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

SS 2017
5. 5. 2017

Functional Data Structures

Exercise Sheet 2

Exercise 2.1 Folding over Trees

Define a datatype for binary trees that store data only at leafs.

datatype *'a ltree* =

Define a function that returns the list of elements resulting from an in-order traversal of the tree.

fun *inorder* :: *"'a ltree ⇒ 'a list"*

Have a look at Isabelle/HOL's standard function *fold*.

thm *fold.simps*

In order to fold over the elements of a tree, we could use *fold f (inorder t) s*. However, from an efficiency point of view, this has a problem. Which?

Define a more efficient function *fold_ltree*, and show that it is correct

fun *fold_ltree* :: *"('a ⇒ 's ⇒ 's) ⇒ 'a ltree ⇒ 's ⇒ 's"*
lemma *"fold f (inorder t) s = fold_ltree f t s"*

Define a function *mirror* that reverses the order of the leafs, i.e., that satisfies the following specification:

lemma *"inorder (mirror t) = rev (inorder t)"*

Exercise 2.2 Shuffle Product

To shuffle two lists, we repeat the following step until both lists are empty: Take the first element from one of the lists, and append it to the result.
That is, a shuffle of two lists contains exactly the elements of both lists in the right order.
Define a function *shuffles* that returns a list of all shuffles of two given lists

fun *shuffles* :: *"'a list ⇒ 'a list ⇒ 'a list list"*

Show that the length of any shuffle of two lists is the sum of the length of the original lists.

lemma *"l∈set (shuffles xs ys) ⟹ length l = length xs + length ys"*

Note: The *set* function converts a list to the set of its elements.

Exercise 2.3 Fold function

The fold function is a very generic function, that can be used to express multiple other interesting functions over lists.

Write a function to compute the sum of the elements of a list. Specify two versions, one direct recursive specification, and one using fold. Show that both are equal.

fun *list_sum* :: *"nat list ⇒ nat"*
definition *list_sum'* :: *"nat list ⇒ nat"*
lemma *"list_sum l = list_sum' l"*

Homework 2.1 Distinct lists

Submission until Friday, May 12, 11:59am. Submit your solution via https://vmnipkow3.in.tum.de. Submit a theory file that runs in Isabelle-2016-1 **without errors**.

Define a function *contains*, that checks whether an element is contained in a list. Define the function directly, not using *set*.

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

Define a predicate *ldistinct* to characterize *distinct* lists, i.e., lists whose elements are pairwise disjoint. Hint: Use the function contains.

fun *ldistinct* :: *"'a list ⇒ bool"*

Show that a reversed list is distinct if and only if the original list is distinct. Hint: You may require multiple auxiliary lemmas.

lemma *"ldistinct (rev xs) ⟷ ldistinct xs"*

Homework 2.2 More on fold

Submission until Friday, May 12, 11:59am.

Isabelle's fold function implements a left-fold. Additionally, Isabelle also provides a right-fold *foldr*.

Use both functions to specify the length of a list.

thm *fold.simps*

thm *foldr.simps*

definition *length_fold* :: *"'a list ⇒ nat"*

definition *length_foldr* :: *"'a list ⇒ nat"*

lemma *"length_fold l = length l"*
lemma *"length_foldr l = length l"*

Homework 2.3 List Slices

Submission until Friday, May 12, 11:59am. Specify a function *slice xs s l*, that, for a list $xs=[x_0,...,x_n]$ returns the slice starting at s with length l, i.e., $[x_s,...,x_{s+len-1}]$.
If *s* or *len* is out of range, return a shorter (or the empty) list.

fun *slice* :: *"'a list ⇒ nat ⇒ nat ⇒ 'a list"*
  where

Hint: Use pattern matching instead of *if*-expressions. For example, instead of writing *f x = (if x>0 then ... else ...)* you should define two equations *f 0 = ...* and *f (Suc n) = ...*.

Some test cases, which should all hold, i.e., yield *True*

value *"slice [0,1,2,3,4,5,6::int] 2 3 = [2,3,4]"* — In range
value *"slice [0,1,2,3,4,5,6::int] 2 10 = [2,3,4,5,6]"* — Length out of range
value *"slice [0,1,2,3,4,5,6::int] 10 10 = []"* — Start index out of range

Show that concatenation of two adjacent slices can be expressed as a single slice:

lemma *"slice xs s l1 @ slice xs (s+l1) l2 = slice xs s (l1+l2)"*

Show that a slice of a distinct list is distinct.

lemma *"ldistinct xs ⟹ ldistinct (slice xs s l)"*