# COMP4161 S2/2018
# Advanced Topics in Software Verification

## Assignment 3

This assignment starts on Friday, 2018-10-05 and is due on Monday, 2018-10-22, 8am. We will accept Isabelle .thy files only. In addition to this pdf document, please refer to the provided Isabelle templates for the definitions and lemma statements.

The assignment is take-home. This does NOT mean you are allowed to 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 a3.thy`

For all questions, you may prove your own helper lemmas, and you may use lemmas proved earlier in other questions. 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 are yet to prove, and you'll be awarded part marks for the earlier question in accordance with the progress you made on it.
You are allowed to use sledgehammer in this assignment.

## 1 Regular Expression Matching (35 marks)

In the lecture and previous assignments we proved various properties about regular expressions. In this assignment, we will prove correctness of a regular expression matcher, i.e. of a function that takes a regular expression and a string and that decides whether the string is in the language of the expression or not.
The template gives a simple, elegant, and inefficient implementation of such a matcher from the following website: https://tinyurl.com/yapdvt8n. See the website for more information on how it works. The regular expressions it operates on are the same as in the lecture, but with `Null` (match nothing) and `One` (match the empty string) instead of negation.

(a) (8 marks) Prove termination of the `matches` function. You can

- either adjust the check `cs' ≠ cs` in the function to something that provides a termination condition that is easier to prove, but in a way that does not change the behaviour of the function. This is the easier option.
- or prove termination directly without changing the function. This is the more challenging option.

You can tell whether an adjustment changed the function's behaviour by checking whether the lemma `matches_correct` still holds.

(b) (27 marks) Prove correctness of the matcher:

```
matches r cs (op = []) = (cs ∈ lang r)
```

The syntax `(op = [])` is the function that takes a list and returns true if and only if the list is empty. Hint: remember that induction on functions often works better when using the function's induction rule. Also remember that to be able to apply such a rule, you might need to generalise your lemma first.

## 2 Binary Search (65 marks)

In the very first lecture of this course, we motivated the field of software verification by showing the code of binary search from the Java standard library, which contained a bug. [1] We transcribed that Java code into C (see the file `binsearch.c`), preserving the bug. The task in this question is to find the preconditions under which the code works correctly and to prove that it does so.

The template uses the C parser and AutoCorres to convert the C code into a monadic specification in Isabelle. We will prove properties about this AutoCorres output.

Namely, we aim to prove the following lemma:

```
lemma binary_search_correct:
  {|λs. sorted (array_list (heap_w32 s) a len) ∧
        valid_array (is_valid_w32 s) (heap_w32 s) a len ∧
        TODO |}
   binary_search' a len key
   {| λr s. (r < 0 ⟶ key ∉ set (array_list (heap_w32 s) a len)) ∧
            (0 ≤ r ⟶ r < len ∧ (array_list (heap_w32 s) a len ! nat r) = key)|}!
 unfolding binary_search'_def
  apply(subst whileLoopE_add_inv[where
              I=λ(high,low) s. valid_array (is_valid_w32 s) (heap_w32 s) a len ∧
                               sorted (array_list (heap_w32 s) a len) ∧
                               TODO1 and
              M=λ((high,low),_). TODO2])
```

The task is to define the `array_list` and `valid_array` functions, find the suitable extra precondition `TODO`, find a suitable loop invariant `TODO1` and a suitable decreasing variant `TODO2`. The variant is needed because we are here proving total correctness (denoted by the exclamation mark in `{|P|} c {|Q|}!`).

(a) The code operates on an array of signed integers (`int []`), i.e. the array is of type `s_int ptr` once formalised. We want to reason about the list of addresses (pointers) that the array contains (to reason about their validity), as well as reason about the values they point to in the memory heap.

For this we will use a function that enumerates the addresses that a signed int array contains:

```
array_addrs p 0 = []
array_addrs p (Suc len) = p # array_addrs (p +p 1) len
```

Prove the following two lemmas:

```
length (array_addrs a len) = len
```

(2 marks)

```
⟦0 ≤ x; nat x < len⟧ ⟹ array_addrs a len ! nat x = a +p x
```

(5 marks)

(b) We will have to deal with the fact that the C heap stores unsigned words, but the program mostly uses signed C ints. Signed and unsigned pointers can be converted into each other using the function `ptr_coerce`.

Using `array_addrs`, define a function `array_list` that takes an unsigned int heap (a function from `u_int ptr` to `u_int`), the array base address (a signed int pointer), and a length (an Isabelle `int`), and returns the elements of the array as a list. (5 marks)

(c) Prove the following lemma about `array_list`:

```
⟦0 ≤ x; x < len⟧
⟹ uint (heap_w32 s (ptr_coerce a +p x)) =
    array_list (heap_w32 s) a len ! nat x
```

(2 marks)

[1] See https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html for more info on this bug.

(d) Using `array_addrs` again, define a function `valid_array` that takes a heap validity predicate such as `is_valid_w32` in the AutoCorres output `binary_search'`, and asserts that each array address is a valid C pointer. The function should also place a condition on the values that the array stores. Find and add this condition. This extra condition requires `array_addrs` to also take the unsigned int heap as parameter. Hint: it is probably easier to explore the invariant and proof obligations of the program first before you add the additional condition. If it's easier for you, you can define two functions, `valid_array1` asserting the validity of each array address and `valid_array2` asserting this extra condition, with `valid_array` being the conjunction of the two (see template) (6 marks)

(e) Prove the following lemmas about sorted lists:

```
⟦key < xs ! nat mid; mid - 1 < x; sorted xs; 0 ≤ mid; x < int (length xs)⟧
⟹ key < xs ! nat x
```

(5 marks)

```
⟦xs ! nat mid < key; 0 ≤ x; x ≤ mid; sorted xs; mid < int (length xs)⟧
⟹ xs ! nat x < key
```

(5 marks)

(f) We can now look again at our main correctness theorem. Firstly you have to strengthen the pre-condition with additional conditions such that the post-condition of `binary_search'` is provable. Think about which precondition is needed to avoid triggering the overflow bug that is present in the code. (5 marks)

(g) Adjust the invariant with any additional conditions you need to prove the lemma. (10 marks)

(h) Prove correctness of the binary search implementation. (15 marks)