Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Mohammad Abdulaziz, Ph.D.

SS 2020
29. 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** Stability of Insertion Sort

Have a look at Isabelle's standard implementation of sorting: *sort_key*. (Use Ctrl-Click to jump to the definition in `~~/src/HOL/List.thy`) Show that this function is a stable sorting algorithm, i.e., the order of elements with the same key is not changed during sorting!

**lemma** *"filter* $(\lambda x.\ k\ x = a)$ $($*sort_key* $k\ xs)$ = *filter* $(\lambda x.\ k\ x = a)\ xs$*"*

Hint: You do not necessarily need Isar, and the auxiliary lemmas you need are already in Isabelle's library. The Query window (or *find_theorems*) is your friend!

## **Homework 6.1** Quickselect

*Submission until Friday, 5 June, 10:00am.*

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

Quickselect is a selection algorithm to find the kth smallest element in an unordered list. It is related to the quicksort sorting algorithm. Like quicksort, 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 can be implemented in Isabelle as follows:

```
fun quickselect :: "'a::linorder list ⇒ nat ⇒ 'a" where
  "quickselect (x#xs) k = (let
    xs1 = [y←xs. y<x];
    xs2 = [y←xs. ¬(y<x)]
  in
    if k<length xs1 then quickselect xs1 k
    else if k=length xs1 then x
    else quickselect xs2 (k−length xs1−1)
  )"
| "quickselect [] _ = undefined"
```

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: "sort xs = sort [x←xs. x<p] @ sort [x←xs. ¬(x<p)]"
```

Hint: Induction, and auxiliary lemmas to transform a term of the form *insort x* (*xs* @ *ys*) when you know that *x* is greater than all elements in *xs* / less than or equal all elements in *ys*.

Next, show that quickselect is correct

```
lemma "k<length xs ⟹ quickselect xs k = sort xs ! k"
```

Proceed by computation induction, and a case distinction according to the cases in the body of the quickselect function

```
proof (induction xs k rule: quickselect.induct)
  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!

```
next
  case 2 then show ?case by simp
qed
```

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

*Submission until Friday, 5 June, 10:00am.* 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. The following is a cost function for the comparsions of quickselect:

```
fun c_quickselect :: "'a::linorder list ⇒ nat ⇒ nat" where
  "c_quickselect (x#xs) k = (let
    xs1 = [y←xs. y<x];
    xs2 = [y←xs. ¬(y<x)]
  in
    length xs +
      (if k<length xs1 then c_quickselect xs1 k + 1
       else if k=length xs1 then 2
       else c_quickselect xs2 (k−length xs1−1) + 3)
  )"
| "c_quickselect [] _ = 0"
```

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

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

**lemma** "$\textit{c\_quickselect xs k} \leq (\textit{length xs} + 1)^2$"