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

SS 2018
27. 4. 2018

Functional Data Structures

Exercise Sheet 3

Exercise 3.1 Membership Test with Less Comparisons

In worst case, the *isin* function performs two comparisons per node. In this exercise, we want to reduce this to one comparison per node, with the following idea:

One never tests 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 tree
- 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 specification, 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: As *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*], or *auto simp del*: *set_inorder simp*: *set_inorder*[*symmetric*] to temporarily remove the lemma from the simpset.
Alternatively, you can write *declare set_inorder*[*simp_del*] to remove it once and forall.

For the *sorted_wrt* predicate, you might want to use these lemmas as simp:

thm *sorted_wrt_append sorted_wrt_Cons*

Show that join preserves the set of entries

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

lemma *"bst t ⟹ bst (delete x t)"*

Homework 3.1 Tree Addressing

Submission until Friday, May 4, 11:59am.

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* | *R*
type_synonym *path* = *"direction list"*

Specify when a path is valid:

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

Specify a function to return the subtree addressed by a given path:

fun *get* :: *"'a tree ⇒ path ⇒ 'a tree"*
| *"get _ _ = undefined"* — Catch-all clause to get rid of missing patterns warning

Specify a function *put t p s*, that returns *t*, with the subtree at *p* replaced by *s*.

fun *put* :: "$'a$ *tree* $\Rightarrow$ *path* $\Rightarrow$ $'a$ *tree* $\Rightarrow$ $'a$ *tree*"

Specify your function such that it does nothing if an invalid path is given, and prove:

lemma *put_invalid*: "$\neg valid\ t\ p \Longrightarrow put\ t\ p\ s = t$"

Note: this convention will simplify some of the lemmas, reducing the required validity preconditions.

Prove the following algebraic laws on *put* and *get*. Add preconditions of the form *valid t p* where needed!

lemma *get_put*[*simp*]: "$put\ t\ p\ (get\ t\ p) = t$"

lemma *put_put*[*simp*]: "$put\ (put\ t\ p\ s)\ p\ s' = put\ t\ p\ s'$"

lemma *put_get*[*simp*]: "$get\ (put\ t\ p\ s)\ p = s$"

lemma *valid_put*[*simp*]: "$valid\ (put\ t\ p\ s)\ p$"

Show the following lemmas about appending two paths:

lemma *valid_append*[*simp*]: "$valid\ t\ (p@q) \longleftrightarrow valid\ t\ p \wedge valid\ (get\ t\ p)\ q$"

lemma *get_append*[*simp*]: "$valid\ t\ p \Longrightarrow get\ t\ (p@q) = get\ (get\ t\ p)\ q$"

lemma *put_append*[*simp*]: "$put\ t\ (p@q)\ s = specify\_a\_meaningful\_term\_here$"

Homework 3.2 Remdups

Submission until Friday, May 4, 11:59am.

Your task is to write a function that removes duplicates from a list, using a BST to efficiently store the set of already encountered elements.

You may want to start with an auxiliary function, that takes the BST with the elements seen so far as additional argument, and then define the actual function.

fun *bst_remdups_aux* :: "$'a$::*linorder tree* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list*"
definition "$bst\_remdups\ xs \equiv bst\_remdups\_aux\ Leaf\ xs$"

Show that your function preserves the set of elements, and returns a list with no duplicates (predicate *distinct* in Isabelle). Hint: Generalization!

lemma "$set\ (bst\_remdups\ xs) = set\ xs$"
lemma "$distinct\ (bst\_remdups\ xs)$"

A list *xs* is a sublist of *ys*, if *xs* can be produced from *ys* by deleting an arbitrary number of elements.

Define a function *sublist xs ys* to check whether *xs* is a sublist of *ys*.

fun *sublist* :: "'a list ⇒ 'a list ⇒ bool"

Show that your remdups function produces a sublist of the original list!
Hint: Generalization. Auxiliary lemma required.

lemma "sublist (bst_remdups xs) xs"