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 9 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.*

Answer. To prove termination we need a variant, e.g.

```
variant { j - i }
```

No other annotations are needed for safety

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.*

Answer. To prove termination we need a variant, e.g.

```
variant { a.length - i }
```

To prove safety of the array access `a[i]` we need the invariant

```
invariant { 0 <= i }
```

which together with the loop condition, ensures that `0 <= i < a.length` for the array access.

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).*

Answer. It is natural to add the loop invariant

```
invariant { n = occ x a.elts 0 i }
```

To obtain the proof of the post-condition, we need to show that `i = a.length` at the loop exit. This can be achieved with an extra invariant `i <= a.length`. The initialization of the loop invariant is a trivial consequence of the definition of `occ`. The preservation of the invariant amounts to prove that

```
occ x a.elts 0 (i+1) = (if a[i]=x then 1 else 0) + occ x a.elts 0 i
```

which is not trivial since the definition of `occ` is recursive on the first index, not the second. Such a result can be obtained by a lemma provable by induction as follows

```
let rec lemma occ_from_right (x:int) (f:int → int) (i j:int) : unit
  requires { i <= j }
  variant { j - i }
  ensures { occ x f i (j+1) = (if f j = x then 1 else 0) + occ x f i j }
= if i < j then occ_from_right x f (i+1) j
```

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)*

Answer. We can view the array as a sequence of five pieces: the segment between 0 and i (excluded), the element $a[i]$ itself, the segment between $i + 1$ and j (excluded), the element $a[j]$ and the segment between $j + 1$ and `a.length`. For any element x, the number of its occurrences in the three segments is unchanged by the swap. This is indeed a frame property for the `occ` predicate, that could be stated as

```
let lemma occ_frame (f g:int → int) (i j:int) (x:int) : unit
  requires { ∀ k. i <= k < j → f k = g k }
  ensures { occ x f i j = occ x g i j }
```

Such a lemma suffices to prove that at the end of `swap`

```
∀ x. occ x a.elts 0 i = occ x (old a).elts 0 i /\
          occ x a.elts (i+1) j = occ x (old a).elts (i+1) j /\
          occ x a.elts (j+1) a.length = occ x (old a).elts (j+1) a.length
```

This is not enough, we need to prove that the number of occurrences in the array is indeed the sum of occurrences in each of the five parts. This can be expressed by a lemma on occurrences in a concatenation:

```
let lemma occ_append (x:int) (f:int → int) (i j k : int) : unit
  requires { i <= j <= k }
  ensures { occ x f i k = occ x f i j + occ x f j k }
```

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

Answer. The frame property can be proved by a recursive lemma function

```
let rec lemma occ_frame (f g:int → int) (i j:int) (x:int) : unit
  requires { ∀ k. i <= k < j → f k = g k }
  variant { j - i }
  ensures { occ x f i j = occ x g i j }
= if i < j then occ_frame f g (i+1) j x
```

The lemma on concatenation can be also proved by induction using

```
let rec lemma occ_append (x:int) (f:int → int) (i j k : int) : unit
  requires { i <= j <= k }
  variant { j - i }
  ensures { occ x f i k = occ x f i j + occ x f j k }
= if i < j then occ_append x f (i+1) j k
```

Note: the solutions above are syntactically accurate so as to be checked using Why3. Yet it is not expected for students to provide a so precise answer, the general idea of lemma functions proved by induction is enough.

---

2 Separation Logic: heap predicates

We recall the definition of a few separation logic connectives:

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

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

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

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

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

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. $p \leadsto \mathsf{MlistSeg}\ q\ [1]$
9. $p \leadsto \mathsf{MlistSeg}\ q\ [1;2] \mathbin{\wedge\!\!\!\wedge} r \leadsto \mathsf{MlistSeg}\ s\ [2;1]$

Answer.

1. one: $\{(1,1),(2,2)\}$
2. infinitely many, e.g., $\{(1,1)\}$ and $\{(1,1),(2,2)\}$
3. zero
4. three: $\{(2,2),(3,3)\}$, $\{(2,2),(4,4)\}$, and $\{(3,3),(4,4)\}$.
5. infinitely many: $\{(2,2)\}$ and any heap mapping 1, e.g., $\{(1,3),(4,4)\}$
6. infinitely many: any heap mapping 1, e.g., $\{(1,1)\}$, $\{(1,3),(4,4)\}$

7. zero
8. one: $\{(p,1),(p+1,q)\}$
9. one: $\{(p,1),(p+1,r),(r,2),(r+1,p)\}$ if $p=q$ and $r=s$, zero otherwise

---

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

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

Answer. Using the usual rule for assignment, `:=`:

$$\{p \mapsto \_\}\ p \texttt{ := } v\ \{p \mapsto v\}$$

then framing with $p \mapsto v \mathrel{-\!\!*} P$, we obtain:

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

we conclude using the consequence rule on the right, using the fact that $p \mapsto v * (p \mapsto v \mathrel{-\!\!*} P) \rhd P$.

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

$$(P * R) \curlywedge (Q * R)\ \rhd\ (P \curlywedge Q) * R \tag{1}$$

Answer. Let $P = 1 \mapsto 1$, $Q = 2 \mapsto 2$, and $R = P \curlyvee Q$. Both $P * R$ and $Q * R$ are equivalent to $P * Q$ and are satisfied by heap $h = \{(1,1),(2,2)\}$. However $P \curlywedge Q$ is always false, so heap h satisfies the left-hand side but not the right hand side.

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*?*

Answer. Yes. Yes and yes, $\ulcorner P \urcorner$ is always precise. $\exists l.\, l \mapsto v$ and GC are not.

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

Answer. $1 \mapsto 1 * 2 \mapsto 2$, $p \leadsto \mathsf{Array}\ L$, and $p \leadsto \mathsf{MTree}\ T$ are precise, as well as their (separating or not) conjunctions. $1 \mapsto 1 \curlyvee 2 \mapsto 2$ and $1 \mapsto 1 \mathrel{-\!\!*} Q$ are not precise, and neither is $\mathsf{GC} * P$ except if $P \rhd \mathsf{False}$.

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

Answer. Let h and $h_1, h_2 \subseteq h$ that both satisfy $P * Q$. Then there are p_i, q_i such that $p_i \uplus q_i = h_i$, $P\ p_i$, and $Q\ q_i$ for $i \in \{1,2\}$. Since $p_i \subseteq h_i$ and $h_i \subseteq h$, both p_1 and p_2 are subsets of h and satisfy P, so $p_1 = p_2$. Similarly $q_1 = q_2$, so $h_1 = h_2$ and $P * Q$ is precise.

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

Answer. If $P \rhd Q$ and Q is precise, then P is precise. Indeed suppose $h_1, h_2 \subseteq h$ such that Ph_1 and Ph_2, it is enough to show $h_1 = h_2$, and so it is also enough to show Qh_1 and Qh_2 since Q is precise, and both hold because $P \rhd Q$.

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

Answer. Suppose $((P * R) \curlywedge (Q * R))h$, then $(P * R)h$ and $(Q * R)h$, then $h = p \uplus r_1 = q \uplus r_2$ with Pp and Qq with r_1 and r_2 both satisfying R and sub-heaps of h, so $r_1 = r_2$. So, $p = h \backslash r_1 = h \backslash r_2 = q$, so Pq and so $(P \curlywedge Q)p$, and so $p \uplus r_1 = h$ satisfies $(P \curlywedge Q) * R$.

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

Answer. Follows from Questions 2.7 and 2.10.

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

Answer. Fix some heap h. For any p we write h_p for $\{(p, h(p)), (p+1, h(p+1))\}$. We define the relation $q \to r$ whenever $q \neq \mathsf{null}$, $h(q+1)$ is defined and $h(q+1) = r$. Let $h^*(p) = \bigcup\{h_q \mid p \to^* q\}$.

We show by induction on the size of h', that for all h', p and L, if $(p \rightsquigarrow \mathsf{Mlist}\ L)h'$ and $h' \subseteq h$, then $h' = h^*(p)$ (which means that h' is unique and finally that $\exists L.\, p \rightsquigarrow \mathsf{Mlist}\ L$ is precise).

If $p = \mathsf{null}$ then $L = \mathsf{nil}$ and $h' = \varnothing = h^*(\mathsf{null})$. Otherwise, $L = x :: L'$ for some x and L', and for some p' we have $h' = h_1 \uplus h_2$, such that:

1. $(p \rightsquigarrow \{\!|x; p'|\!\})(h_1)$. This means that $h_1 = \{(p, x), (p+1, p')\}$ and $h_1 \subseteq h' \subseteq h$, so $x = h(p)$, $p' = h(p+1)$, hence $h_1 = h_p$ and $p \to p'$.
2. $(p' \rightsquigarrow \mathsf{Mlist}\ L')(h_2)$. Since $h_2 \subseteq h' \subseteq h$ and $|h_2| = |h' \backslash h_1| < |h'|$ we know by induction $h_2 = h^*(p')$.

Finally, since $p \to p'$ and $\to$ is deterministic, $h^*(p) = h_p \cup h^*(p') = h_1 \cup h_2 = h'$.

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:

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

where $\mathsf{concat}\ \mathsf{nil} \equiv \mathsf{nil}$ 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).

Answer.

```
let rec plug p q = if p.tl = null then p.tl <- q else plug p.tl q
let mappend p q = if p = null then q else (plug p q; p)
let rec mconcat p = if p = null then null else mappend p.hd (mconcat p.tl)
```

For all q and L' we prove, by induction on L, that

$$\forall p,\ L \neq \mathsf{nil} \Rightarrow \{p \rightsquigarrow \mathsf{Mlistof}\ L * q \rightsquigarrow \mathsf{Mlistof}\ L'\}\ \texttt{plug}\ p\ q\ \{\lambda\_, p \rightsquigarrow \mathsf{Mlistof}\ (L +\!\!+ L')\}$$

In the case $L = x :: \mathsf{nil}$, the assignment changes $p \rightsquigarrow \{\!|x, \mathsf{null}|\!\}$ to $p \rightsquigarrow \{\!|x, \mathsf{q}|\!\}$, which joins with $q \rightsquigarrow \mathsf{Mlistof}\ L'$ to make $p \rightsquigarrow \mathsf{Mlistof}\ (x :: L')$. In the induction case, we frame $p \rightsquigarrow \{\!|x, p'|\!\}$ and conclude with the I.H. with p' and L'.

A simple case analysis on L is required for

$$\forall pqLL', \{p \rightsquigarrow \mathsf{Mlistof}\ L * q \rightsquigarrow \mathsf{Mlistof}\ L'\}\ \texttt{mappend}\ p\ q\ \{\lambda r, r \rightsquigarrow \mathsf{Mlistof}\ (L +\!\!+ L')\}$$

as when $L = \mathsf{nil}$, we return q, and otherwise we call `plug` and return p as r.

We show (2), for all p, by induction on the list of lists L. The case nil is immediate. The case $L = X :: L'$ unfolds to, for some x and p',

$$\{p \rightsquigarrow \{\!|x, p'|\!\} * x \rightsquigarrow \mathsf{Mlist}\ X * p' \rightsquigarrow \mathsf{Mlistof}\ \mathsf{Mlist}\ L'\}\ \texttt{mappend}\ x\ (\texttt{mconcat}\ p')\ \{\lambda r.r \rightsquigarrow \mathsf{Mlist}(X +\!\!+ \mathsf{concat}\ L')\}$$

by induction and framing everything except $p' \rightsquigarrow ...$, we get some r' such that we need to prove:

$$\{p \rightsquigarrow \{\!|x, p'|\!\} * x \rightsquigarrow \mathsf{Mlist}\ X * r' \rightsquigarrow \mathsf{Mlist}(\mathsf{concat}\ L')\}\ \texttt{mappend}\ x\ r'\ \{\lambda r.r \rightsquigarrow \mathsf{Mlist}(X +\!\!+ \mathsf{concat}\ L')\}$$

framing then garbage-collecting $p \rightsquigarrow ...$, this is an instance of the specification of `mappend`.

Consider graphs of the form $G = (V, E)$ with a set V of n nodes of the form $V = \{0, 1, \ldots, n-1\}$ 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 $g \rightsquigarrow \mathsf{Graph}\, G$.*

Answer. Define $\mathsf{ArrayOf}$ as

$$p \rightsquigarrow \mathsf{ArrayOf}\ R\ L \ \equiv \ \mathop{\circledast}_{i \in \mathsf{dom}\, L} \exists v.\, p + i \mapsto v * v \rightsquigarrow R\ L[i]$$

and let $\mathsf{repr}\ V\ E\ L$ be $\forall (i,j) \in V^2, (i,j) \in E \Leftrightarrow j \in L[i]$. Then, $g \rightsquigarrow \mathsf{Graph}\,(V,E)$ is defined as

$$\exists p : \mathsf{loc}, L : \mathsf{list(list(int))}.\ g \rightsquigarrow \{|\mathrm{size}{=}|V|, \mathrm{adj}{=}p|\} * p \rightsquigarrow \mathsf{ArrayOf}\ \mathsf{Mlist}\ L * \ulcorner \mathsf{repr}\ V\ E\ L \urcorner$$

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

Answer. Yes, it is precise (even if there are several representations of the same graph).

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:

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

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).*

Answer. `mmap` must not reuse list cells and its precondition must appear in the postcondition, and not e.g. reuse the structure of the list, otherwise it could modify the graph itself.

```
let rec mmap f p = if p = null then null else { hd = f p.hd; tl = mmap f p.tl }
```

Using the precondition where f is characterized by a logical function F, but still allowing to use some invariant I, i.e. $\forall x\{I\}\ f\ x\ \{\lambda x'.I * \ulcorner x' = F x \urcorner\}$ implies

$$\{I * p \rightsquigarrow \mathsf{Mlist}\ L\}\ \texttt{mmap}\ f\ p\ \{\lambda p', I * p \rightsquigarrow \mathsf{Mlist}\ L * p' \rightsquigarrow \mathsf{Mlist}(\mathsf{map}\ F\ L)\}$$

In fact, to really make `graph_compose` work without modifying `mconcat`, we would need to rather awkwardly add a list copy after each operation f, or alternatively to shadow the above definition with `let mmap f p = mmap mlist_copy (mmap f p)`. See the solution of Question 3.6 for more details.

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

Answer. It forbids the case $g_1 = g_2$. We can bypass the problem making a copy of the graph, or we can overcome it by allowing fractional permissions for Graph (recursively including defining fractional versions of $\mathsf{ArrayOf}$ and Mlist) so as to allow them in pre- and post-conditions of the specification.

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

Answer. Sadly `graph_compose` does not satisfy its specification since `mconcat` modifies the lists inside its argument, which are adjacency lists that do need to be re-established in the postcondition. One way to fix this problem is to add a mutable list copy function `copy` in the argument of `mmap`[1]. Another is to use a non-destructive version of `mconcat`. The natural specification for `mconcat` would be

$$\forall pL\ \{p \leadsto \mathsf{Mlistof\ Mlist}\ L\}\ \texttt{mconcat}\ p\ \{\lambda p'.p \leadsto \mathsf{Mlistof\ Mlist}\ L * p' \leadsto \mathsf{Mlist}(\mathsf{concat}\ L)\}$$

but that would not be enough, as this forbids sharing in the input structure[2]. We need instead

$$\begin{aligned}\forall pLKM,\ K \subseteq \mathsf{dom}(M)\ \Rightarrow\ &\{p \leadsto \mathsf{Mlist}\ K * \mathsf{Cellsof\ Mlist}\ M\}\ \texttt{mconcat}\ p \\ &\{\lambda p'.p \leadsto \mathsf{Mlist}\ K * \mathsf{Cellsof\ Mlist}\ M * p' \leadsto \mathsf{Mlist}(\mathsf{concat}\ (\mathsf{map}\ M\ K))\}\end{aligned} \tag{4}$$

Let V, E_1, E_2, g_1, g_2 be from the precondition. The V is shared by all graphs, so the condition of the `assert` evaluates to `true` and the record part about `size` holds trivially. We are left with proving, for all p_1, p_2, L_1, L_2 assuming that $\mathsf{repr}\ V\ E_1\ L_1$ and $\mathsf{repr}\ V\ E_2\ L_2$, and framing out parts about the `size/adj` record, that

$$\begin{gathered}\{p_1 \leadsto \mathsf{ArrayOf\ Mlist}\ L_1 * p_2 \leadsto \mathsf{ArrayOf\ Mlist}\ L_2\} \\ \texttt{Array.map (fun p} \to \texttt{mconcat (mmap (fun j} \to p_2\texttt{.(j)) p))}\ p_1 \\ \{\lambda r.p_1 \leadsto \mathsf{ArrayOf\ Mlist}\ L_1 * p_2 \leadsto \mathsf{ArrayOf\ Mlist}\ L_2 * r \leadsto \mathsf{ArrayOf\ Mlist}\ L' * \ulcorner \mathsf{repr}\ V\ (E_1 \times E_2)\ L' \urcorner\}\end{gathered} \tag{5}$$

for some L' that we choose to reflect the structure of the program, i.e.

$$L' \equiv \mathsf{map}\ F\ L_1 \quad \text{where } Fl \equiv \mathsf{concat}\ (\mathsf{map}\ F_2\ l) \text{ and } F_2 \equiv \lambda j.L_2[j]$$

We remark that $\mathsf{repr}\ V\ (E_1 \times E_2)\ L'$. Indeed, $\forall (i,k) \in V^2$,

$$\begin{aligned} k \in L'[i] &\Leftrightarrow k \in \mathsf{concat}\ (\mathsf{map}\ F_2\ L_1[i]) \\ &\Leftrightarrow \exists l, k \in l \wedge l \in \mathsf{map}\ F_2\ L_1[i] \\ &\Leftrightarrow \exists l, k \in l \wedge \exists j \in L_1[i] \wedge l = L_2[j] \\ &\Leftrightarrow \exists j, j \in L_1[i] \wedge k \in L_2[j] \\ &\Leftrightarrow \exists j, (i,j) \in E_1 \wedge (j,k) \in E_2 && \text{since } \mathsf{repr}\ V\ E_1\ L_1 \text{ and } \mathsf{repr}\ V\ E_2\ L_2 \\ &\Leftrightarrow (i,k) \in (E_1 \times E_2) && \text{by definition of } \times \end{aligned}$$

We now prove (5) by using the following suitable rule for `Array.map`

$$\begin{gathered}\forall fFRS\ \ (\forall xX\ X \in L \Rightarrow\ \{I * x \leadsto RX\}\ f\ x\ \{\lambda x'.I * x \leadsto RX * x' \leadsto S(FX)\} \Rightarrow \\ \forall Lp\ \{I * p \leadsto \mathsf{ArrayOf}\ R\ L\}\ \texttt{Array.map}\ f\ p\ \{\lambda r.I * p \leadsto \mathsf{ArrayOf}\ R\ L * r \leadsto \mathsf{ArrayOf}\ S\ (\mathsf{map}\ F\ L)\}\end{gathered}$$

by choosing $p = p_1$, $L = L_1$, $F = F$, $R = S = \mathsf{Mlist}$, and $I = p_2 \leadsto \mathsf{ArrayOf\ Mlist}\ L_2$. Note that I is used several times during the application of `Array.map`. We are left to prove the premise of the rule, for all p and l such that $l \in L_1$,

$$\{I * p \leadsto \mathsf{Mlist}\ l\}\ \texttt{mconcat}\ \texttt{(mmap}\ f_2\ p\texttt{)}\ \{\lambda x'.I * p \leadsto \mathsf{Mlist}\ l * x' \leadsto \mathsf{Mlist}(Fl)\} \tag{6}$$

trying to prove $f_2 \equiv$ `fun j` $\to p_2$`.(j)` by using F_2 as its specification, we encounter the access problem typical to higher-order representation predicates. So we unfold I to get a plain Array:

$$I = p_2 \leadsto \mathsf{ArrayOf\ Mlist}\ L_2 = \exists K_2.\, p_2 \leadsto \mathsf{Array}\ K_2 * \ulcorner |K_2| = |L_2| \urcorner \mathop{\circledast}_{i \in \mathsf{dom} L_2} K_2[i] \leadsto \mathsf{Mlist}\ L_2[i]$$

we extract the existentially quantified K_2 from the precondition (and instantiate it in the post), and we let M be the family $(K_2[i], L_2[i])_{i \in \mathsf{dom} L_2}$, so that we can replace I with $p_2 \leadsto \mathsf{Array}\ K_2 * \mathsf{Cellsof\ Mlist}\ M$ in (6). Now, we can have a simple specification for f_2:

$$\forall j \{p_2 \leadsto \mathsf{Array}\ K_2\}\ f_2\ j\ \{\lambda x.\ulcorner x = K_2[j] \urcorner * p_2 \leadsto \mathsf{Array}\ K_2\}$$

which results in the following triple when applying a rule for `mmap`:

$$\{p_2 \leadsto \mathsf{Array}\ K_2 * p \leadsto \mathsf{Mlist}\ l\}\ \texttt{mmap}\ f_2\ p\ \{\lambda q.p_2 \leadsto \mathsf{Array}\ K_2 * p \leadsto \mathsf{Mlist}\ l * q \leadsto \mathsf{Mlist}\ (\mathsf{map}\ K_2\ l)\}$$

---

[1] One could even modify `mmap` so that it copies its result as a mutable list, so that there is no need to change `mconcat`, but this is making `mmap` do more than its name suggests

[2] Lists with sharing happen when there are duplicates in an adjacency list. Such duplicates could be prevented by our definition of `graph`, but such a definition would be too restrictive since `graph_compose` itself can introduce duplicates.

by framing $\mathsf{Cellsof\ Mlist}\ M$ the precondition coincides with (6)'s and the postcondition is

$$\mathsf{Cellsof\ Mlist}\ M * p_2 \rightsquigarrow \mathsf{Array}\ K_2 * p \rightsquigarrow \mathsf{Mlist}\ l * q \rightsquigarrow \mathsf{Mlist}\ (\mathsf{map}\ K_2\ l)$$

it remains to apply `mconcat` to q, so by applying (4) and framing $p_2 \rightsquigarrow \mathsf{Array}\ K_2$ we get postcondition

$$\{\lambda x'.p_2 \rightsquigarrow \mathsf{Array}\ K_2 * q \rightsquigarrow \mathsf{Mlist}\ (\mathsf{map}\ K_2\ l) * \mathsf{Cellsof\ Mlist}\ M * x' \rightsquigarrow \mathsf{Mlist}(\mathsf{concat}\ (\mathsf{map}\ M\ (\mathsf{map}\ K_2\ l)))\}$$

which is (6)'s postcondition once we throw away $q \rightsquigarrow$ ... since

$$\begin{aligned}
& \mathsf{concat}\ (\mathsf{map}\ M\ (\mathsf{map}\ K_2\ l)) \\
= \ & \mathsf{concat}\ (\mathsf{map}\ (M \circ K_2)\ l) \\
= \ & \mathsf{concat}\ (\mathsf{map}\ (\lambda j.M(K_2[i]))\ l) \\
= \ & \mathsf{concat}\ (\mathsf{map}\ (\lambda j.L_2[i])\ l) \\
= \ & \mathsf{concat}\ (\mathsf{map}\ F_2\ l) \\
= \ & Fl
\end{aligned}$$

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).*

Answer. One possibility, writing $p \mapsto_{i,j} L$ for $\circledast_{k\in\{i,\ldots,j\}}\, p+k \mapsto L[k]$ and $R(i,j)$ for $\circledast_{k\in\{i,\ldots,j\}}\, R\ k$:

$$\begin{gathered}\forall f\ R\ S\ L\ i\ j, 0 \leqslant i \leqslant j < |L| \Rightarrow (\forall k \in \{i,\ldots,j\}, \{R\ k * \ulcorner x = L[k]\urcorner\}\ f\ x\ \{\lambda\_.S\ k\}) \Rightarrow \\ \forall p, \{p \mapsto_{i,j} L * R(i,j)\}\ \texttt{par\_iter}\ f\ p\ i\ j\ \{\lambda\_.p \mapsto_{i,j} L * S(i,j)\}\end{gathered} \tag{7}$$

Assuming the premise, we show by induction on $j-i$ that for all i and j with $0 \leqslant i \leqslant j < |L|$, (7)'s RHS holds. The condition is unknown so we prove the triple for both branches.

For the first branch, applying the rule for `for`, we choose the invariant

$$Ik = p \mapsto_{i,j} L * S(i,k-1) * R(k,j)$$

Since $S(i,i-1) = \ulcorner\urcorner = R(j+1,j)$, Ii does entail the precondition and $I(j+1)$ the postcondition. We need to prove the body, i.e. when $k \in \{i,\ldots,j\}$, $\{Ik\}\ f\ \texttt{p.}(k)\ \{\lambda\_I(k+1)\}$, which rewrites to:

$$\{p \mapsto_{i,j} L * S(i,k-1) * Rk * R(k+1,j)\}\ f\ \texttt{p.}(k)\ \{\lambda\_p \mapsto_{i,j} L * S(i,k-1) * Sk * R(k+1,j)\}$$

$\texttt{p.}(k)$ evaluates to $L[k]$, and so it is exactly the premise of (7) after applying the frame rule.

For the second branch, we apply the rule for parallel composition after the following rewritings

$$p \mapsto_{i,j} L = p \mapsto_{i,m} L * p \mapsto_{m+1,j} L \qquad R(i,j) = R(i,m) * R(m+1,j) \qquad S(i,j) = S(i,m) * S(m+1,j)$$

which concludes the proof.

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* Mlistof.

Answer. One possibility is to have two representation predicates, R for `'a` and S for `'b`:

$$\forall f\ R\ S\ \ (\forall x X K K',\ \{x \rightsquigarrow RX * JKK'\}\ fx\ \{\lambda x', \exists X'. x' \rightsquigarrow SX' * J(K\&X)(K'\&X')\}) \Rightarrow \quad (8)$$

$$\forall pL\ \{p \rightsquigarrow \mathsf{Mlistof}\ R\ L * J\ \mathsf{nil}\ \mathsf{nil}\}\ \texttt{mlist\_replace}\ f\ p\ \{\lambda\_.\exists L'. p \rightsquigarrow \mathsf{Mlistof}\ S\ L' * J\ L\ L'\} \quad (9)$$

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

Answer. We assume (8) and prove the following generalisation of (9) by induction on L.

$\forall p\, K\, K'\ \{p \rightsquigarrow \mathsf{Mlistof}\, R\, L * J\, K\, K'\}\ \texttt{mlist\_replace}\ f\, p\, \{\lambda\_.\exists L'. p \rightsquigarrow \mathsf{Mlistof}\, S\, L' * J\ (K +\!\!+ L)\ (K' +\!\!+ L')\}$

If $L = \mathsf{nil}$ then nothing changes. Otherwise, $L = X :: M$. Writing $\overline{R}$ instead of $\mathsf{Mlistof}\ R$,

$\{p \rightsquigarrow \overline{R}\ (X :: M) * J\ K\ K'\}$	expands to, for some p', x,
$\{p \rightsquigarrow \{\!\|x, p'\|\!\} * x \rightsquigarrow RX * p' \rightsquigarrow \overline{R}\ M * J\ K\ K'\}$	then `p.hd` evaluates to x
	and fx returns x' s.t. for some X'
$\{p \rightsquigarrow \{\!\|x, p'\|\!\} * x' \rightsquigarrow SX' * p' \rightsquigarrow \overline{R}\ M * J\ (K\&X)\ (K'\&X')\}$	the assignment of x' gives
$\{p \rightsquigarrow \{\!\|x', p'\|\!\} * x' \rightsquigarrow SX' * p' \rightsquigarrow \overline{R}\ M * J\ (K\&X)\ (K'\&X')\}$	by IH+frame, we get for some M'
$\{p \rightsquigarrow \{\!\|x', p'\|\!\} * x' \rightsquigarrow SX' * p' \rightsquigarrow \overline{S}\ M' * J\ ((K\&X) +\!\!+ M)\ ((K'\&X') +\!\!+ M')\}$	
$\{p' \rightsquigarrow \overline{S}\ (X' :: M') * J\ (K +\!\!+ (X :: M))\ (K' +\!\!+ (X' :: M'))\}$	by folding and rewriting

which concludes the induction.

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* $g \rightsquigarrow \mathsf{AugmentedGraph}\ \hat{G}$.

Answer. Let $p \rightsquigarrow \mathsf{Edge}(j, w, m) \equiv \{\!\|\text{target}{=}j, \text{weight}{=}w, \text{mark}{=}m\|\!\}$, then $g \rightsquigarrow \mathsf{AugmentedGraph}\ (V, \hat{E})$ is

$$\exists L.\ p \rightsquigarrow \mathsf{ArrayOf}\ (\mathsf{MlistOf}\ \mathsf{Edge})\ L * \ulcorner \forall ijwm,\ (i, j, w, m) \in \hat{E} \Leftrightarrow (j, w, m) \in L[i] \urcorner$$

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.*

Answer. It is important to note, here, that $\hat{E}$ needs to be existentially quantified.

$$\{g \rightsquigarrow \mathsf{Graph}\ (V, E)\}\ \texttt{augment\_graph}\ g\ \{\lambda\_.\exists \hat{E}. g \rightsquigarrow \mathsf{AugmentedGraph}\ (V, \hat{E})$$
$$* \ulcorner \forall (i, j, w, m) \in V^2, (i, j, w, m) \in \hat{E} \Leftrightarrow (i, j) \in E \wedge w \in \{0, 1\} \wedge m = 0 \urcorner\}$$
