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 \( \text{occ} \) which, given an element \( x \), a function \( f \) and two integers \( i \) and \( j \), denotes the number of occurrences of \( x \) among \( f_1, f_{i+1}, \ldots, f_{j-1} \). Notice than the first index \( i \) is included in the count but not the last one \( j \). It is defined as follows

\[
\text{let rec ghost function occ } (x:\text{int}) (f:\text{int} \rightarrow \text{int}) (i j:\text{int}) : \text{int} = \text{if } j \leq i \text{ then } 0 \text{ else } (\text{if } f_i = x \text{ then } 1 \text{ else } 0) + \text{occ } x f (i+1) j
\]

**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.

\[
\text{variant } \{ \text{ } j - i \} \quad \text{No other annotations are needed for safety.}
\]

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

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

**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.

\[
\text{variant } \{ \text{ } a.\text{length} - i \} \quad \text{To prove safety of the array access } a[i] \text{ we need the invariant}
\]

\[
\text{invariant } \{ 0 \leq i \}
\]

which together with the loop condition, ensures that \( 0 \leq i < a.\text{length} \) for the array access.
**Question 1.3.** We now express the expected behavior of the program `count_occ` by adding the post-condition $\text{result} = \text{occ x a.elts 0 a.length}$. 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

\[
\text{invariant \{ n = \text{occ x a.elts 0 i} \}}
\]

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 $i \leq a.length$. The initialization of the loop invariant is a trivial consequence of the definition of `occ`. The preservation of the invariant amounts to prove that

\[
\text{occ x a.elts 0 (i+1)} = (\text{if a[i]=x then 1 else 0}) + \text{occ x a.elts 0 i}
\]

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

\[
\text{let rec lemma occ_from_right (x:int) (f:int -> int) (i j:int) : unit}
\]

\[
\text{requires \{ i <= j \}}
\]

\[
\text{variant \{ j - i \}}
\]

\[
\text{ensures \{ \text{occ x f i (j+1)} = (\text{if f j = x then 1 else 0}) + \text{occ x f i j} \}}
\]

\[
= \text{if i < j then occ_from_right x f (i+1) j}
\]

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

\[
\text{let swap (a:array int) (i j :int) : unit}
\]

\[
\text{requires \{ 0 <= i < j < a.length \}}
\]

\[
\text{writes \{ a \}}
\]

\[
\text{ensures \{ \forall x:int. \text{occ x (old a).elts 0 a.length} = \text{occ x a.elts 0 a.length} \}}
\]

\[
= \text{let tmp = a[i] in a[i] <- a[j]; a[j] <- tmp}
\]

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

\[
\text{let lemma occ_frame (f g:int -> int) (i j:int) (x:int) : unit}
\]

\[
\text{requires \{ \forall k. i <= k < j -> f k = g k \}}
\]

\[
\text{ensures \{ \text{occ x f i j} = \text{occ x g i j} \}}
\]

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

\[
\forall x. \text{occ x a.elts 0 i = occ x (old a).elts 0 i} /\n\text{occ x a.elts (i+1) j = occ x (old a).elts (i+1) j} /\n\text{occ x a.elts (j+1) a.length} = \text{occ x (old a).elts (j+1) a.length}
\]

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:

\[
\text{let lemma occ_append (x:int) (f:int -> int) (i j k : int) : unit}
\]

\[
\text{requires \{ i <= j <= k \}}
\]

\[
\text{ensures \{ \text{occ x f i k} = \text{occ x f i j} + \text{occ x f j k} \}}
\]

**Question 1.5.** Explain how the lemmas identified above can be proved correct. (20 lines max)
The frame property can be proved by a recursive lemma function

\[
\text{let rec lemma occ_frame (f g:int \to int) (i j:int) (x:int) : unit}
\]
requires \{ \forall k. i \leq k < j \to f k = g k \}
variant \{ j - i \}
ensures \{ \text{occ } x \ f \ i \ j = \text{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

\[
\text{let rec lemma occ_append (x:int) (f:int \to int) (i j k : int) : unit}
\]
requires \{ i \leq j \leq k \}
variant \{ j - i \}
ensures \{ \text{occ } x \ f \ i \ k = \text{occ } x \ f \ i \ j + \text{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:

\[
H_1 \land H_2 \equiv \lambda m. H_1 m \land H_2 m \quad l \mapsto v \equiv \lambda m. m = \{(l,v)\} \land l \neq \text{null} \quad 'P' \equiv \lambda m. m = \emptyset \land P
\]

\[
H_1 \lor H_2 \equiv \lambda m. H_1 m \lor H_2 m \quad H_1 \land H_2 \equiv \lambda m. \exists m_1 m_2. m = m_1 \lor m_2 \land H_1 m_1 \land H_2 m_2
\]

\[
\exists x. H \equiv \lambda m. \exists x. H m \quad \text{GC} \equiv \lambda m. \text{True} \quad H_1 \lor H_2 \equiv \lambda m. \forall m_1 (m_1 \downarrow m \land H_1 m_1) \Rightarrow H_2(m_1 \uparrow m)
\]

\[
p \rightsquigarrow \text{MlistSeg } q \text{ nil } = 'p = q' \quad p \rightsquigarrow \text{MlistSeg } q (x :: L') \equiv \exists p'. p \mapsto [\text{hd}=x; \ \text{tl}=p'] \land p' \rightsquigarrow \text{MlistSeg } q L'
\]

\[
p \rightsquigarrow \text{Mlist } L \equiv p \rightsquigarrow \text{MlistSeg } \text{null } L \quad l \mapsto _{=} \exists v. l \mapsto v
\]

**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. \(1 \mapsto 1 \mapsto 2 \mapsto 2\)
2. \(1 \mapsto 1 \mapsto \text{GC}\)
3. \(1 \mapsto 1 \mapsto (2 \mapsto 2 \mapsto \text{GC})\)
4. \((2 \mapsto 2 \mapsto 3 \mapsto 3) \mapsto (3 \mapsto 3 \mapsto 4 \mapsto 4)\)
5. \(1 \mapsto 1 \mapsto (2 \mapsto 2 \mapsto 1 \mapsto 1)\)
6. \(1 \mapsto 1 \mapsto 2 \mapsto 2\)
7. \((1 \mapsto 1 \mapsto 2 \mapsto 2) \mapsto 1 \mapsto 1\)
8. \(p \rightsquigarrow \text{MlistSeg } q [1]\)
9. \(p \rightsquigarrow \text{MlistSeg } q [1;2] \mapsto r \rightsquigarrow \text{MlistSeg } s [2;1]\)

**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 \(p = q \text{ and } r = s\), zero otherwise

---

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

\[
\{(p \mapsto \_ ) \ast (p \mapsto v \mapsto P)\} \ p := v \ \{P\}
\]

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

\[
\{p \mapsto \_ \} \ p := v \ \{p \mapsto v\}
\]

then framing with \(p \mapsto v \mapsto P\), we obtain:

\[
\{p \mapsto \_ \ast (p \mapsto v \mapsto P)\} \ p := v \ \{p \mapsto v \ast (p \mapsto v \mapsto P)\}
\]

we conclude using the consequence rule on the right, using the fact that \(p \mapsto v \ast (p \mapsto v \mapsto P) \not\models P\).

---

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

\[
(P \ast R) \models (Q \ast R) \not\models (P \models Q) \ast R \quad (1)
\]

**Answer.** Let \(P = 1 \mapsto 1\), \(Q = 2 \mapsto 2\), and \(R = P \not\models Q\). Both \(P \ast R\) and \(Q \ast R\) are equivalent to \(P \ast Q\) and are satisfied by heap \(h = \{(1,1), (2,2)\}\). However \(P \models Q\) 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 \(P \models m'\).

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

**Question 2.4.** Is \(l \mapsto \_\) precise? Is 'True' precise? Is 'False'? Is \(\exists l. l \mapsto v\) Is GC?

**Answer.** Yes. Yes and yes, 'P' 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.** \(1 \mapsto 1 \mapsto 2 \mapsto 2, p \mapsto \text{Array } L, \) and \(p \mapsto M\text{Tree } T\) are precise, as well as their (separating or not) conjunctions. \(1 \mapsto 1 \not\models 2 \mapsto 2\) and \(1 \mapsto 1 \models Q\) are not precise, and neither is GC \(\ast P\) except if \(P \not\models \text{False}\).

---

**Question 2.6.** Show that when \(P\) and \(Q\) are precise, then \(P \ast Q\) is precise.

**Answer.** Let \(h\) and \(h_1, h_2 \subseteq h\) that both satisfy \(P \ast Q\). 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 \(P \ast Q\) is precise.

---

**Question 2.7.** Complete the affirmation: if \(P \not\models Q\) and ... is precise, then ... is precise. Justify.

**Answer.** If \(P \not\models Q\) and \(Q\) is precise, then \(P\) is precise. Indeed suppose \(h_1, h_2 \subseteq h\) such that \(P h_1\) and \(P h_2\), it is enough to show \(h_1 = h_2,\) and so it is also enough to show \(Q h_1\) and \(Q h_2\) since \(Q\) is precise, and both hold because \(P \not\models Q\).

---

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

**Answer.** Suppose \((P \ast R) \models (Q \ast R)\ h,\) then \((P \ast R) h\) and \((Q \ast R) h,\) then \(h = p \uplus r_1 = q \uplus r_2\) with \(P p\) and \(Q q\) with \(r_1\) and \(r_2\) both satisfying \(R\) and sub-heaps of \(h,\) so \(r_1 = r_2.\) So, \(p = h \setminus r_1 = h \setminus r_2 = q,\) so \(P q\) and so \((P \models Q) p,\) and so \(p \uplus r_1 = h\) satisfies \((P \models Q) \ast R.\)

---

**Question 2.9.** Show that for all \(p\) and \(L,\) the predicate \(p \mapsto \text{Mlist } L\) is precise.
Answer. Follows from Questions 2.7 and 2.10.

Question 2.10. Show that for all \( p \), the predicate \( \exists L. p \leadsto \text{Mist} L \) 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 \( q \leadsto r \) whenever \( q \neq \text{null} \), \( h(q + 1) \) is defined and \( h(q + 1) = r \). Let \( h^*(p) = \bigcup\{h_q \mid p \leadsto^* q\} \).

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

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

1. \( (p \leadsto \{x,p\}\}(h_1)) \). 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 \( h_1 = h_2 \) and \( p \leadsto p' \).

2. \( (p' \leadsto \text{Mist} L')(h_2) \). Since \( h_2 \subseteq h' \subseteq h \) and \( |h_2| = |h' \setminus h_1| < |h'| \) we know by induction \( h_2 = h^*(p') \).

Finally, since \( p \leadsto p' \) and \( \leadsto \) 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 \( \text{hd} \) and \( \text{tl} \):

\[
\text{type 'a cell = \{ mutable hd : 'a; mutable tl : 'a cell \}}
\]

We desire a function \( \text{mconcat} : \text{'a cell cell} \rightarrow \text{'a cell} \) 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:

\[
\forall pL. (p \leadsto \text{Mist list} L) \rightarrow \text{mconcat} p \leadsto (\lambda p'p' \leadsto \text{Mist} (\text{concat} L)) \tag{2}
\]

where \( \text{concat} \text{nil} = \text{nil} \) and \( \text{concat} (X :: L) = X ++ \text{concat} L \), 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 \( \text{mconcat} \) that reuses the list cells of its argument so as to never allocate any new cell. Prove that your implementation satisfies specification (2).

Answer.

\[
\begin{align*}
\text{let rec } & \text{plug} p q = \text{if } p.\text{tl} = \text{null} \text{ then } p.\text{tl} \leftarrow q \text{ else } \text{plug} p.\text{tl} q \\
\text{let mappend} & p q = \text{if } p = \text{null} \text{ then } q \text{ else } (\text{plug} p q; p) \\
\text{let rec } & \text{mconcat} p = \text{if } p = \text{null} \text{ then } \text{null} \text{ else } \text{mappend} p.\text{hd} (\text{mconcat} p.\text{tl})
\end{align*}
\]

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

\[
\forall p, L \neq \text{null} \Rightarrow (p \leadsto \text{Mist list} L * q \leadsto \text{Mist list} L') \rightarrow \text{plug} p q (\lambda p'p' \leadsto \text{Mist list} (L + L'))
\]

In the case \( L = x :: \text{nil} \), the assignment changes \( p \leadsto [x, \text{null}] \) to \( p \leadsto [x, q] \), which joins with \( q \leadsto \text{Mist list} L' \) to make \( p \leadsto \text{Mist list} (x :: L') \). In the induction case, we frame \( p \leadsto [x, p'] \) and conclude with the I.H. with \( p' \) and \( L' \).

A simple case analysis on \( L \) is required for

\[
\forall pqLL', (p \leadsto \text{Mist list} L * q \leadsto \text{Mist list} L') \rightarrow \text{mappend} p q (\lambda r, r \leadsto \text{Mist list} (L + L'))
\]

as when \( L = \text{nil} \), we return \( q \), and otherwise we call \( \text{plug} \) and return \( p \) as \( r \).

We show (2), for all \( p \), by induction on the list of lists \( L \). The case \( \text{null} \) is immediate. The case \( L = x :: L' \) unfolds to, for some \( x \) and \( p' \),

\[
(p \leadsto [x,p'] \rightarrow x \leadsto \text{Mist list} L * p' \leadsto \text{Mist list} L') \rightarrow \text{mappend} x (\text{mconcat} p') (\lambda r, r \leadsto \text{Mist list}(x ++ \text{concat} L'))
\]

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

\[
(p \leadsto [x,p'] \rightarrow x \leadsto \text{Mist list} L * r' \leadsto \text{Mist list}(\text{concat} L')) \rightarrow \text{mappend} x r' (\lambda r, r \leadsto \text{Mist list}(x ++ \text{concat} L'))
\]

framing then garbage-collecting \( p \leadsto ... \), this is an instance of the specification of \( \text{mappend} \).

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:

```ocaml
define g1 = { size = 5; adj = [l1; l2; l3; l4; null]
let l1 = { hd = 1; tl = null }
let l2 = { hd = 2; tl = null }
let l3 = { hd = 3; tl = { hd = 1; tl = null } }
let l4 = { hd = 3; tl = { hd = 1; tl = null } }
```

**Question 3.2.** Write a corresponding representation predicate \( g \leadsto Graph\( G \).

**Answer.** Define ArrayOf as

\[
p \leadsto ArrayOf R L \equiv \forall i : dom L. \exists v. p + i \mapsto v \mapsto R \cdot L[i]
\]

and let \( repr \) \( V \cdot E \cdot L \) be \( \forall (i, j) \in V^2, (i, j) \in E \iff j \in L[i] \). Then, \( g \leadsto Graph(V, E) \) is defined as

\[
\exists p : loc, L : list(int). g \leadsto [size=| V |, adj=p] \cdot p \leadsto ArrayOf Mlist L \cdot 'repr V \cdot E \cdot L'\]  

**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 
\( E_1 \times E_2 \equiv \{(i, k) \in V^2 \mid (i, j) \in E_1 \land (j, k) \in E_2\} \). We would like to design a function of graph composition \( graph\_compose \) such that:

\[
\forall V \cdot E_1 \cdot E_2 \cdot g_1 \cdot g_2 \quad \{ g_1 \leadsto Graph(V, E_1) \cdot g_2 \leadsto Graph(V, E_2) \}
\]

\[
\{ \lambda g, g_1 \leadsto Graph(V, E_1) \cdot g_2 \leadsto Graph(V, E_2) \cdot g \leadsto Graph(V, E_1 \times E_2) \}
\]

A candidate function is:

```ocaml
define 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 \( mmap \) : \( 'a \mapsto 'b \mapsto 'a \ cell \mapsto 'b \ cell \) 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.

```ocaml
define 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. \( \forall x \in I \cdot \lambda x'. I \mapsto \lambda x' : F x' \) implies

\[
\{ I \cdot p \leadsto Mlist L \} \cdot mmap f p \cdot \lambda p', I \cdot p \leadsto Mlist L \cdot p' \leadsto Mlist(map F L) \}
\]

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 \( ArrayOf \) and \( Mlist \)) so as to allow them in pre- and post-conditions of the specification.
We now prove (5) by using the following suitable rule for
Array.map
let
\(\mathcal{K}\) typical to higher-order representation predicates. So we unfold
Array.map
in (6). Now, we can have a simple specification for
\(\text{mmap}\)
for some
\(\pi\), which results in the following triple when applying a rule for
\(\text{mconcat}\).
\[
\forall p L \{ p \rightarrow \text{Mistof Mlist } L \} \ \text{mconcat} \ p (\lambda p'. p \rightarrow \text{Mistof Mlist } L * p' \rightarrow \text{Mist(concat } L))
\]

but that would not be enough, as this forbids sharing in the input structure\(^2\). We need instead
\[
\forall pLK M, \ K \subseteq \text{dom}(M) \Rightarrow \{ p \rightarrow \text{Mist } K * \text{Cellsof Mlist } M \} \ \text{mconcat} \ p
\]
\[
\{ \lambda p'. p \rightarrow \text{Mist } K * \text{Cellsof Mlist } M * p' \rightarrow \text{Mist(concat (map } M K))\}\]

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
\text{assert}
evaluates to \text{true} and the record part about \text{size} holds trivially. We are left with proving, for all
\(p_1, p_2, L_1, L_2\) assuming that \(\text{repr } V E_1 L_1\) and \(\text{repr } V E_2 L_2\), and framing out parts about the \text{size/adj}
record, that
\[
\{p_1 \rightarrow \text{ArrayOf Mlist } L_1 * p_2 \rightarrow \text{ArrayOf Mlist } L_2\}
\]
\[
\text{Array.map} \ (\text{fun } p \rightarrow \text{mconcat (map (fun } j \rightarrow p_2(j)) \ p)) \ p_1
\]
\[
\{\lambda r.p_1 \rightarrow \text{ArrayOf Mlist } L_1 * p_2 \rightarrow \text{ArrayOf Mlist } L_2 * r \rightarrow \text{ArrayOf Mlist } L' * '\text{repr } V (E_1 \times E_2) L''\}
\]

for some \(L'\) that we choose to reflect the structure of the program, i.e.
\[
L' \equiv \text{map } F L_1 \text{ where } Fl \equiv \text{concat (map } F_2 l) \text{ and } F_2 \equiv \lambda j. L_2[j]
\]

We remark that \(\text{repr } V (E_1 \times E_2) L'\). Indeed, \(\forall (i, k) \in V^2,\)
\[
k \in L'[i] \Leftrightarrow k \in \text{concat (map } F_2 L_1[i])
\]
\[
\Rightarrow \exists l, k \in l \wedge l \in \text{map } F_2 L_1[i]
\]
\[
\Rightarrow \exists l, k \in l \wedge \exists j \in L_1[i] \wedge l = L_2[j]
\]
\[
\Rightarrow \exists j, j \in L_1[i] \wedge k \in L_2[j]
\]
\[
\Rightarrow \exists j, (i, j) \in E_1 \wedge (j, k) \in E_2 \text{ since } \text{repr } V E_1 L_1 \text{ and } \text{repr } V E_2 L_2
\]
\[
\Rightarrow (i, k) \in (E_1 \times E_2) \text{ by definition of } \times
\]

We now prove (5) by using the following suitable rule for \text{Array.map}
\[
\forall f FR S \ (\forall x X X \in L \Rightarrow \{ i * x \rightarrow RX \} \ f x \{ \lambda x'. i * x \rightarrow RX * x' \rightarrow S(FX)\} \Rightarrow
\forall Lp \{ l * p \rightarrow \text{ArrayOf R L} \} \ \text{Array.map} \ f p \{ \lambda i. i * p \rightarrow \text{ArrayOf R L} * r \rightarrow \text{Arrayof S (map } F L)\}
\]

by choosing \(p = p_1, L = L_1, F = F, R = S = \text{Mist}, \text{ and } I = p_2 \rightarrow \text{Arrayof Mist } L_2\). Note that \(I\) is used
several times during the application of \text{Array.map}. We are left to prove the premise of the rule, for all \(p\)
and \(I\) such that \(l \in L_1,\)
\[
\{ I * p \rightarrow \text{Mist } I \} \ \text{mconcat} \ (\text{map } f_2 p) \ {\lambda r.p_1 \rightarrow \text{Mist } I * p \rightarrow \text{Mist } F(Fl)}\}
\]

trying to prove \(f_2 = \text{fun } j \rightarrow p_2(j)\) 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 \(\text{Array}\):
\[
I = p_2 \rightarrow \text{Arrayof Mist } L_2 = \exists K_2, p_2 \rightarrow \text{Array } K_2 * 'K_2' = [L_2] * K_2[i] \rightarrow \text{Mist } L_2[i]
\]

we extract the existentially quantified \(K_2\) from the precondition (and instantiate it in the post), and we
let \(M\) be the family \((K_2[i], L_2[i])_{i \in \text{dom}(L_2)},\) so that we can replace \(I\) with \(p_2 \rightarrow \text{Array } K_2 * \text{Cellsof Mist } M\)
in (6). Now, we can have a simple specification for \(f_2:\)
\[
\forall j(p_2 \rightarrow \text{Array } K_2) \ f_2 j \ {\lambda r.x = K_2[j]}' * p_2 \rightarrow \text{Array } K_2\]
\]

which results in the following triple when applying a rule for \(\text{mmap}\):
\[
\{ p_2 \rightarrow \text{Array } K_2 * p \rightarrow \text{Mist } l \} \ \text{mmap} \ f_2 p \ {\lambda q.p_2 \rightarrow \text{Array } K_2 * p \rightarrow \text{Mist } l * q \rightarrow \text{Mist (map } K_2 l)\}
\]

\(^1\)One could even modify \(\text{mmap}\) so that it copies its result as a mutable list, so that there is no need to change \(\text{mconcat}\),
but this is making \(\text{mmap}\) 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 \text{graph}, but such a definition would be too restrictive since \text{graph-compose} itself can introduce duplicates.
by framing \texttt{CellsOf Mlist} \( M \) the precondition coincides with (6)'s and the postcondition is
\[
\text{CellsOf Mlist} \ M \ast p_2 \rightsquigarrow \text{Array} \ K_2 \ast p \rightsquigarrow \text{Mlist} \ l \ast q \rightsquigarrow \text{Mlist} \ (\map K_2 l)
\]
it remains to apply \texttt{mconcat} to \( q \), so by applying (4) and framing \( p_2 \rightsquigarrow \text{Array} \ K_2 \) we get postcondition
\[
\{ \lambda x'.p_2 \rightsquigarrow \text{Array} \ K_2 \ast q \rightsquigarrow \text{Mlist} \ (\map K_2 l) \ast \text{CellsOf Mlist} \ M \ast x' \rightsquigarrow \text{Mlist}(\text{concat} \ (\map M \ (\map K_2 l)))\}
\]
which is (6)'s postcondition once we throw away \( q \rightsquigarrow \ldots \) since
\[
\begin{align*}
\text{concat} \ (\map M \ (\map K_2 l)) & = \text{concat} \ (\map (M \circ K_2) l) \\
& = \text{concat} \ (\map (\lambda j.M(K_2[i]) \ l) \\
& = \text{concat} \ (\map (\lambda j.L_2[i]) l) \\
& = \text{concat} \ (\map F_2 l) \\
& = F l
\end{align*}
\]

Recall the rule for the \textit{parallel composition} of two terms \( e_1 \) and \( e_2 \) (written \( e_1 \parallel e_2 \)), running in parallel on different threads:
\[
\{P_1\} \ e_1 \{\lambda x.Q_1\} \quad \{P_2\} \ e_2 \{\lambda x.Q_2\}
\]
\[
\{P_1 \ast P_2\} \ e_1 \parallel e_2 \{\lambda x.Q_1 \ast Q_2\}
\]

Consider now the following function, where \texttt{all_threads_busy} () returns an unknown Boolean:
\[
\begin{align*}
\text{let rec par_iter } (f : 'a \rightarrow \text{unit}) \ (p : 'a \text{ array}) \ (i \ j : \text{int}) = \\
& \text{ if } i >= j \text{ or all_threads_busy } () \text{ then} \\
& \text{ for } k = i \rightarrow j \text{ do } f \ p.\langle k \rangle \ \text{done} \\
& \text{ else} \\
& \text{ let } m = (i + j) / 2 \text{ in} \\
& \text{ par_iter } f \ p \ i \ m \ ||| \\
& \text{ par_iter } f \ p \ (m + 1) \ j
\end{align*}
\]

\textbf{Question 3.7.} Specify the function \texttt{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).

\textbf{Answer.} One possibility, writing \( p \mapsto_{i,j} L \) for \( \ast_{k \in \{i, \ldots, j\}} p + k \mapsto L[k] \) and \( R(i,j) \) for \( \ast_{k \in \{i, \ldots, j\}} R k \):
\[
\forall f \ R S \ L \ i \ j, 0 \leq i < j < \lvert L \rvert \Rightarrow (\forall k \in \{i, \ldots, j\}, \{R k \mapsto x = L[k]\} f x \{\lambda x. S k\}) \Rightarrow \\
\forall p, \langle p \mapsto_{i,j} L \ast R(i,j) \rangle \text{ par_iter } f \ p \ i \ j \langle \lambda x. p \mapsto_{i,j} L \ast S(i,j) \rangle
\]

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

For the first branch, applying the rule for \texttt{for}, we choose the invariant
\[
I k = p \mapsto_{i,j} L \ast S(i, k - 1) \ast R(k, j)
\]
Since \( S(i, i - 1) = ' ' = R(j + 1, j) \), \( I i \) does entail the precondition and \( I(j + 1) \) the postcondition. We need to prove the body, i.e. when \( k \in \{i, \ldots, j\} \), \( \{I k\} \ f \ p.\langle k \rangle \{\lambda x. I(k + 1)\} \), which rewrites to:
\[
\{p \mapsto_{i,j} L \ast S(i, k - 1) \ast R k \ast R(k + 1, j)\} \ f \ p.\langle k \rangle \{\lambda x. p \mapsto_{i,j} L \ast S(i, k - 1) \ast S k \ast R(k + 1, j)\}
\]
\( p.\langle k \rangle \) 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 rewrites
\[
p \mapsto_{i,j} L = p \mapsto_{i,m} L \ast p \mapsto_{m+1,j} L \quad R(i,j) = R(i,m) \ast R(m + 1, j) \quad S(i,j) = S(i,m) \ast S(m + 1, j)
\]
which concludes the proof.

We define the following function, which modifies the head values of a mutable list according to a function of type \('a \rightarrow 'b\), effectively transforming, in place, \( p \) from an \('a \ \text{cell}\) to a \('b \ \text{cell}\). 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 \( 'b \):

\[
\forall f \, R \, S \quad (\forall x : K \times K', \{ x \mapsto RX \times JKK' \}) \quad \rightarrow \quad \forall p \, L \quad \{ p \mapsto \text{Mlistof} \, R \, L \mapsto J \, nil \, nil \, \text{mlist_replace} \, f \, p \, \{ \lambda_\exists \, \exists L' \, \rightarrow \, \text{Mlistof} \, S \, L' \mapsto J \, L \, L' \} \}
\]

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

\[
\forall p \, K' \quad \{ p \mapsto \text{Mlistof} \, R \, L \mapsto J \, K \, K' \} \quad \rightarrow \quad \text{mlist_replace} \, f \, p \, \{ \lambda_\exists \, \exists L' \, \rightarrow \, \text{Mlistof} \, S \, L' \mapsto J \, \exists \, L \, L' \}
\]

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

\[
\begin{align*}
  \{ p \mapsto \overline{R} \, (X : M) \mapsto J \, K \, K' \} & \quad \rightarrow \quad \text{expands to, for some } p', x, \\
  \{ p \mapsto ([x, p']) \mapsto x \mapsto RX \times p' \mapsto \overline{R} \, M \mapsto J \, K \, K' \} & \quad \rightarrow \quad \text{evaluates to } x \mapsto p' \, x, \\
  \{ p \mapsto ([x', p']) \mapsto x' \mapsto SX' \times p' \mapsto \overline{R} \, M \mapsto J \, (K \times X') \mapsto (K' \times X') \} & \quad \rightarrow \quad \text{assignment of } x' \mapsto \text{gives} \quad \text{by IH+frame, we get for some } M' \quad \rightarrow \quad \text{folding and rewriting}
\end{align*}
\]

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 \text{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 \text{mlist_replace}.

\[
\begin{align*}
\text{type} \quad \text{edge} \quad = \quad \{ \, \text{target} : \text{int}; \, \text{weight} : \text{int}; \, \text{mutable} \quad \text{mark} : \text{int} \, \} \\
\text{let} \quad \text{make_edge} \quad j \quad = \quad \{ \, \text{target} = j; \, \text{weight} = \text{Random}.\text{int} \, 2; \, \text{mark} = 0 \, \} \\
\text{let} \quad \text{augment_graph} \quad g \quad = \quad \text{Array}.\text{iter} \, \text{mlist_replace} \, \text{make_edge} \, \text{g}.\text{adj}
\end{align*}
\]

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

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

Question 3.10. Give a new representation predicate of an augmented graph \( g \mapsto \text{AugmentedGraph} \hat{G} \).

Answer. Let \( p \mapsto \text{Edge}(j, w, m) \mapsto \{ \text{target} = j, \text{weight} = w, \text{mark} = m \} \), then \( g \mapsto \text{AugmentedGraph} \quad (V, \hat{E}) \) is

\[
\exists L. \quad p \mapsto \text{ArrayOf} \, \text{(MlistOf Edge)} \quad L \times \forall \, \text{ijwm}, \quad (i, j, w, m) \in \hat{E} \mapsto (j, w, m) \in L[i]
\]

Question 3.11. Knowing a specification for \text{Random}.\text{int} can be: \( \forall n, \{ \, \text{‘n > 0’} \, \} \) \text{Random}.\text{int} \, n \, \{ \lambda_i. \, 0 \leq i < n \, \}, \text{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.

\[
\{ g \mapsto \text{Graph} \, (V, E) \} \quad \text{augment_graph} \quad g \quad \rightarrow \quad \text{AugmentedGraph} \, (V, \hat{E})
\]

\[
\forall (i, j, w, m) \in V^2, \quad (i, j, w, m) \in \hat{E} \iff (i, j) \in E \wedge w \in \{0, 1\} \wedge m = 0
\]