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

SS 23
2. 6. 2023

# 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* $x$ []!)

Hint: Show an equality rather than an inequality.

```
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** Quickselect

*Submission until Monday, 5 June, 23:59pm.*

From https://en.wikipedia.org/wiki/Quickselect:

Quickselect is a selection algorithm to find the kth smallest element in an unordered list, also known as the kth order statistic. Like the related quicksort sorting algorithm, it was developed by Tony Hoare, and thus is also known as Hoare's selection algorithm. Like quicksort, it is efficient in practice and has good average-case performance, but has poor worst-case performance. Quickselect and its variants are the selection algorithms most often used in efficient real-world implementations.

Quickselect uses the same overall approach as quicksort, choosing one element as a pivot and partitioning the data in two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing into both sides, as in quicksort, quickselect only recurses into one side – the side with the element it is searching for.

Your task is to prove correct the quickselect algorithm, which is already implemented for you.

Note: for a list *xs* and a predicate *P*, [*x*←*xs. P x*] is the same as *filter P xs.*

Your first task is to prove the crucial idea of quicksort, i.e., that partitioning wrt. a pivot element *p* is correct.

**lemma** *partition_correct*: *"insort xs = insort [x←xs. x<p] @ insort [x←xs. ¬(x<p)]"*

Hint: Induction, and auxiliary lemmas to transform a term of the form *insort1 x (xs*

@ *ys*) when you know that *x* is greater than all elements in *xs* / less than or equal all elements in *ys*. Also, not all lemmas about *insort* are already proven!

Next, show that quickselect is correct. Proceed by computation induction, and a case distinction according to the cases in the body of the quickselect function:

**theorem** *quickselect_correct*: *"k<length xs $\Longrightarrow$ quickselect xs k = insort xs ! k"*
**proof** (*induction xs k rule*: *quickselect.induct*)
  **case** *2* **then show** *?case* **by** *simp*
**next**
  **case** (*1 x xs k*)

Note: To make the induction hypothesis more readable, you can collapse the first two premises of the form *?x=...* by reflexivity:

  **note** *IH = "1.IH"*[*OF refl refl*]

Insert your proof here!

## **Homework 6.2** Quickselect running time complexity

*Submission until Monday, 5 June, 23:59pm.* This is a bonus homework, worth 5 bonus points.
Prove that quickselect does a number of comparisons that is at most quadratic in the length of the given list. An exact cost function for quickselect is already given.

Show that the number of required comparisons is at most (*length xs* + *1*)$^2$. Hints:

- Follow a similar proof structure to the one above.
- Have a look at the lemma *sum_length_filter_compl*.

**theorem** *quickselect_quadratic*: *"C_quickselect xs k $\leq$ (length xs + 1)$^2$"*