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

SS 24
28. 6. 2024

Functional Data Structures

Exercise Sheet 10

Exercise 10.1 Union Function on Tries

Define a function to union two tries and show its correctness:

fun *union* :: *"trie ⇒ trie ⇒ trie"*
lemma *"isin (union a b) x = isin a x ∨ isin b x"*

Exercise 10.2 Tries with 2-3-trees

In the lecture, tries stored child nodes with an abstract map. We want to refine the trie data structure to use 2-3-trees for the map. Note: To make the provided interface more usable, we introduce some abbreviations here:

abbreviation *"empty23 ≡ Leaf"*
abbreviation *"inv23 t ≡ complete t ∧ sorted1 (inorder t)"*

The refined trie datatype:

datatype *'a trie' = Nd' bool "('a×'a trie') tree23"*

Define an invariant for trie' and an abstraction function to trie. Based on the original tries, define membership, insertion, and deletion, and show that they behave correctly wrt. the abstract trie. Finally, combine the correctness lemmas to get a set interface based on 2-3-tree tries.

You will need a lemma like the following for termination:

lemma *lookup_size_aux*[*termination_simp*]:
"lookup m k = Some v ⟹ size (v::'a trie') < Suc (size_tree23 (λx. Suc (size (snd x))) m)"

fun *trie'_inv* :: *"'a::linorder trie' ⇒ bool"*
fun *trie'_α* :: *"'a::linorder trie' ⇒ 'a trie"*
definition *empty'* :: *"'a trie'"* **where**
[*simp*]: *"empty' = Nd' False empty23"*

fun *isin'* :: *"'a::linorder trie' ⇒ 'a list ⇒ bool"*
fun *insert'* :: *"'a::linorder list ⇒ 'a trie' ⇒ 'a trie'"*
fun *delete'* :: *"'a::linorder list ⇒ 'a trie' ⇒ 'a trie'"*

definition *set′* :: *"'a::linorder trie′ ⇒ 'a list set"* **where**
[*simp*]: *"set′ t = set (trie′_α t)"*

lemmas *map23_thms*[*simp*] = *M.map_empty Tree23_Map.M.map_update Tree23_Map.M.map_delete Tree23_Map.M.invar_empty Tree23_Map.M.invar_update Tree23_Map.M.invar_delete M.invar_def M.inorder_update M.inorder_inv_update sorted_upd_list*

interpretation *S′*: *Set*
where *empty* = *empty′* **and** *isin* = *isin′* **and** *insert* = *insert′* **and** *delete* = *delete′*
and *set* = *set′* **and** *invar* = *trie′_inv*
proof (*standard*, *goal_cases*)

Homework 10.1 Tries with accepting leaves (5 points)

Submission until Thursday, July 4, 23:59pm.

Consider the following modified binary trie datatype:

datatype *trie* = *LfR* | *LfA* | *Nd bool* (*trie* × *trie*)

Instead of only one leaf constructor, it has two. The intended behaviour of *LfR* is to act as the previous *Lf* constructor from the lecture does, that i,s when reaching it while checking whether a list is in the trie you answer no. The *LfA* in contrast should be seen as accepting, when it is reached during checking whether a list is in the trie, you answer yes. If one draws an analogy to finite autmata: *LfR* are rejecting trap states, *LfA* are accepting trap states.

Your job in this exercise it to instantiate the set interface for this modified trie. The definitions and proofs will be similar to the ones from *HOL−Data_Structures.Tries_Binary*, so take inspiration from there for your definitions and proofs (Feel free to copy, paste and modify)

The definition of *empty_trie* is unchanged:

empty_trie = *LfR*

Start by defining an *isin_trie* function:

fun *isin_trie* :: *"trie ⇒ bool list ⇒ bool"*

The abstraction function stays unchanged as well:

set_trie t = {*xs*. *isin_trie t xs*}

Your tries should always be fully shrunk. The following invariant captures this:

invar LfR = *True*

invar LfA = *True*

invar (*Nd b* (*l*, *r*)) = (*invar l* ∧ *invar r* ∧ (*l* = *LfR* ∧ *r* = *LfR* ⟶ *b*) ∧ (*l* = *LfA* ∧ *r* = *LfA* ⟶ ¬ *b*))

Define a smart constructor for nodes, which ensures that the resulting trie fulfills the invariant (if its arguments also do).

In order to avoid combinatorial explosions in later proofs, be very careful in unfolding its definition and prefer to prove and use lemmas characterizing its behaviour instead.

definition *node* :: *"bool ⇒ trie ∗ trie ⇒ trie"*

Using your *node* smart constructor, define *insert_trie* and *delete_trie* functions and prove them correct

fun *insert_trie* :: *"bool list ⇒ trie ⇒ trie"*
fun *delete_trie* :: *"bool list ⇒ trie ⇒ trie"*
lemma *set_trie_insert_trie*: *"set_trie(insert_trie xs t) = set_trie t ∪ {xs}"*
lemma *set_trie_delete_trie*: *"set_trie(delete_trie xs t) = set_trie t − {xs}"*

Finally, prove that your functions preserve the invariant and instantiate the locale:

lemma *invar_insert_trie*: *"invar t ⟹ invar(insert_trie xs t)"*
lemma *invar_delete_trie*: *"invar t ⟹ invar(delete_trie xs t)"*
interpretation *S*: *Set*
where *empty* = *empty_trie* **and** *isin* = *isin_trie* **and** *insert* = *insert_trie* **and** *delete* = *delete_trie*
and *set* = *set_trie* **and** *invar* = *invar*

Homework 10.2 Be Creative!

Submission until Thursday, July 11, 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!
- Pick a topic this week and work on it (the regular homework is shorter). Next week, the project will be the exclusive task.
- In total, the homework will yield 15 points (for minimal solutions). Additionally, bonus points may be awarded for particularly nice/original/etc solutions.