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

SS 24
14. 6. 2024

Functional Data Structures

Exercise Sheet 8

Exercise 8.1 Round wrt. Binary Search Tree

The distance between two integers x and y is $|x - y|$.

1. Define a function *round* :: *int tree* ⇒ *int* ⇒ *int option*, such that *round t x* returns an element of a binary search tree t with minimum distance to x, and *None* if and only if t is empty.

   Define your function such that it does no unnecessary recursions into branches of the tree that are known to not contain a minimum distance element.
2. Specify and prove that your function is correct. Note: You are required to phrase the correctness properties yourself!

   Hint: Specify 3 properties:

   - None is returned only for the empty tree.
   - Only elements of the tree are returned.
   - The returned element has minimum distance.
3. Estimate the time of your round function to be linear in the height of the tree Note: If you define any non-recursive helper functions, assume them to have cost *0*, by using the *time_fun_0* command

fun *round* :: *"int tree ⇒ int ⇒ int option"*

Exercise 8.2 Interval Lists

Sets of natural numbers can be implemented as lists of intervals, where an interval is simply a pair of numbers. For example the set {*2*, *3*, *5*, *7*, *8*, *9*} can be represented by the list [(*2*, *3*), (*5*, *5*), (*7*, *9*)]. A typical application is the list of free blocks of dynamically allocated memory.

We introduce the type

type_synonym *intervals* = *"(nat∗nat) list"*

Next, define an *invariant* that characterizes valid interval lists: For efficiency reasons intervals should be sorted in ascending order, the lower bound of each interval should be less than or equal to the upper bound, and the intervals should be chosen as large as possible, i.e. no two adjacent intervals should overlap or even touch each other. It turns out to be convenient to define *inv* in terms of a more general function such that the additional argument is a lower bound for the intervals in the list:

fun *inv'* :: *"nat ⇒ intervals ⇒ bool"*
definition *inv* **where** *"inv ≡ inv' 0"*

To relate intervals back to sets define an *abstraction function*

fun *set_of* :: *"intervals ⇒ nat set"*

Define a function to add a single element to the interval list, and show its correctness

fun *add* :: *"nat ⇒ intervals ⇒ intervals"*
lemma *add_correct_1*:
"inv is ⟹ inv (add x is)"
lemma *add_correct_2*:
"inv is ⟹ set_of (add x is) = insert x (set_of is)"

Hints:

- Sketch the different cases (position of element relative to the first interval of the list) on paper first
- In one case, you will also need information about the second interval of the list. Do this case split via an auxiliary function! Otherwise, you may end up with a recursion equation of the form $f\ (x\#xs) = \ldots\ \textit{case xs of } x'\#xs' \Rightarrow \ldots\ f\ (x'\#xs')$ ... combined with *split*: *list.splits* this will make the simplifier loop!

Homework 8.1 Bit-Vectors

Submission until Thursday, 20.06.23, 23:59pm.

A bit-vector is a list of Booleans that encodes a finite set of natural numbers as follows: A number i is in the set, if i is less than the length of the list and the ith element of the list is true. Your bit vector must only be as long as necessary. That means that the abstraction function is:

$\textit{set\_bv bs} = \{i.\ i < \textit{length bs} \wedge \textit{bs}\ !\ i\}$

Define the other operations of the Set interface (including delete) and interpret the locale!

Hints:

- Compose existing functions rather than defining your own. The definitions in the example solution do not even need a single case distinction!
- (As often) delete is the hardest operation

- The syntax to update the n-th element of a list is: $xs[n := x]$.
- Your interpretation proof should start with *unfold_locales*. To get a clickable Isar template for your proof, start it with:
  proof(*unfold_locales*,*goal_cases*).

interpretation *bv_set*: *Set* — Your parameters here

Homework 8.2 Deletion from a disjoint interval tree

Submission until Thursday, 20.06.23, 23:59pm.

An interval tree is a tree whose nodes each contain an interval of elements from an ordered type. We can define its type as follows:

datatype *'a itree* = *iLeaf* | *iNode* (*'a itree*) (*'a* × *'a*) (*'a itree*)

The following are two useful functions for interval trees: one returning the set of intervals in the tree, and another returning the set of elements in the tree.

set_itree2 iLeaf = {}

set_itree2 (*iNode l* (*low*, *high*) *r*) = {*low*..*high*} ∪ (*set_itree2 l* ∪ *set_itree2 r*)

set_itree3 iLeaf = {}

set_itree3 (*iNode l* (*low*, *high*) *r*) = {(*low*, *high*)} ∪ (*set_itree3 l* ∪ *set_itree3 r*)

An ordered disjoint interval tree is an interval tree such that:

- The lower end of an interval in a node is strictly greater than the higher end of every interval in the left subtree.
- The higher end of an interval in a node is strictly smaller than the lower end in every interval in the right subtree.
- The interval in every node has a lower end that is smaller than or equal than its upper end.

Recursively define an invariant for an interval tree that formalises the above conditions.

fun *ibst* :: *"'a::linorder itree ⇒ bool"*

Define and verify a delete function for interval trees. That function should: i) only take an element (i.e. not an interval) and delete it from the tree, ii) exploit the fact that the tree is ordered, and iii) be implemented using an appropriate join function for interval trees.

fun *delete* :: *"int ⇒ int itree ⇒ int itree"*

Hint: this function has to deal with three cases.

- if the element is equal to the two ends of an interval, in which case the interval should be completely removed from the tree,

- if the element is equal to one of the ends of an interval, in which case the interval has to be appropriately shrank, and
- if the element lies with an interval, in which case the interval has to be split into two. One way to deal with that situation is to let the left subinterval inherit the position of the original interval, and position the right subinterval to be the left most leaf in the right subtree.

Prove that the function removes the correct element from the tree.

lemma *delete_set_minus*: *"ibst t ⟹ set_itree2 (delete x t) = (set_itree2 t) − {x}"*

Prove that the resulting interval tree conforms to the invariant. Hint: you might want to define a function that returns an list of intervals in an interval tree and a predicate characterising the sortedness of that list. Proving that *delete* preserves the invariant should reduce to arguing about the sortedness of this list.

lemma *ibst_delete*: *"ibst t ⟹ ibst (delete x t)"*