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

SS 2021
9. 7. 2021

Functional Data Structures

Exercise Sheet 13

Presentation of Mini-Projects:

You are invited, on a voluntary basis, to give a short presentation of your mini-projects in the tutorial on July 16.

Depending on how many presentations we have, the time slots will be 5 to 10 minutes, plus 2 minutes for questions.

If you are interested, please write me a short email until Wednesday.

The following are old exam questions!

Exercise 13.1 Amortized Complexity

A "stack with multipop" is a list with the following two interface functions:

fun *push* :: *"'a ⇒ 'a list ⇒ 'a list"* **where**
"push x xs = x # xs"

fun *pop* :: *"nat ⇒ 'a list ⇒ 'a list"* **where**
"pop n xs = drop n xs"

You may assume

definition *T_push* :: *"'a ⇒ 'a list ⇒ nat"* **where**
"T_push x xs = 1"

definition *T_pop* :: *"nat ⇒ 'a list ⇒ nat"* **where**
"T_pop n xs = min n (length xs)"

Use the potential method to show that the amortized complexity of *push* and *pop* is constant.

If you need any properties of the auxiliary functions *length*, *drop* and *min*, you should state them but you do not need to prove them.

Exercise 13.2 Converting List for Balanced Insert

Recall the standard insertion function for unbalanced binary search trees.

```
fun insert :: "'a::linorder ⇒ 'a tree ⇒ 'a tree" where
"insert x Leaf = Node Leaf x Leaf" |
"insert x (Node l a r) =
  (case cmp x a of
     LT ⇒ Node (insert x l) a r |
     EQ ⇒ Node l a r |
     GT ⇒ Node l a (insert x r))"
```

We define the function *from_list*, which inserts the elements of a list into an initially empty search tree:

```
definition from_list :: "'a::linorder list ⇒ 'a tree" where
  "from_list l = fold insert l Leaf"
```

Your task is to specify a function *preprocess*::*'a*, that preprocesses the list such that the resulting tree is almost complete.

You may assume that the list is sorted, distinct, and has exactly *2^k − 1* elements for some *k*. That is, your *preprocess* function must satisfy:

```
fun preprocess :: "'a list ⇒ 'a list"

lemma
  assumes "sorted l"
      and "distinct l"
      and "length l = 2^k−1"
  shows "set (preprocess l) = set l" and "acomplete (from_list (preprocess l))"
```

Note: No proofs required, only a specification of the *preprocess* function!

Exercise 13.3 Trees with Same Structure

Question 1

Specify the recursion equations of a function *same* that returns true if and only if the two trees have the same structure (i.e., ignoring values).

```
fun same :: "'a tree ⇒ 'a tree ⇒ bool"
```

Question 2

Show, by computation induction wrt. *same*, that insertion of arbitrary elements into two Braun heaps with the same structure yields heaps with the same structure again.

For your proof, it is enough to cover the (Node,Node) case. If you get analogous subcases, only elaborate one of them!

Hint: Here is the definition of *Tree_Set.insert*:

```
fun insert :: "'a::linorder ⇒ 'a tree ⇒ 'a tree" where
```

"insert a Leaf = Node Leaf a Leaf" |
"insert a (Node l x r) =
(if a < x then Node (insert x r) a l else Node (insert a r) x l)"

lemma *same_insert*: *"same t t′ ⟹ same (insert x t) (insert y t′)"*

Homework 13.1 A counter with increment and decrement operations

Submission until Thursday, July 14, 23:59pm.

A k-bit counter can be formalised as a list of booleans. An increment operation for such a counter is defined as follows:

fun *incr* :: *"bool list ⇒ bool list"* **where**
"incr [] = []" |
"incr (False#bs) = True # bs" |
"incr (True#bs) = False # incr bs"

The running time of this increment operation can be defined as follows:

fun *T_incr* :: *"bool list ⇒ nat"* **where**
"T_incr [] = 0" |
"T_incr (False#bs) = 1" |
"T_incr (True#bs) = T_incr bs + 1"

For such a k-bit counter with only an increment operation, an amortised analysis of the running time of a sequence of n increment operations reveals it is $O(n)$. However, if the counter has a decrement operation, then for a sequence of n operations, a lower bound for the running time must be at least linear in the product nk. This holds regardless of the time required to perform the decrement operation. In fact this holds for any operation *decr* satisfying the following two assumption:

decr ((replicate (k−1) False) @ [True]) = (replicate (k−(Suc 0)) True) @ [False]

length (decr bs) = length bs

Above, *replicate n x* is the list [x, ..., x] of length n. The following locale specifies a counter with such an operation.

locale *counter_with_decr* =
  fixes *decr*::*"bool list ⇒ bool list"* **and** *k*::*"nat"*
  assumes
    decr[simp]: *"decr ((replicate (k−(Suc 0)) False) @ [True]) =*
                  (replicate (k−(Suc 0)) True) @ [False]" **and**
    decr_len_eq[simp]: *"length (decr bs) = length bs"* **and**
    k[simp]: *"1 ≤ k"*
begin

In this homework you are required to show that indeed the running time of a sequence of operations of length n is $\Theta(nk)$. You can assume that the running time of the decrement operation is 1.

fun *T_decr*::*"bool list ⇒ nat"* **where**

```
  “T_decr _ = 1”
```

To prove the required running time, you will need to prove an upper and a lower bound on the running time that are linear in *nk*. To prove either bound, you will need to reason about lists whose elements are of the type *op*. Such lists correspond to lists of operations on the counter.

```
datatype op = Decr | Incr
```

The running time of a list of operations is given by the function *T_exec*, is defined as follows:

```
fun exec1::“op ⇒ (bool list ⇒ bool list)” where
  “exec1 Incr = incr” |
  “exec1 Decr = decr”

fun T_exec1::“op ⇒ (bool list ⇒ nat)” where
  “T_exec1 Incr = T_incr” |
  “T_exec1 Decr = T_decr”

fun T_exec :: “op list ⇒ bool list ⇒ nat” where
  “T_exec [] bs = 0” |
  “T_exec (op # ops) bs = (T_exec1 op bs + T_exec ops (exec1 op bs))”
```

Prove the following upper bound on the running time of sequences of operations:

```
theorem inc_dec_seq_ubound: “length bs = k ⟹ T_exec ops bs ≤ (length ops) ∗ length bs”
```

To prove the lower bound, you will need to define a function *oplist* that, given a natural number *n*, constructs a list of operations whose running time is at least linear in *nk* for at least one counter initial configuration, *bs0*.

```
fun oplist :: “nat ⇒ op list”


definition bs0
```

You are required to prove the following lower bound. The two-element list induction scheme and *nat* case distinction might be helpful.

```
lemma induct_list012[case_names empty single multi]:
  “P [] ⟹ (⋀x. P [x]) ⟹ (⋀x y xs. P xs ⟹ P (x#y#xs)) ⟹ P xs”
  by (rule List.induct_list012)

lemma case_nat012[case_names zero one two]:
  “⟦n = 0 ⟹ P; n = 1 ⟹ P; ⋀n'. n = Suc (Suc n') ⟹ P⟧ ⟹ P”
  by (metis One_nat_def nat.exhaust)

theorem inc_dec_seq_lbound: “n ∗ k ≤ 2 ∗ (T_exec (oplist n) bs0)”
```