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

SS 2021
21. 5. 2020

Functional Data Structures

Exercise Sheet 6

Exercise 6.1 Complexity of Naive Reverse

Show that the naive reverse function needs quadratically many *Cons* operations in the length of the input list. (Note that $[x]$ is syntax sugar for *Cons* x []!)

thm *append.simps*

fun *reverse* **where**
  "reverse [] = []"
| *"reverse (x#xs) = reverse xs @ [x]"*

Exercise 6.2 Selection Sort

Selection sort (also known as MinSort) sorts a list by repeatedly moving the smallest element of the remaining list to the front.

Define a function that takes a non-empty list, and returns the minimum element and the list with the minimum element removed

fun *find_min* :: *"'a::linorder list ⇒ 'a × 'a list"*

Show that *find_min* returns the minimum element

lemma *find_min_min*:
  assumes *"find_min xs = (y,ys)"*
    and *"xs≠[]"*
  shows *"a∈set xs ⟹ y ≤ a"*

Show that *find_min* returns exactly the elements from the list

lemma *find_min_mset*:
  assumes *"find_min (x#xs) = (y,ys)"*
  shows *"mset (x#xs) = (mset (y#ys))"*

Show the following lemma on the length of the returned list, and register it as [*termination_simp*]. The function package will require this to show termination of the selection sort function.

lemma *find_min_snd_len_decr*[*termination_simp*]:
  assumes *"(y,ys) = find_min (x#xs)"*
  shows *"length ys < Suc (length xs)"*

Selection sort can now be written as follows:

fun *sel_sort* **where**
  "sel_sort [] = []"
| *"sel_sort xs = (let (y,ys) = find_min xs in y#sel_sort ys)"*

Show that selection sort is a sorting algorithm:

lemma *sel_sort_mset*[*simp*]: *"mset (sel_sort xs) = mset xs"*

lemma *"sorted (sel_sort xs)"*

Homework 6.1 Cost of Selection Sort

Submission until Thursday, May 27, 23:59pm. Recall the selection sort from the tutorial (which can be found in the *Defs*).

Define cost functions for the number of comparisons of *sel_sort*. For if/else, over-estimate the cost by always choosing the more expensive branch.

fun *T_find_min* :: *"'a::linorder list ⇒ nat"*
fun *T_sel_sort* :: *"'a::linorder list ⇒ nat"*
lemma *T_find_min_cmpx*: *"xs ≠ [] ⟹ T_find_min xs = length xs − 1"*

Try to find a closed formula for *T_sel_sort* yourself! (Hint: Should be $O(n^2)$)

If you struggle with finding a closed formula, on paper:

- Put up a recurrence equation (depending only on the length of the list)
- Solve the equation (Assume that the solution is an order-2 polynomial)

theorem *T_sel_sort_cmpx*: *"T_sel_sort xs = undefined"*

Homework 6.2 Sorting Networks

Submission until Thursday, May 27, 23:59pm.

Comparison networks are a model of parallel algorithms on fixed-size lists. A sorting network is a specific comparison network that sorts its input lists.

A comparison network can be viewed as set of *wires* x_i, one for each list element. Between those wires are a number of *comparators* c_i; each comparator is connected to two wires. For Example (lists of size three):

```
x0---[]--------[]----
     | c0      | c2
x1---[]---[]---[]----
          | c1
x2--------[]---------
```

Each comparator will shift the greater element of its inputs up, and the smaller element down.

We represent a network by a list of comparators, where each comparator is characterized by the index of its wires – i.e., c_0=(*0*,*1*), and after the applying c_0, the greater element will be at position of x_1.

type_synonym *comparator* = "(*nat* × *nat*)"
type_synonym *compnet* = "*comparator list*"

Write a function to perform the computation of a single comparator on a *'a list*. If the comparator would compare elements out of the range of the input list, return the input unchanged.

Hint: Use the existing *list_update* and *nth* functions. *list_update* also has nice snytax: *xs*[*0* := *1*, *1* := *2*]

definition *compnet_step* :: "*comparator* ⇒ *'a* :: *linorder list* ⇒ *'a list*"

Some test cases:

value "*compnet_step* (*1*,*100*) [*1*,*2*::*nat*] = [*1*,*2*]"
value "*compnet_step* (*1*,*2*) [*1*,*3*,*2*::*nat*] = [*1*,*2*,*3*]"

The whole network operation is now a step-wise fold over the comparators:

definition *run_compnet* :: "*compnet* ⇒ *'a* :: *linorder list* ⇒ *'a list*" **where**
"*run_compnet* = *fold compnet_step*"

Start by proving that compnets keep the *mset* unchanged.

theorem *compnet_mset*[*simp*]: "*mset* (*run_compnet comps xs*) = *mset xs*"

Sortedness is a bit more difficult. Define a sorting net for lists of length 4 first. Use at most five comparators!

definition *sort4* :: *compnet*
value "*length sort4* ≤ *5*"
value "*run_compnet sort4* [*4*,*2*,*1*,*3*::*nat*] = [*1*,*2*,*3*,*4*]"

We want to prove that this definition is correct:

lemma "*length ls* = *4* ⟹ *sorted* (*run_compnet sort4 ls*)"
oops

However, doing that directly is not easily possible. But we can easily prove that it sorts boolean lists, since there is only a finite number of those.

We use the *all_n_lists* to obtain a version of the lemma that doesn't contain any free variables, so that *eval* can prove it exhaustively. Then we show that this holds when stated in the more obvious way.

lemma *sort4_bool_exhaust*: *"all_n_lists (λbs::bool list. sorted (run_compnet sort4 bs)) 4"*
— Should be provable *by eval* if your definition is correct!

lemma *sort4_bool*: *"length (bs::bool list) = 4 ⟹ sorted (run_compnet sort4 bs)"*
using *sort4_bool_exhaust[unfolded all_n_lists_def] set_n_lists* **by** *fastforce*

From that, we can show that our networks sorts any list – this is known as the *zero-one principle*. First prove that the sorting does not change when mapped with a monotone function (ctrl+click to see the definition of *mono*).

3 bonus points if you don't use *sledgehammer*ed proof steps (i.e., using *metis*, *smt*, *meson*, or *moura*) in the lemma or any required auxiliary theorem! To claim those points, mark the lemma with (∗ *clean* ∗).

lemma *compnet_map_mono*:
assumes *"mono f"*
shows *"run_compnet cs (map f xs) = map f (run_compnet cs xs)"*

Now prove the zero-one principle.

Hint: Proof by contradiction. If you are stuck, look for a proof on paper in existing literature!

theorem *zero_one_principle*:
assumes *"⋀bs:: bool list. length bs = length xs ⟹ sorted (run_compnet cs bs)"*
shows *"sorted (run_compnet cs xs)"* (**is** *"sorted ?rs"*)

Finally, sortedness of the *sort4* net follows (for any type).

corollary *"length xs = 4 ⟹ sorted (run_compnet sort4 xs)"*
by (*simp add*: *sort4_bool zero_one_principle*)