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

SS 24
12. 7. 2024

Functional Data Structures

Exercise Sheet 12

Exercise 12.1 Amortized Complexity

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

fun *push* :: "$'a \Rightarrow 'a\ list \Rightarrow 'a\ list$" **where**
"$push\ x\ xs = x \# xs$"

fun *pop* :: "$nat \Rightarrow 'a\ list \Rightarrow 'a\ list$" **where**
"$pop\ n\ xs = drop\ n\ xs$"

You may assume

definition *T_push* :: "$'a \Rightarrow 'a\ list \Rightarrow nat$" **where**
"$T\_push\ x\ xs = 1$"

definition *T_pop* :: "$nat \Rightarrow 'a\ list \Rightarrow nat$" **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 12.2 Sparse Binary Numbers

Implement operations carry, inc, and add on sparse binary numbers, analogously to the operations link, ins, and merge on binomial heaps.

Show that the operations have logarithmic worst-case complexity.

type_synonym *rank* = *nat*
type_synonym *snat* = "*rank list*"

abbreviation *invar* :: "$snat \Rightarrow bool$" **where** "$invar\ s \equiv sorted\_wrt\ (<)\ s$"
definition α :: "$snat \Rightarrow nat$" **where** "$\alpha\ s = sum\_list\ (map\ ((\hat{}\ )\ 2)\ s)$"

lemmas [*simp*] = *sorted_wrt_append*

fun *carry* :: "*rank* ⇒ *snat* ⇒ *snat*"

lemma *carry_invar*[*simp*]:
  assumes "*invar rs*"
 shows "*invar* (*carry r rs*)"

lemma *carry_α*:
  assumes "*invar rs*"
   and "$\forall r' \in set\ rs.\ r \leq r'$"
 shows "α (*carry r rs*) = $2\hat{}r + \alpha\ rs$"

definition *inc* :: "*snat* ⇒ *snat*"

lemma *inc_invar*[*simp*]: "*invar rs* ⟹ *invar* (*inc rs*)"

lemma *inc_α*[*simp*]: "*invar rs* ⟹ α (*inc rs*) = *Suc* (α *rs*)"

fun *add* :: "*snat* ⇒ *snat* ⇒ *snat*"

lemma *add_invar*[*simp*]:
  assumes "*invar* rs_1"
   and "*invar* rs_2"
 shows "*invar* (*add* rs_1 rs_2)"

lemma *add_α*[*simp*]:
  assumes "*invar* rs_1"
   and "*invar* rs_2"
 shows "$\alpha\ (add\ rs_1\ rs_2) = \alpha\ rs_1 + \alpha\ rs_2$"

thm *sorted_wrt_less_sum_mono_lowerbound*

lemma *size_snat*:
  assumes "*invar rs*"
 shows "$2\hat{}length\ rs \leq \alpha\ rs + 1$"

fun *T_carry* :: "*rank* ⇒ *snat* ⇒ *nat*"

definition *T_inc* :: "*snat* ⇒ *nat*"

lemma *T_inc_bound*:
  assumes "*invar rs*"
 shows "*T_inc rs* ≤ *log 2* (α *rs* + *1*) + *2*"

fun *T_add* :: "*snat* ⇒ *snat* ⇒ *nat*"

lemma *T_add_bound*:
 fixes rs_1 rs_2
 defines "$n_1 \equiv \alpha\ rs_1$"
 defines "$n_2 \equiv \alpha\ rs_2$"

assumes *INVARS*: "*invar* rs_1" "*invar* rs_2"
shows "$T\_add\ rs_1\ rs_2 \le 4*log\ 2\ (n_1 + n_2 + 1) + 2$"

Homework 12.1 A counter with increment and decrement operations

Submission until Thursday, July 18, 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*

In the following, 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)*

You are required to prove the following lower bound:

theorem *inc_dec_seq_lbound: "length (oplist n) * k ≤ 2 * (T_exec (oplist n) bs0)"*