree. We call such a list a path.

datatype *direction* $= L \mid R$
type_synonym *path* = *"direction list"*

Define when a path is valid:

fun *valid* :: *"'a tree ⇒ path ⇒ bool"*

Define a function *delete_substree* t p", that returns "t", with the subtree at "p" replaced with a leaf.

fun *delete_subtree* :: *"'a tree ⇒ path ⇒ 'a tree"*

Define the function such that nothing happens if an invalid path is given. Prove the following for *delete_subtree*:

lemma *delete_subtree_invalid*: *"¬valid t p ⟹ delete_subtree t p = t"*

Similarly define two functions, the first "get t p" to return the subtree of "t" addressed by a given path, and a second one "put t p s", that returns *t*, with the subtree at *p* replaced by *s*. The function "get" should return "undefined" if the path is not valid and "put" should do nothing if the path is not valid.

fun *get* :: *"'a tree ⇒ path ⇒ 'a tree"*
fun *put* :: *"'a tree ⇒ path ⇒ 'a tree ⇒ 'a tree"*

Prove the following algebraic laws on "delete", "put", and "get".

lemma *put_in_delete*: *"put (delete_subtree t p) p (get t p) = t"*

lemma *delete_delete*: *"valid t p ⟹ delete_subtree (delete_subtree t p) p = delete_subtree t p"*

lemma *delete_replaces_with_leaf[simp]*: *"valid t p ⟹ get (delete_subtree t p) p = Leaf"*

lemma *valid_delete*: *"valid t p ⟹ valid (delete_subtree t p) p"*

Show the following lemmas about appending two paths:

lemma *valid_append*: *"valid t (p@q) ⟷ valid t p ∧ valid (get t p) q"*

lemma *put_delete_get_append*:
"valid t (p@q) ⟹ delete_subtree t (p@q) = put t p (delete_subtree (get t p) q) "

lemma *put_get_append*:
"valid t (p@q) ⟹ get (put t (p@q) s) p = put (get t p) q s"

Homework 3.2 Implementing a map using binary trees

Submission until Thursday, May 19, 23:59pm.

A map is a collection of (key, value) pairs, where each possible key appears at most once. For this datatype, one should be able add/update a (key,value) pair, delete a pair, and lookup a value associated with a particular key.

A straightforward implementation of maps can be done using association lists. An existing such implementation uses the functions *upd_list*, *del_list*, and *AList_Upd_Del.map_of* for add/update a (key,value) pair, deleting a pair, and looking up values, respectively.

thm *map_of.simps upd_list.simps del_list.simps*

HINT: to prove facts concerning *del_list*, *upd_list*, or *AList_Upd_Del.map_of* you might find it useful to use *del_list_simps*, *upd_list_simps*, or *map_of_simps* as simp rules. Each one of those is a set of lemmas proven about the respective functions.

thm *del_list_simps*
thm *upd_list_simps*
thm *map_of_simps*

For linearly ordered keys, it is more efficient to implement maps using binary trees. Using binary search trees, implement the three main map functionalities add/update a (key,value) pair, deleting a pair, and looking up values. Then you should prove their equivalence to the corresponding association list implementation of maps.

The first one of those functionalities is map lookup. It should return an option type, i.e. for a key k it should return *Some* v if the map has an entry for k, and *None* otherwise.

fun *map_lookup* :: *“('a::linorder∗'b) tree ⇒ 'a ⇒ 'b option”*

Prove it is equivalent to *AList_Upd_Del.map_of*, if the tree is well-formed (i.e an inorder traversal of its elements returns a sorted list).

HINT: for proving facts about objects of type option, it is useful to use *option.split* as a split rule for ”auto” (check the usage of ”split: tree.split” in the exercise).

lemma *lookup_map_of*:
“sorted1(inorder t) ⟹ map_lookup t x = map_of (inorder t) x”

The second functionality is map update.

fun *map_update* :: *“'a::linorder ⇒ 'b ⇒ ('a∗'b) tree ⇒ ('a∗'b) tree”*

Prove it is equivalent to *upd_list*.

lemma *inorder_update*:
“sorted1(inorder t) ⟹ inorder(map_update a b t) = upd_list a b (inorder t)”

Lastly, define a function that, given a key k, deletes key-value pair (k, v) from a map represented as a tree, if (k, v).

HINT: You can use the function *split_min* defined in the lecture demonstration of trees.

fun *map_delete* :: *“'a::linorder ⇒ ('a∗'b) tree ⇒ ('a∗'b) tree”*

Prove that it works as intended.

HINT: you will need to prove a lemma about *split_min*.

lemma *inorder_delete*:
"$sorted1(inorder\ t) \Longrightarrow inorder(map\_delete\ x\ t) = del\_list\ x\ (inorder\ t)$"