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

SS 2022
3. 6. 2022

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 Bubble Sort

Submission until Thursday, June 9, 23:59pm.

Implement a bubble-sort, i.e. a sorting algorithm, where elements are bubbled up for multiple iterations until the list is sorted. Start by defining the function *bubble*, which should traverse the list once and swap adjacent elements if their order is wrong:

fun *bubble* :: *"'a ::linorder list ⇒ 'a list"*

The sorting algorithm then executes the *bubble* function *length* numer of times:

fun *bsort_aux* :: *"nat ⇒ 'a::linorder list ⇒ 'a list"* **where**
  "bsort_aux 0 xs = xs" |
  "bsort_aux (Suc n) xs = bsort_aux n (bubble xs)"

definition *bsort* :: *"'a::linorder list ⇒ 'a list"* **where**
  "bsort xs = bsort_aux (length xs) xs"

Warmup: Show that the *mset* stays the same. You'll need similar lemmas about *bsort_aux* and *bsort*.

theorem *bsort_mset*: *"mset (bsort xs) = mset xs"*

The easy part: Define the canonical timing function for *bsort* and the involved functions. Assume that *length* has a constant cost and [] a cost of zero.

fun *T_bubble* :: *"'a::linorder list ⇒ nat"*
definition *T_bsort* :: *"'a::linorder list ⇒ nat"*

Show that the run-time is quadratic:

theorem *T_bsort*: *“$\exists$ c d. T_bsort xs $\leq$ c $*$ (length xs)^2 + d”*

The difficult part: Show that the result is sorted. For that, define a measure function that decreases in every *bubble* invocation (if the list is unsorted - otherwise it should at least not increase), and reaches zero for a sorted list. Prove those properties!

Hint: The structure of the measure function should be similar to the *sorted_wrt* function.

fun *measure* :: *“'a::linorder list $\Rightarrow$ nat”*
lemma *measure_sorted_0*: *“sorted xs $\longleftrightarrow$ measure xs = 0”*
lemma *measure_le*[*simp*]: *“measure (bubble xs) $\leq$ measure xs”*
lemma *measure_dec*[*simp*]: *“$\neg$sorted xs $\Longrightarrow$ measure (bubble xs) < measure xs”*

With those properties, show that the list is sorted. The *less_induct* principle may be useful:

$(\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$

lemma *bsort_sorted*: *“sorted (bsort xs)”*