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

SS 2021
23. 4. 2021

Functional Data Structures

Exercise Sheet 2

Exercise 2.1 Fold function

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

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

thm *fold.simps*

Recall the function to compute the sum of the elements of a list from the last homework. Define that function using fold, and show that both are equal.

```
fun listsum :: "nat list ⇒ nat" where
  "listsum [] = 0"
| "listsum (x#xs) = x + listsum xs"

definition listsum' :: "nat list ⇒ nat"
lemma "listsum xs = listsum' xs"
```

Exercise 2.2 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"
```

In order to fold over the elements of a tree, we could use *fold f (inorder t) s*.

Define a function *fold_ltree* that is recursive on the structure of the tree, and that returns the same result as *fold f (inorder t) s*.

```
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 2.3 Shuffle Product

A shuffle of two lists, *xs* and *ys*, is a list that contains exactly the elements of *xs* and *ys* s.t. every two elements $x \in xs$ (resp. *ys*) and $x' \in xs$ (resp. *ys*) occur in the shuffle in the same order they do in *xs* (resp. *ys*).
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 *"zs ∈set (shuffles xs ys) ⟹ length zs = length xs + length ys"*

Homework 2.1 Association Lists

Submission until Thursday, April 29, 23:59pm.

An association list is a list of pairs. An entry (k, v) means that key k is associated to value v.
For an association list xs, the *collect k xs* operation returns a list of all values associated to key k, in the order stored in the list. Specify the function collect by a set of recursion equations:

fun *collect* :: *"'a ⇒ ('a × 'b) list ⇒ 'b list"*

Test cases

value *"collect (2::nat) [(2,3::int),(4,4),(2,5),(2,7),(3,0)] = [3,5,7]"*
value *"collect (1::nat) [(2,3::int),(4,4),(2,5),(2,7),(3,0)] = []"*

An experienced functional programmer might also write this function as

map snd (filter (λkv. fst kv = x) ys)

Show that this specifies the same function:

lemma *collect_functional*: *"collect x ys = map snd (filter (λkv. fst kv = x) ys)"*

Note that Isabelle pretty-prints the filter function to *[kv←ys . fst kv = x]*, which is just a pretty-printing conversion, but does not change the term in the underlying logic.

When the lists get bigger, efficiency might be a concern. To avoid stack overflows, you might want to specify a tail-recursive version of *collect*. The first parameter is the accumulator, that accumulates the elements to be returned, and is returned at the end.

Note: To avoid appending to the accumulator, we accumulate the elements in reverse order, and reverse the accumulator at the end.

Complete the second equation!

fun *collect_tr* :: *"'a list ⇒ 'b ⇒ ('b × 'a) list ⇒ 'a list"* **where**
  "collect_tr acc x [] = rev acc"

Show correctness of your tail-recursive version. Hint: Generalization!

lemma *collect_tr_correct*: *"collect_tr [] x ys = collect x ys"*

Homework 2.2 Complete Trees

Submission until Thursday, Apr 29, 23:59pm.

Recall the tree datatype *'a ltree* from the tutorial. Define functions to return the height (A leaf has height 0) and the number of leafs:

fun *lheight* :: *"'a ltree ⇒ nat"*
fun *num_leafs* :: *"'a ltree ⇒ nat"*

A tree is complete iff, for each node, the left and right subtree have the same height. Specify a function to check that a tree is complete.

fun *complete* :: *"'a ltree ⇒ bool"*

Show that, for a complete tree with height h and number of leafs l, we have $l = 2^h$:

theorem *complete_size*: *"complete t ⟹ num_leafs t = 2^lheight t"*

Homework 2.3 Bonus: Delta-Encoding

Submission until Thursday, Apr 29, 23:59pm.

This is a bonus homework, worth 4 bonus points.

(When computing your homework performance as a percentage, bonus points will only count on your side, but not towards the total score.)

We want to encode a list of integers as follows: The first element is unchanged, and every next element only indicates the difference to its predecessor.

For example: (Hint: Use this as test cases for your spec!)

enc [1,2,4,8] = [1,1,2,4]

enc [3,4,5] = [3,1,1]

enc [5] = [5]

enc [] = []

Background: This algorithm may be used in lossless data compression, when the difference between two adjacent values is expected to be small, e.g., audio data, image data, sensor data.

It typically requires much less space to store the small deltas, than the absolute values.

Disadvantage: If the stream gets corrupted, recovery is only possible when the next absolute value is transmitted. For this reason, in practice, one will submit the current absolute value from time to time. (This is not modeled in this exercise!)

Specify a function to encode a list with delta-encoding. The first argument is used to represent the previous value, and can be initialized to 0.

fun *denc* :: *"int ⇒ int list ⇒ int list"*

Specify the decoder. Again, the first argument represents the previous decoded value, and can be initialized to 0.

fun *ddec* :: *"int ⇒ int list ⇒ int list"*

Show that encoding and then decoding yields the same list. HINT: The lemma will need generalization.

theorem *denc_enc_id*: *"ddec 0 (denc 0 l) = l"*