Final exam, February 28, 2023

- Duration: 3 hours.
- Answers may be written in English or French.
- **Please write your answers for Exercise 1 on a separate piece of paper.**
- Lecture notes and personal notes are allowed. **Mobile phones must be switched off.** Electronic notes are allowed, but **all network connections must be switched off**, and the use of any electronic device must be restricted to reading notes: no typing, no using proof-related software.
- There are 4 pages and **3** exercises. Exercise 1 is about verification using weakest preconditions. Exercises 2 to 3 are about separation logic.
- Write your name, and page numbers under the form 1/7, 2/7, etc., on each piece of paper.

1 Counting Occurrences of Elements in an Array

In this exercise, we are interested in counting the occurrences of elements in an array. We start by specifying a logic function `occ` which, given an element `x`, a function `f` and two integers `i` and `j`, denotes the number of occurrences of `x` among `f i`, `f (i+1)`, ..., `f (j-1)`. Notice than the first index `i` is included in the count but not the last one `j`. It is defined as follows

```
let rec ghost function occ (x:int) (f:int → int) (i j:int) : int
= if j <= i then 0 else (if f i = x then 1 else 0) + occ x f (i+1) j
```

Question 1.1. *Which annotations (e.g. pre-conditions, invariants, etc.) should be given to the ghost function above so as to be able to prove it safe and terminating? Justify informally (5 lines max) why your annotations suffice.*

The following program computes the number of occurrences in an array, with a while loop.

```
let count_occ (x:int) (a:array int) : int =
  let ref n = 0 in let ref i = 0 in
  while i < a.length do
    if a[i] = x then n ← n+1;
    i ← i + 1;
  done;
  n
```

Question 1.2. *Which annotations should be given to the program above so as to be able to prove it safe and terminating? Justify informally (10 lines max) why your annotations suffice.*

Question 1.3. *We now express the expected behavior of the program* `count_occ` *by adding the post-condition* `result = occ x a.elts 0 a.length`*. Which extra annotations are needed to prove it? If you need an additional lemma to achieve the proof, state it clearly and explain how it can be proved (20 lines max).*

We know consider the classical function for swapping two elements in an array as follows.

```
let swap (a:array int) (i j :int) : unit
  requires { 0 <= i < j < a.length }
  writes { a }
  ensures { ∀ x:int. occ x (old a).elts 0 a.length = occ x a.elts 0 a.length }
= let tmp = a[i] in a[i] ← a[j]; a[j] ← tmp
```

Notice the specific post-condition expressing that the number of occurrences of elements in array `a` are unchanged. Notice also that for simplicity we assume `i < j`.

Question 1.4. *To achieve a proof of the post-condition of* `swap` *above, several lemmas should be proved first. Identify what could be these lemmas. State them clearly with logic formulas, and explain why they suffice to prove the post-condition. (20 lines max)*

Question 1.5. *Explain how the lemmas identified above can be proved correct. (20 lines max)*

2 Separation Logic: heap predicates

We recall the definition of a few separation logic connectives:

$$H_1 \mathbin{\wedge\!\!\!\wedge} H_2 \equiv \lambda m.\ H_1\, m \wedge H_2\, m \qquad l \mapsto v \equiv \lambda m.\ m = \{(l,v)\} \wedge l \neq \mathsf{null} \qquad \ulcorner P \urcorner \equiv \lambda m.\ m = \varnothing \wedge P$$

$$H_1 \mathbin{\vee\!\!\!\vee} H_2 \equiv \lambda m.\ H_1\, m \vee H_2\, m \qquad H_1 * H_2 \equiv \lambda m.\ \exists m_1 m_2.\, m = m_1 \uplus m_2 \wedge H_1\, m_1 \wedge H_2\, m_2$$

$$\exists x.\, H \equiv \lambda m. \exists x. H m \qquad \mathsf{GC} \equiv \lambda m.\ \mathsf{True} \qquad H_1 \mathrel{-\!\!*} H_2 \equiv \lambda m.\ \forall m_1\ (m_1 \perp m \wedge H_1\, m_1) \Rightarrow H_2(m_1 \uplus m)$$

$$p \rightsquigarrow \mathsf{MlistSeg}\, q\, \mathsf{nil} \equiv \ulcorner p = q \urcorner \qquad p \rightsquigarrow \mathsf{MlistSeg}\, q\, (x :: L') \equiv \exists p'.\ p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\} * p' \rightsquigarrow \mathsf{MlistSeg}\, q\, L'$$

$$p \rightsquigarrow \mathsf{Mlist}\, L \equiv p \rightsquigarrow \mathsf{MlistSeg}\, \mathsf{null}\, L \qquad l \mapsto \_ \equiv \exists v.\, l \mapsto v$$

Question 2.1. *For each of the following heap predicates, say how many unique heaps satisfy it, and give examples of such heaps when applicable. When there are several examples, provide a minimum of two.*

1. $1 \mapsto 1 * 2 \mapsto 2$
2. $1 \mapsto 1 * \mathsf{GC}$
3. $1 \mapsto 1 \mathbin{\wedge\!\!\!\wedge} (2 \mapsto 2 * \mathsf{GC})$
4. $(2 \mapsto 2 \mathbin{\vee\!\!\!\vee} 3 \mapsto 3) * (3 \mapsto 3 \mathbin{\vee\!\!\!\vee} 4 \mapsto 4)$
5. $1 \mapsto 1 \mathrel{-\!\!*} (2 \mapsto 2 * 1 \mapsto 1)$
6. $1 \mapsto 1 \mathrel{-\!\!*} 2 \mapsto 2$
7. $(1 \mapsto 1 \mathrel{-\!\!*} 2 \mapsto 2) * 1 \mapsto 1$
8. $p \rightsquigarrow \mathsf{MlistSeg}\ q\ [1]$
9. $p \rightsquigarrow \mathsf{MlistSeg}\ q\ [1;2] \mathbin{\wedge\!\!\!\wedge} r \rightsquigarrow \mathsf{MlistSeg}\ s\ [2;1]$

Question 2.2. *Derive, from the usual rule for assignment, the triple:*

$$\{(p \mapsto \_) * (p \mapsto v \mathrel{-\!\!*} P)\}\ p \mathrel{:=} v\ \{P\}$$

Question 2.3. *Show that entailment* (1) *below does not hold.*

$$(P * R) \mathbin{\wedge\!\!\!\wedge} (Q * R) \ \rhd\ (P \mathbin{\wedge\!\!\!\wedge} Q) * R \tag{1}$$

Definition 1. *A heap predicate P is* precise *if, for all heap m, there is at most one sub-heap $m' \subseteq m$ such that Pm'.*

For example, $l \mapsto v$ is precise for all l and v.

Question 2.4. *Is $l \mapsto \_$ precise? Is $\ulcorner \mathsf{True} \urcorner$ precise? Is $\ulcorner \mathsf{False} \urcorner$? Is $\exists l.\, l \mapsto v$? Is GC?*

Question 2.5. *Name a few other precise predicates, and then a few other non-precise predicates.*

Question 2.6. *Show that when P and Q are precise, then $P * Q$ is precise.*

Question 2.7. *Complete the affirmation: if $P \rhd Q$ and ... is precise, then ... is precise. Justify.*

Question 2.8. *Show that entailment* (1) *holds when R is precise.*

Question 2.9. *Show that for all p and L, the predicate $p \rightsquigarrow \mathsf{Mlist}\ L$ is precise.*

Question 2.10. *Show that for all p, the predicate $\exists L.\, p \rightsquigarrow \mathsf{Mlist}\ L$ is precise.*

3 Separation Logic: adjacency lists

Recall that list cells are records with mutable fields `hd` and `tl`:

```
type 'a cell = { mutable hd : 'a; mutable tl : 'a cell }
```

We desire a function `mconcat : 'a cell cell -> 'a cell` that returns a mutable list containing the concatenation of all the mutable lists contained in its argument. In other words, it should have the following specification:

$$\forall p L \ \{p \rightsquigarrow \mathsf{Mlistof\ Mlist}\ L\}\ \texttt{mconcat}\ p\ \{\lambda p'. p' \rightsquigarrow \mathsf{Mlist}(\mathsf{concat}\ L)\} \tag{2}$$

where $\mathsf{concat\ nil} \equiv \mathsf{nil}$ and $\mathsf{concat}\ (\mathsf{X} :: \mathsf{L}) \equiv \mathsf{X} +\!\!+ \mathsf{concat}\ L$, where X is a list and L is a list of lists, and $+\!\!+$ is the usual concatenation of two lists. To save up on memory, we want to make as few allocations as possible.

Question 3.1. *Give an implementation of* `mconcat` *that reuses the list cells of its argument so as to never allocate any new cell. Prove that your implementation satisfies specification* (2).

Consider graphs of the form $G = (V, E)$ with a set V of n nodes of the form $V = \{0, 1, \dots, n-1\}$ and a set of edges $E \subseteq V \times V$. We represent a graph by a record of its size n and an array of (non-necessarily sorted) mutable adjacency lists, i.e. $(i, j) \in E$ if j is present in the list at index i.

```
type graph = { size : int; adj : int cell array }
```

For example, the graph $G_1 = (\{0, 1, 2, 3, 4\}, \{(0, 1), (0, 3), (1, 3), (3, 1), (3, 2), (3, 3)\})$ can be represented as:

```
let l0 = { hd = 3; tl = { hd = 1; tl = null } }
let l1 = { hd = 3; tl = null }
let l3 = { hd = 2; tl = { hd = 3; tl = { hd = 1; tl = null } } }
let g1 = { size = 5; adj = [| l0; l1; null; l3; null |] }
```

Question 3.2. *Write a corresponding representation predicate* $g \rightsquigarrow \mathsf{Graph}\, G$.

Question 3.3. *Is it precise, in the sense of Definition 1?*

The relational composition of two sets of edges E_1 and E_2 on the same set of nodes V is defined as $E_1 \times E_2 \equiv \{(i, k) \in V^2 \mid (i, j) \in E_1 \wedge (j, k) \in E_2\}$. We would like to design a function of graph composition `graph_compose` such that:

$$\begin{array}{c} \forall V\ E_1\ E_2\ g_1\ g_2\ \{g_1 \rightsquigarrow \mathsf{Graph}(V, E_1) * g_2 \rightsquigarrow \mathsf{Graph}(V, E_2)\} \\ \texttt{graph\_compose}\ g_1\ g_2 \\ \{\lambda g, g_1 \rightsquigarrow \mathsf{Graph}(V, E_1) * g_2 \rightsquigarrow \mathsf{Graph}(V, E_2) * g \rightsquigarrow \mathsf{Graph}(V, E_1 \times E_2)\} \end{array} \tag{3}$$

A candidate function is:

```
let graph_compose g1 g2 =
  assert (g1.size = g2.size);
  { size = g1.size;
    adj = Array.map (fun p -> mconcat (mmap (fun j -> g2.adj.(j)) p)) g1.adj }
```

where `mmap : ('a -> 'b) -> 'a cell -> 'b cell` is a map function on mutable lists.

Question 3.4. *Give an implementation and a specification of* `mmap` *so that the* `graph_compose` *function behaves as expected (no proof required).*

Question 3.5. *Give a limitation that specification* (3) *is suffering from. Suggest two ways of dealing with this problem.*

Question 3.6. *Give a sketch of the proof that* `graph_compose` *satisfies its specification.*

Recall the rule for the *parallel composition* of two terms `e1` and `e2` (written `e1 ||| e2`), running in parallel on different threads:

$$\frac{\{P_1\}\ \texttt{e1}\ \{\lambda\_.Q_1\} \qquad \{P_2\}\ \texttt{e2}\ \{\lambda\_.Q_2\}}{\{P_1 * P_2\}\ \texttt{e1 ||| e2}\ \{\lambda\_.Q_1 * Q_2\}}$$

Consider now the following function, where `all_threads_busy ()` returns an unknown Boolean:

```
let rec par_iter (f : 'a -> unit) (p : 'a array) (i j : int) =
  if i >= j or all_threads_busy () then
    for k = i to j do f p.(k) done
  else
    let m = (i + j) / 2 in
    par_iter f p i m |||
    par_iter f p (m + 1) j
```

Question 3.7. *Specify the function* `par_iter`*, so that it can be used on the array of adjacency lists for graphs. Give a sketch of a proof of correctness (if you make an induction, at least provide its statement, but it is not necessary to write out details for all steps).*

We define the following function, which modifies the head values of a mutable list according to a function of type `'a -> 'b`, effectively transforming, in place, `p` from an `'a cell` to a `'b cell`. Note that during the execution, `p` is ill-typed if `'a` and `'b` are incompatible.

```
let rec mlist_replace (f : 'a -> 'b) (p : 'a cell) =
  if p <> null then begin
    p.hd <- f p.hd;
    mlist_replace f p.tl
  end
```

Question 3.8. *Give a specification of* `mlist_replace` *in terms of* $\mathsf{Mlistof}$.

Question 3.9. *Prove that* `mlist_replace` *satisfies its specification (give a good amount of details).*

We want to run a parallel graph algorithm that manipulates graphs but requires to have weights and integer markings on each edge. The function `make_edge` is provided, to help modify adjacency lists accordingly. Because we are under tight memory constraints, we add those in-place by using the function `mlist_replace`.

```
type edge = { target : int; weight : int; mutable mark : int }

let make_edge j = { target = j; weight = Random.int 2; mark = 0 }

let augment_graph g = Array.iter (mlist_replace make_edge) g.adj
```

Note that after a call to `augment_graph g`, the original pointer `g` points to an object no longer fitting the type `graph`. It instead represents an "augmented" graph $\hat{G} = (V, \hat{E})$ where is the set of weighted marked edges, and $\hat{E} \subseteq V^2 \times \mathbb{N}^2$.

Question 3.10. *Give a new representation predicate of an augmented graph* $g \rightsquigarrow \mathsf{AugmentedGraph}\ \hat{G}$.

Question 3.11. *Knowing a specification for* `Random.int` *can be:* $\forall n,\ \{\ulcorner n > 0 \urcorner\}$ `Random.int` n $\{\lambda i. \ulcorner 0 \leqslant i < n \urcorner\}$*, and give a specification for the function* `augment_graph`*, together with a proof sketch of correctness.*