COMP4161 T3/2024
Advanced Topics in Software Verification

Assignment 3

This assignment is released on November 8th, and is due on November 19th 17:59:59. We will accept Isabelle theory (.thy) files only. You are allowed to make late submissions up to five days (120 hours) after the deadline, but at a cost: -5 marks per day.

The assignment is take-home. This does NOT mean you can work in groups. Each submission is personal. For more information, see the plagiarism policy: https://student.unsw.edu.au/plagiarism

Submit using `give` on a CSE machine:

```
give cs4161 a3 files ...
```

For example:

```
give cs4161 a3 a3_pt1.thy a3_pt2.thy
```

The second part of this assignment requires Isabelle2024 and AutoCorres 1.11. AutoCorres releases can be fetched from github:

https://github.com/seL4/l4v/releases

AutoCorres requires a Unix-based operating system. It does not support native Windows. Linux, Mac, and Windows WSL should work.

After extracting the `autocorres-1.11.tar.gz` archive, load the template theory files via e.g.

```
L4V_ARCH=ARM isabelle jedit -d <path-to-autocorres-1.11> -l AutoCorres a3_pt1.thy
```

For this assignment, all proof methods and proof automation available in the standard Isabelle distribution is allowed. This includes, but is not limited to, `simp`, `auto`, `blast`, `force`, and `fastforce`.

However, if you're going for full marks, you shouldn't use "proof" methods that bypass the inference kernel, such as `sorry`. We *may* award partial marks for plausible proof sketches where some subgoals or lemmas are sorried.

If you use `sledgehammer`, it's important to understand that the proofs suggested by `sledgehammer` are just suggestions, and aren't guaranteed to work. Make sure that the proof suggested by `sledgehammer` actually terminates on your machine (assuming an average spec machine). If not, you can try to reconstruct the proof yourself based on the output, or apply a few manual steps to make the subgoal smaller before using `sledgehammer`.

Note: this document contains a quick explanation of the problems and your assignment tasks. You should examine the associated Isabelle theory files. Some of the relevant definitions are constructed automatically by the AutoCorres framework. Remember to use the query mechanisms (e.g. `find_theorems`) built into Isabelle to explore the set of facts that have been defined for you.

Tree Heap operations for a Priority Queue

A tree-heap (also just called a heap), is a tree-shaped data structure. It is typically used to implement a prioritised queue. The tree is sorted top to bottom, or from root to leaves. Unlike a search tree, there is no particular order on the left and right element at any node. The highest-priority element, which is also the head of the queue, is always at the root of the tree.

The queue head can be queried or fetched in $O(1)$ time, and a new element can be pushed into the queue in $O(log(n))$ time if the tree is balanced.

A new element is placed at the root and then pushed into the tree/queue to restore sorted order. This process repeatedly swaps the position of the new element further into the tree, each time swapping out the element which has the highest priority. The process stops once the new element is higher priority than any of its children.

You can read more about the heap structure online, for instance at the wikipedia article:
https://en.wikipedia.org/wiki/Heap_(data_structure)

Constant Shape

In an unusual use-case, we provide an operation that pops the current head element and then immediately pushes a replacement. Equivalently, we provide an operation that changes the current head element to some other value (with a possibly lower priority) and then pushes it into the heap to reach its correct priority position.

This operation can be implemented without changing the tree and pointer structure. Instead, the values stored in the tree nodes are swapped to the correct locations.

This implementation is motivated by the idea of processing the requests of a collection of clients waiting to perform some operations on a shared resource. The clients are kept in a queue. The operations take variable time to perform, so, a client that performs a quick operation may not be sent to the very back of the queue. Instead, its priority in this priority queue is decreased proportional to the cost of its operation. This is implemented by recording a "delay" value which is the negative of the priority. It stores the approximate resource-use of the client so far, or, the approximate delay before it will be selected again.

A simple C implementation of this operation is given in `heap.c`. A functional implementation is given in the supplied part 1 theory. In the first part of the assignment, you will prove that this implementation maintains the sortedness invariant. In the supplied part 2 theory, the C parser and AutoCorres are used to introduce a monadic abstraction of the C implementation, on which you will prove some key correctness properties that would form part of a completed verification.

1 Question 1

The tree datastructure represents a binary tree with values in the nodes.

```
datatype 'a tree =
    Node 'a 'a tree 'a tree
  | Empty

fun tree-elements :: 'a tree ⇒ 'a list
  where
    tree-elements Empty = []
  | tree-elements (Node x left right) =
    x # tree-elements left @ tree-elements right


fun heap-tree-sorted :: ('a ⇒ 'a ⇒ bool) ⇒ 'a tree ⇒ bool
  where
    heap-tree-sorted ord Empty = True
  | heap-tree-sorted ord (Node x left right) =
    ((∀ y ∈ set (tree-elements left). ord x y) ∧
        (∀ y ∈ set (tree-elements right). ord x y) ∧
        heap-tree-sorted ord left ∧ heap-tree-sorted ord right)
```

The *tree-elements* function lists all elements of a tree, in prefix and left-to-right order. The *heap-tree-sorted* predicate is the sortedness invariant for heap trees.

Q1.1: Show that sortedness of the list of elements implies heap-tree sortedness (6 marks).

```
fun tree-top :: 'a tree ⇒ 'a
  where
    tree-top (Node x - -) = x

fun tree-set-top :: 'a tree ⇒ 'a ⇒ 'a tree
  where
    tree-set-top Empty - = Empty
  | tree-set-top (Node - left right) x = Node x left right


fun heap-tree-push :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a tree ⇒ 'a tree
  where
    heap-tree-push ord x Empty = Empty
  | heap-tree-push ord x (Node - left right) =
    (if left ≠ Empty ∧ ord (tree-top left) x ∧
          (right = Empty ∨ ord (tree-top left) (tree-top right))
        then Node (tree-top left) (heap-tree-push ord x left) right
    else if right ≠ Empty ∧ ord (tree-top right) x
        then Node (tree-top right) left (heap-tree-push ord x right)
    else Node x left right)
```

The *tree-top* and *tree-set-top* operations access the value at the root (or top) of the tree.

The *heap-tree-push* function defines the push-into operation for the functional implementation of the tree heap.

Q1.2: Show that this push operation preserves the structure/shape of the tree (8 marks).

This lemma makes use of the *rel-tree* constant that was automatically defined by the datatype declaration of trees. You can use the find-theorems mechanism to find out the properties that are true of such constants.

Q1.3: Show that the push operation preserves the collection of elements of the tree (8 marks).

Q1.4: Show that the push operation maintains the sortedness invariant. (18 marks).

You may find that this proof requires some care. The automatic tools tend to generate a lot of cases and diverge. In particular, the simplifier can break up the goal into excessively many similar parts when it uses the automatic split rule for if. The supplied theory describes how to deactivate this automatic rule and apply it as a single-step action instead. We recommend you experiment with that approach.

2 Question 2

Part 2 of this assignment uses the C parser to import the C code of the heap push into Isabelle, and AutoCorres to define a monadic version.

Many types and constants are defined automatically as part of this process. You can use the find-theorems mechanism to find out more about them.

The type *tree-heap-C* is defined by the C parser to represent elements of the C struct type. Again, you can use the find-theorems mechanism to examine some of the automatically-proven facts about it.

Q2.1: Prove that two values of type *tree-heap-C* are equal if each of their four structure fields are equal. (6 marks)

type-synonym *th-dom* = (*tree-heap-C ptr* ⇒ *bool*)
type-synonym *th-vals* = (*tree-heap-C ptr* ⇒ *tree-heap-C*)

AutoCorres abstracts the part of the C general-purpose memory (also called a heap) which contains heap-tree struct element into two functions. The first is a domain predicate which says which pointers are currently valid. The second is a lookup function from pointers to structure values. The *th-dom* and *th-vals* type synonyms record the types of these functions.

inductive *tree-in-C* :: *th-dom* ⇒ *th-vals* ⇒ *tree-heap-C ptr* ⇒ *bool*
where
tree-in-C-NULL:
tree-in-C th-dom th-vals NULL
| *tree-in-C-Node*:
p ≠ *NULL* ⟹ *th-dom p* ⟹
tree-in-C th-dom th-vals (*left-C* (*th-vals p*)) ⟹
tree-in-C th-dom th-vals (*right-C* (*th-vals p*)) ⟹
tree-in-C th-dom th-vals p

The *tree-in-C* inductive predicate captures the cases where a finite tree exists starting at a particular pointer. Its arguments are the domain and lookup functions which represent the *tree-heap-C* memory. This predicate excludes cases where the pointer structure loops back on itself and thus the tree datatype would be infinite.

function *get-tree* :: *th-dom* ⇒ *th-vals* ⇒ *tree-heap-C ptr* ⇒
(*tree-heap-C ptr* × *word32* × *unit ptr*) *tree*
where
get-tree th-dom th-vals NULL = *Empty*
| *p* ≠ *NULL* ⟹
get-tree th-dom th-vals p = (*if tree-in-C th-dom th-vals p*
then Node (*p*, *delay-C* (*th-vals p*), *data-C* (*th-vals p*))
(*get-tree th-dom th-vals* (*left-C* (*th-vals p*)))
(*get-tree th-dom th-vals* (*right-C* (*th-vals p*)))
else Empty)
by *auto auto*

The *get-tree* function is proven to terminate based on the induction principle for the *tree-in-C* predicate. You don't have to understand the termination proof in detail.

Q2.2: Prove that an update to the memory lookup function that does not change the *left-C* or *right-C* fields preserves the *tree-in-C* property (8 marks).

The *set-top′* function is the AutoCorres-created abstraction of the C function which encodes a simple set of the top/root values of the tree.

Q2.3: Prove that *set-top′* is safe to execute (does not fail), and that it preserves the *tree-in-C* property at some other pointer *p2* (10 marks).

Q2.4: Prove that *min-delay′* is safe to execute, and preserves any property Q of the global state. You are given the lemma statement with some *FIXME* elements, which you will need to adjust. (8 marks)

Q2.5: Prove that the *push-down′* operation is safe. This is a more substantial task. You are given a skeleton lemma proof. It makes use of your lemma about *min-delay′*, so you will need a provable precondition for this proof to work. We expect that you will need to prove other lemmas about *tree-in-C*, *get-tree* and others to complete this proof (18 marks).

Q2.6: Prove that *set-top′* performs the same operation in the C heap as *heap-set-top* does in the functional definition (10 marks).

We have now seen a little bit of C verification!

A bigger task would be to prove that the *push-down′* operation produces the correct output tree, in the same sense as Q2.6 examines *set-top′*. That proof would be quite a bit more involved, and for this assignment you don't need to do that. We have already introduced all the relevant concepts, however, so you are welcome to explore the problem if you are interested in a challenge.