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

SS 24
31. 5. 2024

Functional Data Structures

Exercise Sheet 6

Exercise 6.1 Estimate for Number of Leafs

Note: Use Isar, proofs using *metis*, *smt*, *meson*, *or moura* (as generated by sledgehammer) are forbidden!

Define a function to count the number of leafs in a binary tree:

fun *num_leafs* :: *"'a tree ⇒ nat"*

Show that we can estimate the number of leafs in a tree as follows:

theorem *num_leafs_est*: *"num_leafs t ≤ 2^height t"*

Exercise 6.2 Paths in Graphs

A graph is described by its adjacency matrix, i.e., G :: *'a ⇒ 'a ⇒ bool*.

Define a predicate *path G u p v* that is true if p is a path from u to v, i.e., p is a list of nodes, not including u, such that the nodes on the path are connected with edges. In other words, *path G u* $(p_1 \dots p_n)$ v, iff $G\ u\ p_1$, $G\ p_i\ p_{i+1}$, and $p_n = v$. For the empty path (n=*0*), we have u=v.

fun *path* :: *"('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a list ⇒ 'a ⇒ bool"*

Test cases

definition *"nat_graph x y ⟷ y=Suc x"*
value *‹path nat_graph 2 [] 2›*
value *‹path nat_graph 2 [3,4,5] 5›*
value *‹¬ path nat_graph 3 [3,4,5] 6›*
value *‹¬ path nat_graph 2 [3,4,5] 6›*

Show the following lemma, that decomposes paths. Register it as simp-lemma.

lemma *path_append[simp]*: *"path G u (p1@p2) v ⟷ (∃ w. path G u p1 w ∧ path G w p2 v)"*

Show that, for a non-distinct path from u to v, we find a longer non-distinct path from u to v. Note: This can be seen as a simple pumping-lemma, allowing to pump the length of the path.

Hint: Theorem *not_distinct_decomp*.

lemma *pump_nondistinct_path*:
  assumes *P*: "*path G u p v*"
  and *ND*: "$\neg$*distinct p*"
shows "$\exists\, p'.$ *length* $p' >$ *length p* $\wedge \neg$*distinct* $p' \wedge$ *path G u* p' *v*"

Exercise 6.3 Level-order Traversal

(adapted from a previous exam question, only if time left)

Write a function *levels* that lists all elements of a binary tree level by level, from left to right, e.g.: *levels* $\langle\langle\langle\rangle, 2, \langle\rangle\rangle, 1, \langle\langle\langle\rangle, 4, \langle\rangle\rangle, 3, \langle\rangle\rangle\rangle = [[1], [2, 3], [4]]$. You may define auxiliary functions, but your function should only traverse the tree once.

fun *levels* :: "'*a tree* $\Rightarrow$ '*a list list*"

Show that the number of levels is exactly the height of the tree:

lemma *levels_height*: "*length*(*levels t*) = *height t*"

The set function for a list of levels is defined by first creating a set of sets and then taking the union over those (denoted $\bigcup$):

definition *set2* :: "'*a list list* $\Rightarrow$ '*a set*" **where**
"*set2 xss* $\equiv \bigcup$(*set* (*map set xss*))"

Show that *levels* returns the correct elements.

Hint: In your induction step, you will likely need a chain of equations.

lemma *levels_set*: "*set2* (*levels t*) = *set_tree t*"

Homework 6.1 Simple Paths

Submission until Thursday, June 6, 23:59pm.

A simple path is a path without loops, or, in other words, a path where no node occurs twice. (Note that the first node of the path is not included, such that there may be a simple path from *u* to *u*.)

Show that for every path, there is a corresponding simple path:

lemma *exists_simple_path*:
  assumes "*path G u p v*"
shows "$\exists\, p'.$ *path G u* p' *v* $\wedge$ *distinct* p'"

Your proof should be by induction on the length of *p*. Use the induction principle *length_induct* for this.

Homework 6.2 Sorting Networks

Submission until Thursday, June 06, 23:59pm.

Comparison networks are a model of parallel algorithms on fixed-size lists. A sorting network is a specific comparison network that sorts its input lists.

A comparison network can be viewed as set of *wires* x_i, one for each list element. Between those wires are a number of *comparators* c_i; each comparator is connected to two wires. For Example (lists of size three):

```
x0---[]--------[]----
      | c0       | c2
x1---[]---[]---[]----
           | c1
x2--------[]---------
```

Each comparator will shift the greater element of its inputs up, and the smaller element down.

We represent a network by a list of comparators, where each comparator is characterized by the index of its wires – i.e., c_0=(*0*,*1*), and after the applying c_0, the greater element will be at position of x_1.

That is, a comparator (*i*,*j*) should place the smaller/larger of its two inputs at wire *i*/*j* respectively.

type_synonym *comparator* = *"(nat × nat)"*
type_synonym *compnet* = *"comparator list"*

Write a function to perform the computation of a single comparator on a *'a list*. If the comparator would compare elements out of the range of the input list, return the input unchanged.

Hint: Use the existing *list_update* and *nth* functions. *list_update* also has nice snytax: *xs*[*0* := *1*, *1* := *2*]

definition *compnet_step* :: *"comparator ⇒ 'a :: linorder list ⇒ 'a list"*

Some test cases:

value *"compnet_step* (*1*,*100*) [*1*,*2*::*nat*] = [*1*,*2*]*"*
value *"compnet_step* (*1*,*2*) [*1*,*3*,*2*::*nat*] = [*1*,*2*,*3*]*"*

The whole network operation is now a step-wise fold over the comparators:

definition *run_compnet* :: *"compnet ⇒ 'a :: linorder list ⇒ 'a list"* **where**
"run_compnet = fold compnet_step"

Start by proving that compnets keep the *mset* unchanged.

theorem *compnet_mset*[*simp*]: *"mset* (*run_compnet comps xs*) = *mset xs"*

Sortedness is a bit more difficult. Define a sorting net for lists of length 4 first. Use at most five comparators!

definition *sort4* :: *compnet*
value *"length sort4 ≤ 5"*
value *"run_compnet sort4 [4,2,1,3::nat] = [1,2,3,4]"*

We want to prove that this definition is correct:

lemma *"length ls = 4 ⟹ sorted (run_compnet sort4 ls)"*
oops

However, doing that directly is not easily possible. But we can easily prove that it sorts boolean lists, since there is only a finite number of those.

We use the function *all_n_lists* to obtain a version of the lemma that doesn't contain any free variables, so that *eval* can prove it exhaustively. Then we show that this holds when stated in the more obvious way.

lemma *sort4_bool_exhaust*: *"all_n_lists (λbs::bool list. sorted (run_compnet sort4 bs)) 4"*
— Should be provable *by eval* if your definition is correct!

lemma *sort4_bool*: *"length (bs::bool list) = 4 ⟹ sorted (run_compnet sort4 bs)"*
using *sort4_bool_exhaust[unfolded all_n_lists_def] set_n_lists* **by** *fastforce*

From that, we can show that our networks sorts any list – this is known as the *zero-one principle*. First prove that the sorting does not change when mapped with a monotone function (ctrl+click to see the definition of *mono*).

lemma *compnet_map_mono*:
assumes *"mono f"*
shows *"run_compnet cs (map f xs) = map f (run_compnet cs xs)"*

Now prove the zero-one principle.

Hint: Prove the theorem by contradiction using the properties we have already shown. You will not need an induction. If you are stuck, look for a proof on paper in existing literature (For example from Wikipedia: https://en.wikipedia.org/wiki/Sorting_network).

For local abbreviations within a proof use *let*, as introduced in the lecture.

theorem *zero_one_principle*:
assumes *"⋀bs:: bool list. length bs = length xs ⟹ sorted (run_compnet cs bs)"*
shows *"sorted (run_compnet cs xs)"* (**is** *"sorted ?rs"*)

Finally, sortedness of the *sort4* net follows (for any type).

corollary *"length xs = 4 ⟹ sorted (run_compnet sort4 xs)"*
by (*simp add*: *sort4_bool zero_one_principle*)