# Advanced Topics in Formal Methods and Prog. Languages
# – Software Verification with Isabelle/HOL –

## Assignment 3

ver 1.0

---

**Submission Guidelines**

- Due time: Oct 25, 2024, 6pm (Canberra Time)
- Submit via Wattle.
- Accepted formats are plain text (.txt) files, PDF (.pdf) files, and Isabelle theory (.thy) files.
- Isabelle files should be executable (a template is provided on the course webpage).
- Please read and sign the declaration on the last page and attach a copy to your submission.
- **No late submission, deadline is strict.**

---

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 are going for full marks, you should not 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.

For all questions, you may prove your own helper lemmas, and you may use lemmas proved earlier in other questions. You can also use automated tools like sledghammer. If you can't finish an earlier proof, use sorry to assume that the result holds so that you can use it if you wish in a later proof. You won't be penalised in the later proof for using an earlier true result you were unable to prove, and you'll be awarded partial marks for the earlier question in accordance with the progress you made on it.

## Exercise 1 (AVL Trees) (50 Marks)

In this exercise we extend our analysis of trees started in Assignment 2. Please note, that we provide all definitions and lemmas needed so that this exercise is self-contained. Don't forget that splitting lemmas such as `tree.splits` could be useful.

In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. This exercise aims to verify that the following insert function is correct.

```
fun avl_insert :: "'a::linorder => 'a tree => 'a tree" where
  "avl_insert x Leaf = Branch Leaf x Leaf" |
  "avl_insert x (Branch l y r) =
     (if x < y then
        (let new_l = avl_insert x l in
```

```
         let balanced_tree = Branch new_l y r in
         if factor balanced_tree < -1 then
           if factor new_l <= 0
           then rotate_right balanced_tree
           else rotate_left_right balanced_tree
        else balanced_tree)
      else if x > y then
            (let new_r = avl_insert x r in
            let balanced_tree = Branch l y new_r in
            if factor balanced_tree > 1 then
                if factor new_r >= 0
                then rotate_left balanced_tree
                else rotate_right_left balanced_tree
        else balanced_tree)
      else
        Branch l y r)"
```

We will work through the definitions in the following.

We use the definition of `tree`, function `nrl_list`, and corresponding functions from the last assignment. An example is a useful helper function

**lemma** *insort-lt:*

*"∀y ∈ set xs. x > y ⟹ insort x (xs @ ys) = xs @ (insort x ys)"*

**by** *(induct xs arbitrary: x; fastforce)*

### Question 1: Rotating Trees (14 marks)

A crucial definition within the context of AVL trees is rotation. We rotate a node from right to left by the following function. We rotate a node from right to left by the following function.

**definition** *rotate-right :: "'a::linorder tree ⇒'a tree"* **where**

*"rotate-right b = (case b of*

*Branch (Branch ll y lr) x r ⇒Branch ll y (Branch lr x r)*

*| _ ⇒b)"* The function **rotate-left** is defined in a similar fashion.

(a) Show that rotating left and then rotating right is the identity, under some weak circumstances. Try to find the weakest assumption. Explain in a short sentence why this assumption is necessary (e.g. provide a counterexample), and useful.

(b) Using the flatten-function `lnr_list` from the last assignment, prove that rotating does not change the order of leaves.

We now lift single rotation to a double-rotation construct.

**definition** *rotate-right-left :: "('a::linorder) tree ⇒'a tree"* **where**

*"rotate-right-left b = (case b of*

*Branch l x (Branch (Branch rl y rr) z r)*

*⇒rotate-left (Branch l x (rotate-right (Branch (Branch rl y rr) z r)))*

*| _ ⇒b)"* Again, we define a symmetric (*rotate-left-right*) one as well.

(c) Create a similar lemma to `lnr_list_rotate` (Question (b)), featuring double rotation `rotate_left_right` and `rotate_right_left`, respectively.

**Question 2: Balanced Trees** (10 marks)

The AVL tree ensures that the height difference (balance factor) between the left and right subtrees of any node is at most 1.

(d) Using the height-function, define a function `balanced` that checks whether a given tree (input) is balanced.

(e) Prove that under some weak precondition, rotation balances a tree. (The lemmas can be found in the Isabelle file.) **Hint:** Similar lemmas for double-rotation may come in handy later on.

**Question 3: Verification** (25 marks)

We are now turning towards the main theorem of this exercise, the verification of the algorithm presented above.

(f) Describe the algorithm in a few sentences. (Feel free to use example trees)

We require a couple of auxiliary lemmas to complete the verification task.

(g) Using the function `ordered` from the last assignment, show that `avl_insert` inserts the new element at the right position.

**lemma** *lnr-list-avl-insert:*
*"⟦ordered t; x ∉set-tree t⟧*
*⟹ lnr-list (avl-insert x t) = insort x (lnr-list t)"*

To reason about balanced tree, we require lemmas that relate the function `avl_insert` to the height of the tree.

(h) Prove *"balanced t ⟹height t ≤ height (avl-insert x t)"*

(i) The previous lemma gives a lower bound of the tree height. Provide a (strict) upper bound.

And finally,

(j) Prove that the algorithm is functionally correct. Please note that this proof could be potentially long.

*"balanced tree ⟹ balanced (avl_insert x tree)"*

## Exercise 2 (Stack) (50 Marks)

This exercise should be completed using **Isabelle2023 and AutoCorres 1.10.** `https://github.com/seL4/l4v/releases/download/autocorres-1.10/autocorres-1.10.tar.gz`.
You will need a Unix-based machine, AutoCorres does not support native Windows. Linux, Mac, and Windows WSL should work. After extracting the autocorres-1.10.tar.gz archive, load the template theory files via e.g.

```
L4V_ARCH=ARM isabelle jedit -d <path-to-autocorres-1.10> -l AutoCorres a3.thy
```

In this question we will be verifying a simple stack implementation in C. The objective is to familiarise yourself with proofs about imperative programs and reasoning about finite machine word arithmetic in C.

The file `stack.c` contains a global array `content` of length `LEN` storing the contents of the stack (of type `unsigned int`). The global variable `top` is the index of the top-most element of the stack when the stack contains elements and -1 otherwise. Note that `top` is an `unsigned int`, which means that -1 is the same as `MAX_INT`.

To reason about the C functions, the assignment template defines an abstraction predicate `is_stack xs s` that is true if and only if the list `xs` contains the contents of the global stack in state `s`. The definition is based on the recursive definition `stack_from xs n s` that starts looking at the stack not from the top, but from index `n` instead.

After processing by AutoCorres, the template opens the context stack, in which monadic versions of the C functions are available under names ending with `'`, for instance `pop'` for the C function `pop` and so on. The global state is an Isabelle record with fields `top_''` and `contents_''`. The `contents_''` field is of Isabelle type `array`. Array types are written `t[n]` where `t` is the element type, and `n` is the size of the array. The type provides an `Arrays.index` function to access fields and an `Arrays.update` function to update elements. `Array.index` a i is written `a.[i]`. Use `find_theorems` to discover rules about the `array` type.

Finally, the C program operates on finite machine words, but some of our predicates operate on natural numbers. The function `unat` converts a machine word into a natural number. The operators $<$ and $\leq$ on machine words can also be expressed via `unat`. Use `find_theorems` to discover rules about `unat` and its interactions with operators on natural numbers.

We begin the proof by showing same basic properties of the abstraction predicates:

(a) `is_stack [] s = (top_'' s = - 1)`

(b) `is_stack [] s = (is_empty' s = 1)`

(c) `stack_from xs (- 1) s = (xs = [])`

(d) `is_stack [x] s = (top_'' s = 0 ∧ content_'' s.[0] = x)`

(e) `is_stack (x # xs) s =`
`(top_'' s < LEN ∧`
`content_'' s.[unat (top_'' s)] = x ∧ stack_from xs (top_'' s - 1) s)`

For C functions that change the state, we will want to know under which changes the predicate remains the same.

(f) The `stack_from` predicate takes the index as a parameter and therefore does not depend on the value of the variable `top_''`:

`stack_from xs n (s ⦇top_'' := t ⦈)= stack_from xs n s`

(g) The `stack_from` predicate also does not change if we update the array at an index that is outside of the range `0..n`, for instance at `n+1`.

```
nat (n + 1) < LEN ⟹
stack_from xs n
    (s ⦇(top_'' := n+1, content_'' := update (content_'' s) (unat (n + 1)) x ⦈)
  = stack_from xs n s
```

The template contains an optional lemma that might help with induction over the `xs`.

We are now ready to prove properties of the C functions.

(h) Complete the Hoare logic statement in the assignment template and prove partial correctness of `pop'`.

(i) Complete the Hoare logic statement in the assignment template and prove total correctness of `pop'`, using the $\{\!|\_|\!\}!$ syntax, instead of $\{\!|\_|\!\}$ . Total correctness means you will also have to show all side conditions that could lead to undefined behaviour in C.

(j) Complete the Hoare logic statement in the assignment template and prove total correctness of `push'`.

(k) Prove partial correctness of `sum'`, which empties the stack and sums up all of its elements. The Isabelle function `sum_list xs` in the template stands for the sum of all elements of `xs`. It is easier in this proof to unfold the definition of `pop'` again than to use the previous correctness lemma.