Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Simon Roßkopf

SS 24
7. 6. 2023

Functional Data Structures

Exercise Sheet 7

Exercise 7.1 Complexity of Naive Reverse

Show that the naive reverse function has quadratic running time in the length of the input list. Use the command *time_fun* to generate the running time functions

Hint: Show an equality rather than an inequality.

thm *append.simps*

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

Exercise 7.2 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 *split_min* :: *"'a::linorder list ⇒ 'a × 'a list"*

Show that *split_min* returns the minimum element

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

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

lemma *split_min_mset*:
  assumes *"split_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 *split_min_snd_len_decr*[*termination_simp*]:
  assumes *"(y,ys) = split_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) = split_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 7.1 Cost of Selection Sort

Submission until Thursday, June 15, 23:59pm. Recall the selection sort from the tutorial (which can be found in the *Defs*).
The running time functions for *split_min*/*sel_sort* are already defined using the *time_fun* command (in *Defs*).

Show the following closed form for *T_split_min*:

lemma *T_split_min_cmpx*: *"xs ≠ [] ⟹ T_split_min xs = length xs"*

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, you could try:

- Look at the first few values of *T_sel_sort*
- Put up a recurrence equation (depending only on the length of the list) and solve it

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

Homework 7.2 Quicksort runtime complexity

Submission until Thursday, June 15, 23:59pm.
Prove that quicksort does at most a number of comparisons that is at most the square of the length of the given list. The following is a cost function for the number of comparisons of quicksort:

fun *C_qsort* :: *"'a::linorder list ⇒ nat"* **where**
  "C_qsort [] = 0"
| *"C_qsort (p#xs)*

$= C\_qsort\ (filter\ (\lambda x.\ x < p)\ xs) + C\_qsort\ (filter\ (\lambda x.\ x \geq p)\ xs) + 2*length\ xs$"

Show that the number of required comparisons is at most $(length\ xs)^2$.

Hints:

- Do an induction on the length of the list (*length_induct*), and, afterwards, a proof by cases on the list constructors.
- Note that for natural numbers $a^2+b^2 \leq (a+b)^2$
- Have a look at the lemma *sum_length_filter_compl*

lemma *C_qsort_bound*: "$C\_qsort\ xs \leq (length\ xs)^2$"

Homework 7.3 Pancake sorting

Submission until Thursday, June 15, 23:59pm.

Pancake sorting (https://en.wikipedia.org/wiki/Pancake_sorting)/sorting by prefix reversal is a special kind of sorting problem, in which the only operation allowed to modify the list is to reverse some prefix of the list.

In this exercise, you should develop an algorithm that sorts a list this way, using a linear number of reversals (in the length of the list) and prove that it is a sorting algorithm.

First, define a function to perform a prefix-reversal and show that it preserves the multiset of its elements. *rev_pre n xs* should reverse the order of the first n elements of *xs*. If $n \geq length\ xs$, it should reverse the entire list.

fun *rev_pre*:: "$nat \Rightarrow 'a\ list \Rightarrow 'a\ list$"

lemma *mset_rev_pre*[*simp*]: "$mset\ (rev\_pre\ n\ xs) = mset\ xs$"

Based on this definition, define a function which moves the biggest element in a list to the end using exactly two prefix-reversals. If there are multiple maximal elements, move the first one first. Prove that it preserves the multiset of elements and that it moves the maximum to the end:

definition *place_max_correct* :: "$('a::linorder)\ list \Rightarrow 'a\ list$"

lemma *mset_place_max_correct*[*simp*]: "$mset\ (place\_max\_correct\ (x\#xs)) = mset\ (x\#xs)$"

lemma *last_place_max_correct*[]: "$xs \neq [] \Longrightarrow last\ (place\_max\_correct\ xs) = Max\ (set\ xs)$"

Using *place_max_correct* define a simple algorithm that sorts a list by prefix reversal. The algorithm should work similar to selection sort (*sel_sort*) from the tutorial. First, move the maximum to the end using *place_max_correct*, then sort the remaining list.

Note: Your algorithm must actually follow this scheme, in particular do not use/implement some different sorting algorithm.

Hint: You will probably need the following lemma for termination of *psort*

Hint: The functions *last*/*butlast* might be useful.

lemma *length_place_max_correct[simp]*: *"length (place_max_correct (x#xs)) = length (x#xs)"*

fun *psort* :: *"('a::linorder) list ⇒ 'a list"*

Show that your algorithm is a sorting algorithm, that is show it preserves the multiset of elements and produces a sorted list:

lemma *psort_mset[simp]*: *"mset (psort xs) = mset xs"*
lemma *sorted_psort*: *"sorted (psort xs)"*

Homework 7.4 Pancake sorting 2

Submission until Thursday, June 15, 23:59pm.

(This is a bonus exercise, worth 5 bonus points, when computing your homework performance as a percentage, bonus points will only count on your side, but not towards the total score, it will not be checked by the submission system. If you want to have it corrected, please put a **(*** bonus ***)** into your file)

We want to show that it is possible to sort a list using a linear (in the length of the list) number of reversals.

We could try to do this by defining a cost function for *psort*, counting the number of reversals performed, and give a bound for it. However, here we try a different approach, directly computing a certificate that tells us exactly which reversals to perform.

First, we define a function *psortable_in xs k*, which specifies what it means for a list *xs* to be pancake-sortable in *k* reversals:

fun *rev_pre_chain* :: *"nat list ⇒ 'a list ⇒ 'a list"* **where**
"rev_pre_chain [] xs = xs"
| *"rev_pre_chain (r#rs) xs = rev_pre_chain rs (rev_pre r xs)"*

definition *"psortable_in xs k ≡ ∃ rs . length rs ≤ k ∧ (let ys = rev_pre_chain rs xs in mset ys = mset xs ∧ sorted ys)"*

We now want to give an algorithm that computes such a *rs*. For this, give a modified version of *psort*, which, instead of directly computing the sorted list, computes a list of reversals one can to perform to sort the list.

fun *psort_revs* :: *"('a :: linorder) list ⇒ nat list"*

Give and prove a linear (in the length of the input list) bound for the length of the list of reversals computed by *psort_revs*

lemma *length_psort_revs*: *“length (psort_revs xs) ≤ undefined”*

Prove that applying the computed list of reversals sorts the list:

lemma *mset_rev_pre_chain_psort_revs*: *“mset (rev_pre_chain (psort_revs xs) xs) = mset xs”*

lemma *sorted_psort_revs*: *“sorted (rev_pre_chain (psort_revs xs) xs)”*

Finally, conclude that you can sort any list in a linear number of reversals (in the length of the input list):

theorem *psortable_in_linear*: *“psortable_in xs undefined”*