Final Exam

Functional Data Structures

28. 7. 2020

Proof Guidelines: We expect valid Isabelle proof scripts to be submitted as a solution to the questions of this exam. Major proof steps, especially inductions, need to be stated explicitly. The use of "sorry" may lead to the deduction of points but is preferable to spending a lot of time on individual proof steps.

Please submit your solution via the submission system and ALSO via email to mansour@in.tum.de. You have to include your final answer to all questions in one email. If you do not submit the exam by the deadline (10 minutes after the official end of the exam) either via the submission system or via email you will have failed the exam (X = no show = 5,0). The solutions you send by email are primarily intended as a backup in case of technical problems. Unless you state explicitly that we should grade the email submission (and it was submitted in time), we will grade the submission on the submission system.

In the unlikely event that you discover a mistake in one of the questions you should communicate this to nipkow@in.tum.de and mansour@in.tum.de. We will communicate any corrections to the exam questions by email to you. Thus you do need to watch your email at regular intervals.

Also, if you encounter unforeseen technical problems during the exam, you can send an e-mail to nipkow@in.tum.de and mansour@in.tum.de.

1 Induction

1.1 Question 1

Consider the following definitions:

datatype *t3* = *Lf* | *Nd t3 t3 t3*

fun *sz* :: *"t3 ⇒ nat"* **where**
"sz Lf = 0" |
"sz (Nd r s t) = sz r + sz s + sz t + 1"

fun *lvs* :: *"t3 ⇒ nat"* **where**
"lvs Lf = 1" |
"lvs (Nd r s t) = lvs r + lvs s + lvs t"

There is a linear relationship between the size (*sz*) and the number of leaves (*lvs*) of a tree. Find this relationship and prove *lvs t = a ∗ sz t + b* for the correct *a* and *b*.

You have to use a structured Isar proof. The only two proof methods you are allowed to use are *induction* and *simp*. Every call of *simp* must use exactly one named fact or fact collection. Permitted are the definitions *sz.simps* and *lvs.simps*, the induction hypotheses, or *algebra_simps*, and this is an exclusive or. For instance, *simp only*: *lvs.simps* is a valid proof method, but *simp add*: *lvs.simps* is not for this question.

abbreviation *a* **where**

abbreviation *b* **where**

lemma *"lvs t = a ∗ sz t + b"*

2 Trees with Same Structure

2.1 Question 1

The following is the data type that describes a binary tree:

datatype *'a tree = Leaf | Node "'a tree" 'a "'a tree"*

Define a recursive function

fun *same :: "'a tree ⇒ 'b tree ⇒ bool"* **where**

that returns true if and only if the two binary trees have the same structure (i.e. ignoring values).

2.2 Question 2

The following is the definition of a function that inserts an element into a Braun heap that is based on the binary trees described above:

fun *insert_pq :: "'a::linorder ⇒ 'a tree ⇒ 'a tree"* **where**
"insert_pq a Leaf = Node Leaf a Leaf" |
"insert_pq a (Node l x r) =
(if a < x then Node (insert_pq x r) a l else Node (insert_pq a r) x l)"

Show that insertion of arbitrary elements into two Braun heaps with the same structure yields heaps with the same structure again.

lemma *same_insert: "same t t' ⟹ same (insert_pq x t) (insert_pq y t')"*

3 2-3-4 Trees

2-3-4 trees are trees where nodes can either have 2, 3, or 4 children. The following data type encodes 2-3-4 trees.

```
datatype 'a tree234 =
  Leaf |
  Node2 "'a tree234" 'a "'a tree234" |
  Node3 "'a tree234" 'a "'a tree234" 'a "'a tree234" |
  Node4 "'a tree234" 'a "'a tree234" 'a "'a tree234" 'a "'a tree234"
```

To realise logarithmic access and update running times, the tree has to maintain a certain completeness invariant, as well as an ordering invariant on the data stored in the tree. Both invariants are defined as follows:

```
fun complete :: "'a tree234 ⇒ bool" where
"complete Leaf = True" |
"complete (Node2 l _ r) = (complete l & complete r & height l = height r)" |
"complete (Node3 l _ m _ r) =
  (complete l & complete m & complete r & height l = height m & height m = height r)" |
"complete (Node4 l _ m1 _ m2 _ r) =
  (complete l & complete m1 & complete m2 & complete r & height l = height m1 & height m1
= height m2 & height m2 = height r)"

fun ordered :: "'a::linorder tree234 ⇒ bool" where
"ordered Leaf ⟷ True" |
"ordered (Node2 t1 v t2) ⟷
 ordered t1 ∧ ordered t2 ∧ (∀x∈set_tree234 t1. (<) x v) ∧ (∀x∈set_tree234 t2. (<) v x)" |
"ordered (Node3 t1 v1 t2 v2 t3) ⟷
 ordered t1 ∧ ordered t2 ∧ ordered t3 ∧ v1 < v2
 ∧ (∀x∈set_tree234 t1. (<) x v1) ∧ (∀x∈set_tree234 t2. (<) v1 x)
 ∧ (∀x∈set_tree234 t2. (<) x v2) ∧ (∀x∈set_tree234 t3. (<) v2 x)" |
"ordered (Node4 t1 v1 t2 v2 t3 v3 t4) ⟷
 ordered t1 ∧ ordered t2 ∧ ordered t3 ∧ ordered t4 ∧ v1 < v2 ∧ v2 < v3
 ∧ (∀x∈set_tree234 t1. (<) x v1) ∧ (∀x∈set_tree234 t2. (<) v1 x)
 ∧ (∀x∈set_tree234 t2. (<) x v2) ∧ (∀x∈set_tree234 t3. (<) v2 x)
 ∧ (∀x∈set_tree234 t3. (<) x v3) ∧ (∀x∈set_tree234 t4. (<) v3 x)"
```

In the functions above, the function *height* returns the height of a 2-3-4 tree.

In this section you are required to define functions that perform set operations using 2-3-4 trees. Your functions have to traverse the tree at most once but may use function *height* as much as they please. You can use an operation you define in the answer to one part as an auxiliary function to define an operation required in another part.

Note: *you are not required to prove properties about those functions.*

3.1 Part 1

Define a function *join* that, given two ordered 2-3-4 trees and a root, computes an ordered 2-3-4 tree containing the given root and the elements of the two given trees.

fun *join*::"'*a tree234* ⇒ '*a* ⇒ '*a tree234* ⇒ '*a tree234*" **where**

The function join has to preserve the members in the given trees as well as their relative ordering, and the output tree has to conform to the completeness invariant. Formally, *join* has to conform to the following properties:

lemma *join_complete*: "*complete l* ⟹ *complete r* ⟹ *complete* (*join l x r*)"
lemma *set_join*: "*set_tree234* (*join l x r*) = *set_tree234 l* ∪ {*x*} ∪ *set_tree234 r*"
lemma *ordered_join*: "⟦*ordered l*; *ordered r*; ∀ *x*∈*set_tree234 l*. *x* < *a*; ∀ *x*∈*set_tree234 r*. *a* < *x*⟧
⟹ *ordered* (*join l a r*)"

3.2 Part 2

Define a function that inserts an element in a given 2-3-4 tree. The function has to insert the element such that the resulting tree is ordered, complete, and contains all the elements in the given tree. You have to use the function *join* that you defined in the last exercise.

definition *insert* :: "'*a*::*linorder* ⇒ '*a tree234* ⇒ '*a tree234*" **where**

Formally, the function has to conform to the following properties:

lemma *set_tree_insert*: "*ordered t* ⟹ *set_tree234* (*insert x t*) = *Set.insert x* (*set_tree234 t*)"
lemma *ordered_insert*: "*ordered t* ⟹ *ordered* (*insert x t*)"
lemma *inv_insert*: "*complete t* ⟹ *complete* (*insert x t*)"

4 Amortized Complexity

Consider a new kind of stack that consists of two stacks that are swapped with each push and pop. Moreover, the pop is a multi-pop, where any number of elements can be popped:

type_synonym *'a stk = "'a list * 'a list"*

fun *push* :: *"'a ⇒ 'a stk ⇒ 'a stk"* **where**
"push x (xs,ys) = (ys,x#xs)"

fun *pop* :: *"nat ⇒ 'a stk ⇒ 'a stk"* **where**
"pop n (xs,ys) = (ys, drop n xs)"

Assume

fun *t_push* :: *"'a ⇒ 'a stk ⇒ nat"* **where**
"t_push x s = 1"

fun *t_pop* :: *"nat ⇒ 'a stk ⇒ nat"* **where**
"t_pop n (xs,ys) = min n (length xs)"

Use the potential method to show that the amortized complexity of both *push* and *pop* is constant. Find the exact constants, not just upper bounds. You are required to state all the definitions and theorem statements yourself. Please use comments and/or meaningful names to clarify what your definitions and theorem statements stand for.