Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Mohammad Abdulaziz, Ph.D.

SS 2020
26. 6. 2020

Functional Data Structures

Exercise Sheet 10

Exercise 10.1 Tries with 2-3-trees

In this exercise, you shall develop a trie data structure for keys of type $'a\ list$ (instead of $bool\ list$).

Thus, a node needs to store a map from $'a$ to the next trie.

In a first step, we encode the map as $'a \Rightarrow\ 'b\ option$

datatype $'a\ trie = Leaf \mid Node\ bool\ “'a \rightharpoonup\ 'a\ trie”$

Define and prove correct membership, insertion and deletion (without shrinking the trie).

fun $isin :: “'a\ trie \Rightarrow\ 'a\ list \Rightarrow bool”$

fun $ins :: “'a\ list \Rightarrow\ 'a\ trie \Rightarrow\ 'a\ trie”$

lemma $ins\_correct$: $“isin\ (ins\ as\ t)\ bs = (as{=}bs \lor isin\ t\ bs)”$

fun $delete :: “'a\ list \Rightarrow\ 'a\ trie \Rightarrow\ 'a\ trie”$ **where**

lemma $delete\_correct$: $“isin\ (delete\ as\ t)\ bs = (as \neq bs \land isin\ t\ bs)”$

Now 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 $“empty23 \equiv Tree23.Leaf”$
abbreviation $“inv23\ t \equiv complete\ t \land sorted1\ (inorder\ t)”$

The refined trie datatype

datatype $'a\ trie' = Leaf' \mid Node'\ bool\ “('a \times 'a\ trie')\ tree23”$

Define an invariant for trie' and an abstraction function to trie. Then 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.

fun $trie'\_inv :: “'a{::}linorder\ trie' \Rightarrow bool”$
fun $trie'\_\alpha :: “'a{::}linorder\ trie' \Rightarrow\ 'a\ trie”$

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

fun *delete'* :: "'a::linorder list ⇒ 'a trie' ⇒ 'a trie'"

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*

lemma *ins'_correct*: "trie'_inv t ⟹ (isin' (ins' xs t) ks ⟷ xs=ks ∨ isin' t ks) ∧ trie'_inv (ins' xs t)"
lemma *delete'_correct*: "trie'_inv t ⟹ (isin' (delete' xs t) ks ⟷ xs≠ks ∧ isin' t ks) ∧ trie'_inv (delete' xs t)"

Exercise 10.2 Union Function on Tries

Define a function to merge two tries and show its correctness

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

Homework 10.1 Tries with Same-Length Keys

Submission until Friday, 3. 7. 2020, 10:00am.

Consider the following trie datatype:

datatype *trie* = *LeafF* | *LeafT* | *Node* "trie * trie"

It is meant to store keys of the same length only. Thus, the *Node* constructor stores inner nodes, and there are two types of leaves, *LeafF* if this path is not in the set, and *LeafT* if it is in the set.

Define an invariant *is_trie N t* that states that all keys in *t* have length *N*, and that there are no superfluous nodes, i.e., no nodes of the form *Node* (*LeafF*, *LeafF*).

fun *is_trie* :: "nat ⇒ trie ⇒ bool"

Hint: The following should evaluate to true!

value "is_trie 42 LeafF"
value "is_trie 2 (Node (LeafF,Node (LeafT,LeafF)))"

Whereas these should be false

value "is_trie 42 LeafT"

Wrong key length

value "is_trie 2 (Node (LeafT,Node (LeafT,LeafF)))"

Wrong key length

value *"is_trie 1 (Node (LeafT,Node (LeafF,LeafF)))"*

Superfluous node

Define membership, insert, and delete functions, and prove them correct!

fun *isin* :: *"trie ⇒ bool list ⇒ bool"*
fun *ins* :: *"bool list ⇒ trie ⇒ trie"*
lemma *isin_ins*:
  assumes *"is_trie n t"* **and** *"length as = n"*
  shows *"isin (ins as t) bs = (as = bs ∨ isin t bs)"*
    and *"is_trie n (ins as t)"*

fun *delete2* :: *"bool list ⇒ trie ⇒ trie"* **where**
lemma
  assumes *"is_trie n t"*
  shows *"isin (delete2 as t) bs = (as≠bs ∧ isin t bs)"*
    and *"(is_trie n (delete2 as t))"*

Hints:

- Like in the *delete2* function for standard tries, you may want to define a "smart-constructor" *node* :: *trie × trie ⇒ trie* for nodes, that constructs a node and handles the case that both successors are *LeafF*.
- Consider proving auxiliary lemmas about the smart-constructor, instead of always unfolding it with the simplifier.

Homework 10.2 Enumeration of Keys in Tries

Submission until Friday, 3. 7. 2020, 10:00am.

Write a function that enumerates all keys in a trie, in lexicographic order! Prove it correct.

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

  lemma *enum_correct*:
    "set (enum t) = { xs. isin t xs }" **and** *"sorted_wrt (<) (enum t)"*

Note that Booleans are ordered by *False* < *True*, and that we imported *HOL−Library.List_Lexorder*, which defines a lexicographic ordering on lists, if the elements are ordered.

value *"[True,True,False] < [True,True,True,True]"*

Homework 10.3 Be Original!

Submission until Friday, 17. 7. 2020, 10:00am.

Develop a nice Isabelle formalisation yourself!

- This homework goes in parallel to other homeworks for the rest of the lecture period. From next sheet on, we will reduce regular homework load a bit, such that you have a time-frame of 3 weeks with reduced regular homework load.
- This homework will yield 15 points (for minimal solutions). Additionally, up to 15 bonus points may be awarded for particularly nice/original/etc solutions.
- You may develop a formalisation from all areas, not only functional data structures.
- Document your solution, 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 after 3 weeks.
- You are welcome to discuss the realisability of your project with the tutor or ask him for possible ideas!
- Should you need inspiration to find a project: Sparse matrices, skew binary numbers, arbitrary precision arithmetic (on lists of bits), interval data structures (e.g. interval lists), spatial data structures (quad-trees, oct-trees), Fibonacci heaps, prefix tries/arrays and BWT, etc.