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

SS 23
11. 7. 2023

Functional Data Structures

Exercise Sheet 12

Exercise 12.1 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 ⇒ bool" where "invar s ≡ sorted_wrt (<) s"
definition α :: "snat ⇒ nat" where "α s = sum_list (map ((^) 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 "∀r'∈set rs. r≤r'"
  shows "α (carry r rs) = 2^r + α 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 rs₁"
     and "invar rs₂"
  shows "invar (add rs₁ rs₂)"
```

```
lemma add_α[simp]:
    assumes "invar rs₁"
      and "invar rs₂"
  shows "α (add rs₁ rs₂) = α rs₁ + α rs₂"

thm sorted_wrt_less_sum_mono_lowerbound

lemma size_snat:
    assumes "invar rs"
  shows "2^length rs ≤ α 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 rs₁ rs₂
  defines "n₁ ≡ α rs₁"
  defines "n₂ ≡ α rs₂"
  assumes INVARS: "invar rs₁" "invar rs₂"
  shows "T_add rs₁ rs₂ ≤ 4*log 2 (n₁ + n₂ + 1) + 2"
```

Homework 12.1 Be Original!

Submission until Monday, July 17, 23:59pm.

Develop a nice Isabelle formalisation yourself!

- You may develop a formalisation from all areas, not only functional data structures. Creative topics are encouraged!
- Document your solution well, such that it is clear what you have formalised and what your main theorems state!
- Set yourself a time frame and some intermediate/minimal goals. Your formalisation needs not be universal and complete.
- You are encouraged to discuss the realisability of your project with us!
- In total, the homework will yield 15 points (for minimal solutions). Additionally, bonus points may be awarded for particularly nice/original/etc solutions.
- This week, polish up your project for the final submission!
- To submit, use the submission system if you have a single file. **Submitting is sufficient, ignore any errors that the submission system may raise when**

checking the submission. If the project is more than one file, send an archive by e-mail.

Homework 12.2 Modified Binomial Heaps (7 points)

Submission until Monday, July 17, 23:59pm.

In its simplist form, a binomial heap can be implemented using binomial trees that store the the rank of every tree in its root. One optimisation is to eliminate the redundancy of storing ranks in the root of every tree, and instead store the rank only at the top level by pairing every tree with its rank in the heap. The following types describe a binomial heap with this optimisation:

datatype *'a tree = Node 'a ('a tree list)*

type_synonym *'a heap = "(nat*'a tree) list"*

For such a heap to be a binomial heap it has to conform to the invariant *invar* defined as follows:

$btree\ r\ (Node\ x\ ts) = (length\ ts = r \wedge (\forall (x,\ y) \in set\ (zip\ (rev\ [0..{<}r])\ ts).\ btree\ x\ y))$

$bheap\ ts = ((\forall (r,\ t) \in set\ ts.\ btree\ r\ t) \wedge sorted\_wrt\ (<)\ (map\ fst\ ts))$

$heap\ (Node\ x\ ts) = (\forall t \in set\ ts.\ heap\ t \wedge x \leq tree.root\ t)$

$heaps\ ts = (\forall t \in set\ ts.\ heap\ t)$

$invar\ ts = (bheap\ ts \wedge heaps\ (map\ snd\ ts))$

In this homework you are required to define an insertion and a merging functions for this heap and show that they preserve the elements of their inputs as well as produce heaps that conform to the invariant *invar*.

definition *insert* :: *"'a::linorder ⇒ 'a heap ⇒ 'a heap"*

lemma *invar_insert[simp]*: *"invar t ⟹ invar (insert x t)"*
lemma *mset_heap_insert[simp]*: *"mset_heap (insert x t) = {#x#} + mset_heap t"*
fun *merge* :: *"'a::linorder heap ⇒ 'a heap ⇒ 'a heap"*

lemma *invar_merge[simp]*: "⟦ $invar\ ts_1$; $invar\ ts_2$ ⟧ ⟹ $invar\ (merge\ ts_1\ ts_2)$"

lemma *mset_heap_merge[simp]*: "$mset\_heap\ (merge\ ts_1\ ts_2) = mset\_heap\ ts_1 + mset\_heap\ ts_2$"

Start from "src/HOL/Data_Structures/Binomial_Heap.thy" that has an implementation of binomial heaps without this optimisation.