Final exam, March 8, 2022

- Duration: 3 hours.
- Answers may be written in English or French.
- 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 8 pages and **4** exercises. Exercise 1 is about verification using weakest preconditions. Exercises 2 to 4 are about separation logic.
- Don't forget to put your name on each sheet of paper, and page numbers under the form 1/7, 2/7, etc. **Please write your answers for Exercise 1 on a separate piece of paper.**

1 Cumulative Sums of Arrays

In this exercise, we are interested in specifying and proving correct data structures that support efficient computation of *array range sums*: given an array a of integers, and given a range delimited by indices i (inclusive) and j (exclusive), we wish to compute the value $\sum_{k=i}^{j-1} a[k]$. Such a sum can be computed using the following recursive program.

```
let rec sum (a: array int) (i j:int) : int
  = if j <= i then 0 else a[i] + sum a (i+1) j
```

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

Answer.

```
requires { 0 <= i /\ j < a.length } (* 0 <= i <= j < a.length is also OK *)
variant { j - i }
```

The precondition allows to show the array access safe, and to show that the precondition of the recursive calls is satisfied. The variant allows to prove the termination.

The program above being pure and terminating, it provides a logic function `sum` that can be used further in logical annotations. We consider now an imperative code for computing a range sum as follows.

```
let query_sum (a:array int) (i j:int) : int
  ensures { result = sum a i j }
= let ref s = 0 in
  for k=i to j-1 do s ← s + a[k] done;
  s
```

Question 1.2. *Which annotations should be given to the program above so as to be able to prove it safe, terminating and satisfying its post-condition? Justify informally (10 lines max) why your annotations suffice. If any logical property is needed on the* `sum` *logic function, state it explicitly as a lemma, and explain how this lemma should be proved.*

Answer. As a function contract:

```
requires { 0 <= i /\ j < a.length }
```

As a loop invariant:

```
invariant { s = sum a i k }
```

The preservation of the loop invariant amounts to show that

```
sum a i (k+1) = sum a i k + a[k]
```

This property is not a direct consequence of the definition of sum, but a property that can be proved by induction on `k`, as follows:

```
let rec lemma sum_right (a : array int) (i k : int)
  requires { 0 <= i <= k < a.length }
  variant { k - i }
  ensures { sum a i (k+1) = sum a i k + a[k] }
= if i < k then sum_right a (i+1) k
```

The *cumulative array* data structure is an idea to compute the range sum queries in constant time, the cost being transferred to the array update. The cumulative array of an array a of length l is an array of length $l + 1$ whose value at index i is $\sum_{k=0}^{i-1} a[k]$. To express that c is the cumulative array of a, we provide the predicate

```
predicate is_cumulative_array_for (c:array int) (a:array int) =
  c.length = a.length + 1 /\
  ∀ i. 0 <= i < c.length → c[i] = sum a 0 i
```

The creation of the cumulative array of an array a is thus made as follows

```
let create (a:array int) : array int
  ensures { is_cumulative_array_for result a }
= let l = a.length in let s = Array.make (l+1) 0 in
  for i=1 to l do
      s[i] ← s[i-1] + a[i-1]
  done;
  s
```

Question 1.3. *Which annotations should be given to the program above so as to be able to prove it safe, terminating and satisfying its postcondition? Justify informally (10 lines max) why your annotations suffice.*

Answer. The safety of array accesses and the termination is granted without need for more annotations, because a for loop is always terminating, and `1 <= i <= l` is true inside the loop body. To establish to post-condition, we pose the loop invariant

```
invariant { ∀ k. 0 <= k < i → s[k] = sum a 0 k }
```

Its preservation follows from the lemma `sum_right` already stated.

After creation of the cumulative array for a, it is not intended to keep a copy of a, so in the following a is kept only as a ghost parameter, for specification purposes. The constant-time function for range sum query is

```
let query (c:array int) (i j:int) (ghost a:array int): int
  requires { is_cumulative_array_for c a }
  ensures { result = sum a i j }
= c[j] - c[i]
```

Question 1.4. *Which annotations should be given to the program above so as to be able to prove it safe and satisfy its post-condition? Justify informally (10 lines max) why your annotations suffice. If any logical property is needed on the* `sum` *logic function, state it explicitly as a lemma, and explain how this lemma should be proved.*

Answer. The safety is guaranteed by the precondition

```
requires { 0 <= i <= j < c.length }
```

To establish to post-condition, one should prove that

$$\sum_{k=i}^{j-1} a[k] = \sum_{k=0}^{j-1} a[k] - \sum_{k=0}^{i-1} a[k]$$

which is a particular case of a general lemma on sums of concatenation of ranges, that can be stated and proved by induction as follows:

```
let rec lemma sum_concat (a:array int) (i j k:int) : unit
  requires { 0 <= i <= j <= k <= a.length }
  variant { j - i }
  ensures { sum a i k = sum a i j + sum a j k }
= if i < j then sum_concat a (i+1) j k
```

An update of the original array a should impose an update to its cumulative array. This is done by the following program.

```
let update (c:array int) (i:int) (v:int) (ghost a:array int) : unit
    requires { is_cumulative_array_for c a }
    writes { a, c }
    ensures { is_cumulative_array_for c a }
    ensures { a[i] = v }
    ensures { ∀ k. 0 <= k < a.length /\ k <> i → a[k] = (old a)[k] }
= let incr = v - c[i+1] + c[i] in a[i] ← v;
  for j=i+1 to c.length-1 do c[j] ← c[j] + incr done
```

Question 1.5. *Which annotations should be given to the program above so as to be able to prove it safe and satisfy its post-condition? Justify informally (20 lines max) why your annotations suffice. If any logical property is needed on the* `sum` *logic function, state it explicitly as a lemma, and explain how this lemma should be proved.*

Answer. The safety is ensured by the precondition

```
requires { 0 <= i < a.length }
```

The last two post-conditions are proved trivially as a consequence of the assignement `a[i] ← v`.
For proving the first post-condition, we pose the two following loop invariants.

```
invariant { ∀ k. 0 <= k < j → c[k] = sum a 0 k }
invariant { ∀ k. j <= k < c.length → c[k] = sum a 0 k - incr }
```

The first loop invariant stated that the cumulative array remains OK for indices lower than i. To prove it preserved, it is required to state a frame lemma for `sum`, as follows

```
let rec lemma sum_frame (a1 a2 : array int) (i j : int) : unit
  requires { 0 <= i <= j }
  requires { j <= a1.length }
  requires { j <= a2.length }
  requires { ∀ k : int. i <= k < j → a1[k] = a2[k] }
  variant { j - i }
  ensures { sum a1 i j = sum a2 i j }
= if i < j then sum_frame a1 a2 (i+1) j
```

The second loop invariant is proved also thanks to the frame lemma, but it is also required to state a change property of the sum predicate, for example as follows.

```
let rec lemma sum_update (a:array int) (i v l h:int) : unit
    requires { 0 <= l <= i < h <= a.length }
    variant { h }
    ensures { sum (a[i←v]) l h = sum a l h + v - a[i] }
= if h > i + 1 then sum_update a i v l (h-1)
```

---

2 Separation Logic: Heap Predicates

We recall that *heaps* are finite maps, or finite partial functions, from locations $l \in L$ to values $v \in V$, *i.e.* finite subsets m of $L \times V$ such that for all l, v_1, v_2: $(l, v_1) \in m \wedge (l, v_2) \in m \Rightarrow v_1 = v_2$. We write $|m|$ for the number of elements in m. We also recall the definition of two simple forms of heap predicates: the singleton heap predicate $\mapsto$, and the empty heap predicate with pure fact $[P]$:

$$\begin{array}{rcl} l \mapsto v & \equiv & \lambda m.\ m = \{(l, v)\} \ \wedge \ l \neq \mathsf{null} \\ [P] & \equiv & \lambda m.\ m = \varnothing \ \wedge \ P \end{array}$$

We also recall the definition of separating conjunction, $\star$ (note that $m_1 \perp m_2$ is short for $\mathsf{dom}(m_1) \cap \mathsf{dom}(m_2) = \varnothing$, where $\mathsf{dom}(m) \equiv \{l \mid (l, v) \in m\}$)

$$H_1 \star H_2 \quad \equiv \quad \lambda m.\ \exists m_1 m_2. \begin{cases} m_1 \perp m_2 \\ m = m_1 \uplus m_2 \\ H_1\, m_1 \\ H_2\, m_2 \end{cases}$$

Question 2.1. *Give a detailed proof that* $((a \mapsto b) \star (b \mapsto a))(m)$ *implies* $a \neq b$ *(3 lines should be enough).*

Answer. There exist m_1 and m_2 such that $m_1 \perp m_2$, $m = m_1 \uplus m_2$, $(a \mapsto b)\, m_1$, $(b \mapsto a)\, m_2$. Therefore, $m_1 = \{(a, b)\}$ and $m_2 = \{(b, a)\}$, so $\{(a, b)\} \perp \{(b, a)\}$, which means $\mathsf{dom}(\{(a, b)\}) \cap \mathsf{dom}(\{(b, a)\}) = \{a\} \cap \{b\} = \varnothing$, which is equivalent to $a \neq b$.

We recall the definition of heap entailment $\rhd$ and of existential quantification $\exists$:

$$\begin{array}{rcl} H_1 \rhd H_2 & \equiv & \forall m.\, H_1\, m \Rightarrow H_2\, m \\ \exists x.\, H & \equiv & \lambda m. \exists x. H m \end{array}$$

Question 2.2. *Let* GC *be the predicate* $\exists H.\, H$*. Give a detailed proof that* $(a \mapsto b) \star (b \mapsto a) \rhd [a \neq b] \star \mathsf{GC}$ *(6 lines should be enough).*

Answer. Let m such that $((a \mapsto b) \star (b \mapsto a))(m)$, we need to prove that $([a \neq b] \star \mathsf{GC})(m)$. We choose $m_1 = \varnothing$ the empty heap and $m_2 = m$. We indeed have that $m_1 \perp m_2$ since $\mathsf{dom}(m_1) \cap \mathsf{dom}(m_2) = \varnothing \cap \mathsf{dom}(m) = \varnothing$, and $m_1 \uplus m_2 = \varnothing \uplus m = m$. It remains to prove that $[a \neq b](m_1)$, which is $m_1 = \varnothing \wedge a \neq b$, which holds by definition of m_1 and by the previous question, and that $\mathsf{GC}(m_2)$, i.e. that there exists H such that $H\, m_2$, which works by choosing $H = (a \mapsto b) \star (b \mapsto a)$.

We also recall the definitions of (non-separating) conjunction ($\curlywedge$) and disjunction ($\curlyvee$):

$$\begin{array}{rcl} H_1 \curlywedge H_2 & \equiv & \lambda m.\ H_1\, m \wedge H_2\, m \\ H_1 \curlyvee H_2 & \equiv & \lambda m.\ H_1\, m \vee H_2\, m \end{array}$$

Question 2.3. *Prove that* $(a \mapsto b) \curlywedge (b \mapsto a) \rhd [a = b] \star \mathsf{GC}$.

Answer. Let m such that $((a \mapsto b) \curlywedge (b \mapsto a))(m)$, we need to prove that $([a = b] \star \mathsf{GC})(m)$ i.e. we need to prove that $a = b$. By definition of $\curlywedge$ we have $(a \mapsto b)(m)$ and $(b \mapsto a)(m)$ which means that $m = \{(a, b)\} = \{(b, a)\}$, which means that $a = b$.

Question 2.4. *For each of the following equations of the form* $A \stackrel{?}{=} B$*, say whether* $A \rhd B$ *holds or not, and whether* $B \rhd A$ *holds or not. When the entailment holds, shortly explain why, and when it does not, give a counterexample.*

$$H_1 \curlywedge (H_2 \curlyvee H_3) \stackrel{?}{=} (H_1 \curlywedge H_2) \curlyvee (H_1 \curlywedge H_3) \tag{1}$$

$$H_1 \curlyvee (H_2 \curlywedge H_3) \stackrel{?}{=} (H_1 \curlyvee H_2) \curlywedge (H_1 \curlyvee H_3) \tag{2}$$

$$H_1 \star (H_2 \curlyvee H_3) \stackrel{?}{=} (H_1 \star H_2) \curlyvee (H_1 \star H_3) \tag{3}$$

$$H_1 \curlyvee (H_2 \star H_3) \stackrel{?}{=} (H_1 \curlyvee H_2) \star (H_1 \curlyvee H_3) \tag{4}$$

Answer. The first two hold in both directions: the first is the consequence of $\wedge$ distributing over $\vee$ in propositional logic, the second of $\vee$ distributing over $\wedge$.
$H_1 \star (H_2 \curlyvee H_3) \stackrel{?}{=} (H_1 \star H_2) \curlyvee (H_1 \star H_3)$ hold in both directions:

($\rhd$): suppose $m_1 \perp m_2$ with $H_1\, m_1$ and either $H_2\, m_2$ or $H_3\, m_2$. Then either $(H_1 \star H_2)(m_1 \uplus m_2)$ or $(H_1 \star H_3)(m_1 \uplus m_2)$, QED.

($\lhd$): in both cases there are some $m_1 \perp m_2$ with $H_1\, m_1$ and either $H_2\, m_2$ or $H_3\, m_2$ (even though the m_1 need not be the same), which means $(H_2 \curlyvee H_3)\, m_2$, QED.

$H_1 \curlyvee (H_2 \star H_3) \stackrel{?}{=} (H_1 \curlyvee H_2) \star (H_1 \curlyvee H_3)$ does not hold in either direction, which can be seen easily with $H_2 = H_3 = [\mathsf{False}]$, for which it simplifies to $H_1 \stackrel{?}{=} H_1 \star H_1$, which is false in both directions *e.g.* with $H_1 = \exists l.\, l \mapsto 0$.

Let $-\!\!\star$ be the "magic wand" or "separating implication" operator on heap predicates defined by:

$$H_1 -\!\!\star H_2 \quad \equiv \quad \lambda m.\ \forall m_1\ (m_1 \perp m \wedge H_1\, m_1) \Rightarrow H_2(m_1 \uplus m)$$

Question 2.5. *Show that the following rules hold:*

$$\frac{H_1 \star H_2 \rhd H_3}{H_1 \rhd (H_2 -\!\!\star H_3)} \qquad \frac{H_1 \rhd (H_2 -\!\!\star H_3) \qquad H_4 \rhd H_2}{H_1 \star H_4 \rhd H_3}$$

Answer. Suppose $H_1 \star H_2 \rhd H_3$, we need show that $H_1 \rhd (H_2 -\!\!\star H_3)$, *i.e.* let m_1 such that $H_1\, m_1$, we show that $(H_2 -\!\!\star H_3)m_1$, *i.e.* let m_2 such that $m_2 \perp m_1$ and $H_2\, m_2$, we show that $H_3(m_1 \uplus m_2)$, which is implied (thanks to $H_1 \star H_2 \rhd H_3$) by $(H_1 \star H_2)(m_1 \uplus m_2)$, which holds in turn by definition of $\star$.

Suppose now that $H_1 \rhd (H_2 -\!\!\star H_3)$ and that $H_4 \rhd H_2$, and let us prove that $H_1 \star H_4 \rhd H_3$. Let $m_1 \perp m_4$ such that $H_1\, m_1$ and $H_4\, m_4$ and show that $H_3(m_1 \uplus m_4)$. By definition of $\rhd$ we have $H_2\, m_4$ and $(H_2 -\!\!\star H_3)\, m_1$, which expands to $\forall m\ (m \perp m_1 \wedge H_2\, m) \Rightarrow H_3(m_1 \uplus m)$, and we conclude by choosing $m = m_4$.

Question 2.6. *For each of the following equation $A \stackrel{?}{=} B$, prove or disprove $A \rhd B$ and $B \rhd A$.*

$$([\,] -\!\!\star H) \stackrel{?}{=} H \tag{5}$$

$$([\mathsf{False}] -\!\!\star H) \stackrel{?}{=} [\,] \tag{6}$$

$$(H -\!\!\star [\,]) \stackrel{?}{=} [H \rhd [\,]] \tag{7}$$

$$(l \mapsto v -\!\!\star [\mathsf{False}]) \stackrel{?}{=} (l \mapsto \_ \star \mathsf{GC}) \tag{8}$$

(assume that $l \neq \mathsf{null}$)

Answer.

- $([\,] -\!\!\star H) \rhd H$ holds: second rule using $H_1 = [] -\!\!\star H$, $H_2 = []$, $H_3 = H$, $H_4 = []$:

$$\dfrac{\dfrac{}{([] -\!\!\star H) \rhd ([] -\!\!\star H) \star []}\ {\scriptstyle []\text{-NEUTRAL-FOR-}\star} \qquad \dfrac{\dfrac{}{([] -\!\!\star H) \rhd ([] -\!\!\star H)}\ {\scriptstyle \rhd\text{-REFL}} \qquad \dfrac{}{[] \rhd []}\ {\scriptstyle \rhd\text{-REFL}}}{([] -\!\!\star H) \star [] \rhd H}\ {\scriptstyle -\!\!\star\text{-ELIM}}}{[] -\!\!\star H \rhd H}\ {\scriptstyle \rhd\text{-TRANS}}$$

- $([\,] -\!\!\star H) \lhd H$ holds: first rule using $H_1 = H_3 = H$ and: $H_2 = []$: $\dfrac{H \star [\,] \rhd H}{H \rhd ([\,] -\!\!\star H)}$
- $([\mathsf{False}] -\!\!\star H) \rhd [\,]$ does not hold: with any $m \neq \varnothing$, the LHS holds but the RHS does not.
- $([\mathsf{False}] -\!\!\star H) \lhd [\,]$ holds, since $[\mathsf{False}] -\!\!\star H$ holds for any m.
- $(H -\!\!\star [\,]) \rhd [H \rhd [\,]]$ does not hold *e.g.* with $H = [\mathsf{False}]$ and any $m \neq \varnothing$.
- $(H -\!\!\star [\,]) \lhd [H \rhd [\,]]$ holds: suppose $m = \varnothing$ and $H \rhd [\,]$. Let m_1 such that $m_1 \perp m$ and $H\, m_1$ (and hence $[\,]m_1$ *i.e.* $m_1 = \varnothing$), we need to prove that $[\,](m_1 \uplus m)$ i.e. that $(m_1 \uplus m) = \varnothing$, which holds since $m = \varnothing = m_1$.
- $(l \mapsto v -\!\!\star [\mathsf{False}]) \rhd (l \mapsto \_ \star \mathsf{GC})$: Let m such that $(l \mapsto v -\!\!\star [\mathsf{False}])$. In other words, for every m_1 such that $m_1 \perp m$ and $(l \mapsto v)m_1$, we have $[\mathsf{False}](m \uplus m_1)$ which is equivalent to False.

  It is thus impossible to simultaneously have $m_1 \perp m$ and $(l \mapsto v)m_1$.

  Considering $m_1 = \{(l, v)\}$, it means that $m_1 \perp m$ does not hold, *i.e.* that $\mathsf{dom}(\{(l, v)\}) \cap \mathsf{dom}(m) \neq \varnothing$, *i.e.* that $l \in \mathsf{dom}(m)$.

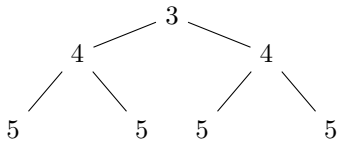
  This means that $m = \{(l, m(l))\} \uplus (m \backslash \{(l, m(l))\})$ and hence m satisfies $(l \mapsto \_ \star \mathsf{GC})$.

- $(l \mapsto v \mathrel{-\!\!\star} [\mathsf{False}]) \lhd (l \mapsto \_ \star \mathsf{GC})$ holds: let m satisfying the RHS. Then $l \in \mathsf{dom}(m)$. Then any m_1 satisfying both $m_1 \perp m$ and $(l \mapsto v)m_1$ both does not contain and does contain l, hence the contradiction.

---

3 Separation Logic: Mutable Trees from Mutable Lists

Hugo works in a startup that specializes in decision trees. He needs to turn lists into binary trees in such a way that the list `[3; 4; 5]` would be turned into the following tree:



He writes the following function on mutable lists:

```
let rec mtree_of_mlist (p : 'a cell) : 'a node =
  if p == null then null else
    let t = mtree_of_mlist p.tl in
    { item = p.hd; left = t; right = t }
```

He also defines the pure function $\mathsf{TreeOfList}$ of type $\forall A.\ \mathsf{list}\, A \to \mathsf{tree}\, A$ as:

$$\begin{aligned} \mathsf{TreeOfList}\,\mathsf{nil}) &= \mathsf{Leaf} \\ \mathsf{TreeOfList}(x :: L) &= \mathsf{Node}\, x\, (\mathsf{TreeOfList}\, L)\, (\mathsf{TreeOfList}\, L) \end{aligned}$$

Hugo now wants to relate the two using the following specification:

$$\forall p\, L,\ \{p \rightsquigarrow \mathsf{Mlist}\, L\}(\texttt{mtree\_of\_mlist p})\{\lambda r.\, r \rightsquigarrow \mathsf{Mtree}(\mathsf{TreeOfList}\, L)\} \tag{9}$$

Question 3.1. *Prove that specification* (9) *does not hold. Give another of its disadvantages.*

Answer. If L is of length two or more, there is sharing, and $p \rightsquigarrow \mathsf{Mtree}$ forbids sharing between subtrees. Also, the specification forgets about $p \rightsquigarrow \mathsf{Mlist}\, L$.

It turns out that Hugo's functions must be used with no modification because he is the CEO.

Question 3.2. *Find a new specification for* `mtree_of_mlist` *and a new specification for a known function* `f` *such that the composition of* `f` *and* `mtree_of_mlist` *can be shown to satisfy specification* (9).

Answer. We choose `f` = `tree_copy`. There are several ways to go about the new specification. One way is to introduce the derived operator $\star^?$ defined by $A \star^? B \ \equiv\ (A \star \mathsf{GC}) \mathbin{⩓} (B \star \mathsf{GC})$. The new specification is (9) where we replace `Mtree` with `Mtree`$^?$, defined by

$$\begin{aligned} p \rightsquigarrow \mathsf{Mtree}^? T \ \equiv\ & \mathsf{match}\, T\, \mathsf{with} \\ & \mid \mathsf{Leaf} \Rightarrow [p = \mathsf{null}] \\ & \mid \mathsf{Node}\, x\, T_1\, T_2 \Rightarrow \exists p_1 p_2. \\ & \qquad p \mapsto \{\!|\mathrm{item}{=}x;\ \mathrm{left}{=}p_1;\ \mathrm{right}{=}p_2|\!\} \\ & \qquad \star\ (p_1 \rightsquigarrow \mathsf{Mtree}^? T_1 \quad \star^? \quad p_2 \rightsquigarrow \mathsf{Mtree}^? T_2) \end{aligned}$$

and the new specification for `tree_copy` is $\{\mathsf{RO}(p \rightsquigarrow \mathsf{Mtree}^? T)\}(\texttt{tree\_copy p})\{\lambda r. r \rightsquigarrow \mathsf{Mtree} T\}$.

Question 3.3. *Prove that* `mtree_of_mlist` *satisfies its new specification. Be sure to state your induction hypothesis with the right quantifiers.*

Answer. We prove by induction on L that

$$\forall p \quad \{p \rightsquigarrow \mathsf{Mlist}\, L\}(\texttt{mtree\_of\_mlist p})\{\lambda r.\, r \rightsquigarrow \mathsf{Mtree}^?(\mathsf{TreeOfList}\, L)\} \tag{10}$$

when $L = \mathsf{nil}$, p is null so `mtree_of_mlist p` returns $r = \mathsf{null}$, which does satisfy $r \rightsquigarrow \mathsf{Mtree}^? \mathsf{Leaf}$. Suppose now that (10) holds for L, we show it holds for $x :: L$.

From precondition $p \rightsquigarrow \mathsf{Mtree}\,(x :: L)$ we get precondition $p \rightsquigarrow \{x; q\} \star q \rightsquigarrow \mathsf{Mtree}\,L$ for some q, hence p is not null. By using A-normal form, `p.tl` resolves to q, and by framing out $p \rightsquigarrow \{x; q\}$ then the induction hypothesis, we get after `let t = mtree_of_mlist q` the intermediate condition $p \rightsquigarrow \{x; q\} \star t \rightsquigarrow \mathsf{Mtree}^?(\mathsf{TreeOfList}\,L)$. Then, `p.hd` resolves to x, we can garbage-collect away $p \rightsquigarrow \{x; q\}$, and after the cell allocation `item = x; left = t; right = t` we get that the return value r satisfies

$$r \rightsquigarrow \{\mathsf{item} = x; \mathsf{left} = t; \mathsf{right} = t\} \star t \rightsquigarrow \mathsf{Mtree}^?(\mathsf{TreeOfList}\,L)$$

and because $H \rhd (H \star \mathsf{GC}) \rhd (H \star^? H)$ for any H, we can derive the postcondition

$$r \rightsquigarrow \{\mathsf{item} = x; \mathsf{left} = t; \mathsf{right} = t\} \star (t \rightsquigarrow \mathsf{Mtree}^?(\mathsf{TreeOfList}\,L) \star^? t \rightsquigarrow \mathsf{Mtree}^?(\mathsf{TreeOfList}\,L))$$

which entails in turn $r \rightsquigarrow \mathsf{Mtree}^?(\mathsf{TreeOfList}\,(x :: L)))$ by choosing $p_1 = p_2 = t$.

Question 3.4. *Give a proof sketch that* `f` *satisfies its new specification.*

Answer. For the specification

$$\{\mathsf{RO}(p \rightsquigarrow \mathsf{Mtree}^?T)\}(\texttt{tree\_copy p})\{\lambda r.r \rightsquigarrow \mathsf{Mtree}T\}$$

we exploit the fact that $\mathsf{RO}(...)$ can be duplicated. For a specification of the form

$$\{p \rightsquigarrow \mathsf{Mtree}^?T\}(\texttt{tree\_copy p})\{\lambda r.p \rightsquigarrow \mathsf{Mtree}^?T \star r \rightsquigarrow \mathsf{Mtree}T\}$$

no need for duplication, but we use the fact that $A \star^? B$ can be replaced with A or with B in any precondition (using the COMBINE rule). That also works if we used $\curlywedge$ instead of $\star^?$.

Question 3.5. *Adapt the new specification for* `f` *so that it works for Question 3.2 and so that* `f`*'s usual specification can be derived from it.*

Answer. It is already more general than the usual specification

$$\{p \rightsquigarrow \mathsf{Mtree}\,T\}(\texttt{tree\_copy p})\{\lambda r.p \rightsquigarrow \mathsf{Mtree}\,T \star r \rightsquigarrow \mathsf{Mtree}\,T\}$$

since the latter can be derived by rule FRAME-RO from

$$\{\mathsf{RO}(p \rightsquigarrow \mathsf{Mtree}\,T)\}(\texttt{tree\_copy p})\{\lambda r. \rightsquigarrow \mathsf{Mtree}\,T\}$$

which can be derived from the new specification by the consequence rule, by proving that $(p \rightsquigarrow \mathsf{Mtree}T) \rhd (p \rightsquigarrow \mathsf{Mtree}^?T)$ and hence $\mathsf{RO}(p \rightsquigarrow \mathsf{Mtree}T) \rhd \mathsf{RO}(p \rightsquigarrow \mathsf{Mtree}^?T)$. It would also have been possible to use a more custom $\mathsf{Mtree}^?$ (for example with a list as an argument or using $\curlywedge$ instead of $\star^?$) and then to more artificially combine the two. Combining two specifications $\forall \vec{x_1}, \{H_1\}c\{Q_1\}$ and $\forall \vec{x_2}, \{H_2\}c\{Q_2\}$ can always be done via the construction $\forall i \in \{1,2\}\ \forall \overrightarrow{x_1 \cup x_2}\ \{([i=1] \curlywedge H_1) \curlyvee ([i=2] \curlywedge H_2)\}c\{\lambda r.([i=1] \curlywedge Q_1 r) \curlyvee ([i=2] \curlywedge Q_2 r)\}$

---

4 Separation Logic: List Partitions

We consider the following OCaml function on pure lists:

```
let rec partition_tf (f : 'a -> bool) : 'a list -> 'a list * 'a list =
  function
  | [] -> ([], [])
  | x :: l -> let y = f x in
              let (a, b) = partition_tf f l in
              if y then (x :: a, b) else (a, x :: b)
```

Question 4.1. *Give a specification for* `partition_tf f l` *in the case when* `f` *is a pure deterministic function that can be represented by a logical function* $F : A \to$ `bool` *(and* `l` *is a pure list). Your specification should show explicitly how* `f` *and* F *are related.*

Answer. Writing $\overline{F}$ for $\lambda x.\neg(Fx)$,

$$(\forall x\ \{[\,]\}(\texttt{f x})\{\lambda b.[b = Fx]\}) \quad \Rightarrow \quad \{[\,]\}(\texttt{partition\_tf f l})\{\lambda r.[r = (\mathsf{Filter}\ F\ l, \mathsf{Filter}\ \overline{F}\ l)]\}$$

Suppose now that `f` cannot be represented a pure deterministic function F, but is still pure in the sense that it can be represented by a binary predicate P of type $A \to$ `bool` $\to \mathsf{Prop}$, so that a precondition for the specification of `partition_tf f` is $\forall x\ \{[\,]\}(\texttt{f x})\{\lambda b.[P\,x\,b]\}$.

Question 4.2. *Give some examples where* `f` *is pure in the above sense but such that the previous specification is not sufficient to establish a satisfactory Hoare triple for* `partition_tf f`.

Answer. We can have an f that is underspecified on some inputs x, for which we would have both $P\,x\,\mathsf{true}$ and $P\,x\,\mathsf{false}$. In fact f could be non-deterministic such as `fun _ → Random.bool ()` (though it is in fact not really pure), or if f cannot be easily expressed as a logical function, for example if f is supposed to implement the inverse of a cryptographic hash function, or if $f\,x$ is the result of some complex analysis on x if x is a program or a logical formula.

Question 4.3. *Give a more general specification for* `partition_tf f` *in the case when* `f` *is a pure function that can be represented by a binary predicate* P.

Answer.

$$(\forall x\ \{[]\}(\texttt{f x})\{\lambda b.[P\,x\,b]\}) \quad\Rightarrow\quad \{[]\}(\texttt{partition\_tf f l})\{\lambda r.[\exists l', \mathsf{Forall2}\,P\,l\,l' \wedge \mathsf{select}\ l\ l'\ r]\}$$

where select is a predicate inductively defined by

$$\frac{}{\mathsf{select}\ \mathsf{nil}\ \mathsf{nil}\ (\mathsf{nil},\mathsf{nil})} \qquad \frac{\mathsf{select}\ l\ l'\ (a,b)}{\mathsf{select}\ (x :: l)\ (\mathsf{true} :: l')\ (x :: a, b)} \qquad \frac{\mathsf{select}\ l\ l'\ (a,b)}{\mathsf{select}\ (x :: l)\ (\mathsf{false} :: l')\ (a, x :: b)}$$

Question 4.4. *Give a specification for* `partition_tf f l` *in the more general case where* `f` *is not necessarily deterministic or pure. You may use an auxiliary predicate of type* $\mathsf{list}\,A \to \mathsf{list}\,A \to \mathsf{list}\,A \to \mathsf{Hpred}$.

Answer.

$$\begin{aligned}&(\forall x lab\ \{J\ l\ a\ b\}(\texttt{f x})\{\lambda y.\ \begin{array}{l}\mathsf{if}\ y\ \mathsf{then}\ J\ (l\&x)\ (a\&x)\ b\\ \mathsf{else}\ J\ (l\&x)\ a\ (b\&x)\end{array}\ \})\\ \Rightarrow\quad &\{J\ \mathsf{nil}\ \mathsf{nil}\ \mathsf{nil}\}(\texttt{partition\_tf f l})\{\lambda(a,b).J\ l\ a\ b\}\end{aligned}$$

Question 4.5. *Give an implementation and a specification for the variant* `mutable_partition_tf f p` *that takes a mutable list as argument and returns a mutable record of two mutable lists (no proof required).*

Answer. For example,

```
let rec mutable_partition_tf f p =
  if p == null then { trues = null; falses = null } else
    let y = f p.hd in
    let q = mutable_partition_tf f p.tl in
    if y then q.trues  <- { hd = p.hd; tl = q.trues  }
    else      q.falses <- { hd = p.hd; tl = q.falses }
```

Not much change in the specification:

$$\begin{aligned}&(\forall x lab\ \{J\ l\ a\ b\}(\texttt{f x})\{\lambda y.\ \begin{array}{l}\mathsf{if}\ y\ \mathsf{then}\ J\ (l\&x)\ (a\&x)\ b\\ \mathsf{else}\ J\ (l\&x)\ a\ (b\&x)\end{array}\ \})\\ \Rightarrow\quad &\{J\ \mathsf{nil}\ \mathsf{nil}\ \mathsf{nil} \star p \rightsquigarrow \mathsf{Mlist}\,l\}(\texttt{partition\_tf f l})\{\lambda r.J\ l\ a\ b \star p \rightsquigarrow \mathsf{Mlist}\,l \star \exists uv.\ \begin{array}{l}r \rightsquigarrow \{u;v\}\\ \star\ u \rightsquigarrow \mathsf{Mlist}\ a\\ \star\ v \rightsquigarrow \mathsf{Mlist}\ b\end{array}\ \}\end{aligned}$$
