Final exam, March 10, 2021

- There are 9 pages and **3** exercises. Exercise 1 is about weakest preconditions. Exercises 2 and 3 are about separation logic. It is advised to do Exercise 2 before Exercise 3.
- Duration: 3 hours.
- Answers must be written by hand on some paper. They can be written in English or French.
- Don't forget to put your name on the each sheet of paper, and page numbers under the form 1/7, 2/7, etc.
- **How to return your answers:** at the end of the exam session, you must scan the sheets of paper you wrote, so as to produce a PDF file. Send that PDF file by e-mail to `Claude.Marche@inria.fr` and `Jean-Marie.Madiot@inria.fr`.

1 Loops of the form "repeat ... until"

In this exercise, we consider the programming language of the first part of the course, involving expressions with side effects but no pointers (as defined in lectures 2 and 3). We propose to add to this language a construct for "repeat" loops, with a syntax of the form

$$\texttt{repeat}\ e_1\ \texttt{until}\ e_2\ \texttt{done}$$

The expression e_1 must be of type `unit` and e_2 must be of type `bool`. Informally, the operational semantics is that e_1 is executed once, then e_2 is evaluated to a Boolean value v. If $v = \texttt{true}$ then execution stops and returns `()`, otherwise the execution of the loop starts again. Since the loop body is executed at least once, a natural idea is to state a loop invariant at the *end* of the loop body, under the syntax

$$\texttt{repeat}\ e_1\ \texttt{invariant}\ I\ \texttt{until}\ e_2\ \texttt{done}$$

It is formalized using the following operational semantics rule:

$$\frac{}{\Sigma, \Pi, \texttt{repeat}\ e_1\ \texttt{invariant}\ I\ \texttt{until}\ e_2\ \texttt{done} \rightsquigarrow \Sigma, \Pi, e_1; \texttt{assert}\ I; \texttt{if}\ e_2\ \texttt{then}\ ()\ \texttt{else repeat}\ e_1\ \texttt{invariant}\ I\ \texttt{until}\ e_2\ \texttt{done}}$$

An example program is as follows:

```
val ref m : int

let f (a : array int) : int
  writes a, m
  ensures 1 <= result <= a.length
  ensures result < a.length → m < 0
  ensures m = a[result-1]+1
  ensures m = a@Old[result-1]
  = let ref i = 0 in
    repeat
      m ← a[i];
      a[i] ← m - 1;
    invariant ???
    until (i ← i+1; i = a.length \/ m < 0)
    done;
    i
```

We emphasize, as shown in the operational semantics rule and illustrated in the example above, that the loop condition may have side effects, and that the invariant is supposed to hold *before* these side effects take place.

Question 1.1. *Propose a rule for computing the weakest precondition (WP) of a "repeat" loop. Explain informally (in 10 lines maximum) how you constructed your formula, and why the validity of this formula ensures that the corresponding "repeat" loop must executes safely.*

Answer. The rule can be more or less derived from the rule for "while" loops.

$$\begin{array}{l}\mathrm{WP}(\texttt{repeat}\ e\ \texttt{invariant}\ I\ \texttt{until}\ c\ \texttt{done}, Q) = \\ \quad \mathrm{WP}(e, I) \wedge \\ \quad \forall \vec{v}, (I \rightarrow \mathrm{WP}(c, \texttt{if}\ \mathit{result}\ \texttt{then}\ Q\ \texttt{else}\ \mathrm{WP}(e, I))) \\ \qquad [w_i \leftarrow v_i])\end{array}$$

where the w_i are the variables modified in the loop. The differences with the rule for "while" loops is first that the invariant must be initially establish through one first iteration of the body (first line above) and second that the case of the if must be inverted because the condition has a reversed meaning.
The rule can be justified also directly by showing that its validity implies that an arbitrary execution

$$e; \texttt{assert}\ I; (c \text{ (assumed false)}; e; \texttt{assert}\ I)^*; c \text{ (assumed true)}; \texttt{assert}\ Q$$

is safe.

Question 1.2. *Propose additional annotations (e.g. pre-conditions, loop invariants) for the example program, so as to prove that the array accesses are inside the correct bounds. Justify informally (5 lines max) why your annotations suffice.*

Answer.

```
requires { a.length >= 1 }
invariant { 0 <= i < a.length }
```

the pre-condition is needed to ensure the validity of accesses at the first iteration, when $i = 0$. The invariant is initially true ($i = 0$) and easily preserved thanks to the first part of the loop condition. And is enough to prove validity of accesses in subsequent iterations.

Question 1.3. *Propose suitable additional annotations so as to prove the first post-condition. Justify informally (5 lines max) why your annotations suffice.*

Answer. The post-condition is directly implied by the invariant of the previous question, after taking the side effect of the loop condition into account.

Question 1.4. *Propose suitable additional annotations so as to prove the second post-condition. Justify informally why your annotations suffice.*

Answer. The post-condition is directly implied by loop condition being false

---

Question 1.5. *Propose suitable additional annotations so as to prove the third post-condition. Justify informally why your annotations suffice.*

Answer.

```
invariant { m = a[i]+1 }
```

It is establish by each loop body, indeed even without knowing it is true before. It entails the post-condition, after taking into account the side effect in the loop condition.

Question 1.6. *Propose suitable additional annotations so as to prove the fourth post-condition. Justify informally why your annotations suffice.*

Answer. Even if very similar to the previous post-condition, this one does not come for free because we have to state how the array cells may have been modified since the beginning of the loop. In particular we need to state that the values for indexes larger than `i` were unchanged.

```
invariant { ∀ j. i < j < a.length → a[j] = a@init[j] }
```

where `init` is a label at the beginning of the function body (`old` would be an accepted answer too, although not formally defined in the lecture). It is trivially true at loop entrance, and easily seen preserved. Yet, it is still insufficient to establish to post-condition, because the only knowledge we have of `a@Old[result-1]` at exit is what is in the invariant. We have also to state

```
invariant { m = a@init[i]+1 }
```

This invariant is establish at each loop iteration, from the previous invariant above. And this time it entails the post-condition.

Let us now consider the problem of proving termination of "repeat" loops.

Question 1.7. *Suggest a form of annotation to add to "repeat" loops so has to be able to prove termination of such a loop. Provide a new form of the WP rule accordingly. Show the termination of the "repeat" loop of the example program.*

Answer. A natural idea is to allow to add a `variant` clause, as for "while loops". The new WP rule would be

$$\begin{array}{l}\mathrm{WP}(\texttt{repeat}\ e\ \texttt{invariant}\ I\ \texttt{variant}\ t\ \texttt{until}\ c\ \texttt{done}, Q) = \\ \quad \mathrm{WP}(e, I) \wedge \\ \quad \forall \vec{v}, (I \rightarrow \mathrm{WP}(L : c, \texttt{if}\ \mathit{result}\ \texttt{then}\ Q\ \texttt{else}\ \mathrm{WP}(e, I \wedge t \prec t@L))) \\ \qquad [w_i \leftarrow v_i])\end{array}$$

For the example program, a suitable variant is

```
variant a.length - i
```

which evidently decreases at each loop iteration, while remaining non-negative.

---

2 Separation logic: sizes

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: singleton heap predicate $\mapsto$, and 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. *Prove that* $((r \mapsto 0) \star (s \mapsto 0))(m)$ *implies* $|m| = 2$.

Answer. Let m such that $((r \mapsto 0) \star (s \mapsto 0))(m)$. By definition of $\star$ there exists m_1, m_2 such that $m_1 \perp m_2$, $m = m_1 \uplus m_2$, $(r \mapsto 0)(m_1)$, and $(s \mapsto 0)(m_2)$. This means that $m_1 = \{(r, 0)\}$ and $m_2 = \{(s, 0)\}$. Since $m_1 \perp m_2$, we know that $r \neq s$ and since $m = \{(r, 0), (s, 0)\}$, $|m| = 2$.

We 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 2.2. *Prove that* $((r \mapsto 0) \curlywedge (s \mapsto 0))(m)$ *implies* $|m| = 1$.

Answer. By definition of $\curlywedge$ $(r \mapsto 0)(m)$ and $(s \mapsto 0)(m)$, so $m = \{(r, 0)\}$ which means that $|m| = 1$. We know in addition that $s = r$.

For every integer $n \in \mathbb{Z}$, we define the heap predicate $\square(n)$ as $\lambda m.(|m| = n)$.

Question 2.3. *Give a formula (not using the $\square$ symbol) that is equivalent to $\square(0)$. Justify your answer.*

Answer. $\square(0)(m) \Leftrightarrow (|m| = 0) \Leftrightarrow (m = \varnothing) \Leftrightarrow [](m)$, so an equivalent formula is $[]$, or $[\mathsf{True}]$, or **emp**.

Question 2.4. *Give a formula (not using the $\square$ symbol) that is equivalent to $\square(-1)$. Justify your answer.*

Answer. $\square(-1)(m) \Leftrightarrow (|m| = -1)$ which is never true, since the size of a set is never negative. An equivalent formulat is hence $[\mathsf{False}]$ or $[\mathsf{False}] \star H$ for any H.

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 2.5. *Show that $(\square(1) \star \square(2)) \rhd \square(3)$ and that $(\square(1) \twoheadrightarrow \square(2)) \rhd \square(-1)$.*

Answer. Let m such that $(\square(1) \star \square(2))(m)$. There exists m_1, m_2 such that $m_1 \perp m_2$, $m = m_1 \uplus m_2$, $\square(1)(m_1)$, and $\square(2)(m_2)$. Then, $|m_1| = 1$, $|m_2| = 2$, so $m = |m_1 \uplus m_2| = |m_1| + |m_2| = 3$, and so $\square(3)(m)$.
Suppose now $(\square(1) \twoheadrightarrow \square(2))(m)$. Then by definition of $\twoheadrightarrow$, $|m| = 1$ and $|m| = 2$, which implies false (and in particular $|m| = 1 = 2 - 1 = |m| - |m| = 1 - 2 = -1$), which implies $\square(-1)(m)$.

Question 2.6. *Do we have $(\exists l.\, \exists v.\, l \mapsto v) \rhd \square(1)$? Why? Same questions for $\square(1) \rhd (\exists l.\, \exists v.\, l \mapsto v)$.*

Answer. Yes, $(\exists r.\, \exists l.\, l \mapsto v)(m)$ implies that $m = \{(l, v)\}$ for some l, v, and so $|m| = 1$. The opposite does not quite hold, however, as m might be $\{(l, v)\}$ with $l = \mathsf{null}$.

Question 2.7. *On which condition on the integers n and k do we have $(\square(n) \star \square(k)) \rhd \square(n + k)$?*

Answer. No condition is required, since if $|m_1| = n$ and $|m_2| = k$ then $|m_1 \uplus m_2| = n + k$.

Question 2.8. *On which condition on the integers n and k do we have $\square(n + k) \rhd (\square(n) \star \square(k))$?*

Answer. On the condition $(n + k < 0) \vee (n \geqslant 0 \wedge k \geqslant 0)$. Suppose $|m| = n + k$, the problem is to find a condition under which there exist disjoint m_1, m_2 such that $m = m_1 \uplus m_2$ and $|m_1| = n$ and $|m_2| = k$, which is possible if $k \geqslant 0$ and $n \geqslant 0$. It (as everything) is also possible when $n + k < 0$.

Question 2.9. *When is the rule* $\dfrac{\{\square(n_1)\}\ c\ \{\lambda v.\ \square(n_2)\}}{\{\square(n_1 + k)\}\ c\ \{\lambda v.\ \square(n_2 + k)\}}$ *derivable for every program c? Justify your answer carefully by explicitely mentioning the names of the rules you are using.*

Answer. When $(n_1 + k < 0) \vee (n_1 \geqslant 0 \wedge k \geqslant 0)$. In that case, it is an instance of the frame rule, by adding $\square(k)$ on each side, combined with the consequence rule, and the two previous questions.

$$\dfrac{\dfrac{\text{QUESTION 2.8}\quad (n_1 + k < 0) \vee (n_1 \geqslant 0 \wedge k \geqslant 0)}{\square(n_1 + k) \rhd \square(n_1) \star \square(k)} \qquad \dfrac{\text{FRAME}\quad \{\square(n_1)\}\ c\ \{\lambda v.\ \square(n_2)\}}{\{\square(n_1) \star \square(k)\}\ c\ \{\lambda v.\ \square(n_2) \star \square(k)\}} \qquad \dfrac{\text{QUESTION 2.7}}{\square(n_2) \star \square(k) \rhd \square(n_2 + k)}}{\{\square(n_1 + k)\}\ c\ \{\lambda v.\ \square(n_2 + k)\}}$$

Question 2.10. *Write a program* `p` *such that for all n, $\{[n \geqslant 0]\}$ `p` n $\{\lambda\_.\ \square(n)\}$ and prove the triple.*

Answer. `p` can be `let rec p n = if n = 0 then () else ref 4; p (n - 1)`. Equivalently, we prove by induction on $n \in \mathbb{N}$ that $\{[]\}$ `p` n $\{\lambda\_.\ \square(n)\}$:

- if $n = 0$ then $\{[]\}$ () $\{\lambda\_.\square(0)\}$ which holds by the VAL rule since $[] \rhd (\lambda\_.\square(0))(()) = []$.
- otherwise, we know $\{[]\}$ `p` n $\{\square(n)\}$ (*) by induction, and we prove $\{[]\}$ `ref 4;p` n $\{\lambda\_.\ \square(n + 1)\}$. By the rule for `ref`, we have $\{[]\}$ `ref 4` $\{\lambda r.\ r \mapsto 4\}$. By the rule for sequence, it remains to prove that for all r, $\{r \mapsto 4\}$ `p` n $\{\lambda\_.\ \square(n + 1)\}$, which can be derived from (*) by the COMBINED rule by framing with $\square(1)$, since $r \mapsto 4 \rhd [] \star \square(1)$ and $\square(n) \star \square(1) \rhd \square(n + 1)$.

Question 2.11 (Bonus, open-ended). *Suppose that real life is catching up with separation logic and that the size of the heap is bounded by some fixed integer N. Why is the triple $\{[]\}$ `ref` v $\{\lambda r.\ r \mapsto v\}$ not valid anymore? Do you think that it is possible to change the precondition to make the triple valid (and usable)? What about changing the postcondition?*

Answer. Because using it $N+1$ times (or framing with $\square(N)$), we would obtain $\square(N+1)$, which means that the size of the heap is $> N$. However a precondition for `ref` seems impossible to do if we keep $\star$ and the frame rule, since it is always possible to complete the precondition with a heap that makes the combined heap of size N and hence the `ref` would fail. Changing the underlying heap and the definition of $\star$ by keeping track of available space is a possible solution. Changing the post-condition to allow `ref` to fail seems also possible, for example $\lambda r.\ r \mapsto v \curlyvee [r = \mathsf{null}]$.

3 Separation logic: back pointers

A doubly-linked list is a mutable list (or "linked list") with "back pointers". Similarly to linked list cells, the first field of a doubly-linked list cell is its content, and the second field is a pointer to the next cell (respectively named `data` and `next`). Doubly-linked list cells also have a third field `prev` pointing to the previous cell, which is sometimes called a back pointer. The type of cells can be written as follows:

```
type 'a dllcell = { mutable data : 'a;
                    mutable next : 'a dllcell;
                    mutable prev : 'a dllcell }  (* or null *)
```

The back pointer of the first cell is null (as is, as usual, the forward pointer of the last cell). For example, the pure list `[3; 4; 7; 13]` is represented in memory as a doubly-linked list using four cells at addresses p_1, p_2, p_3, p_4 as showed in Figure 1:

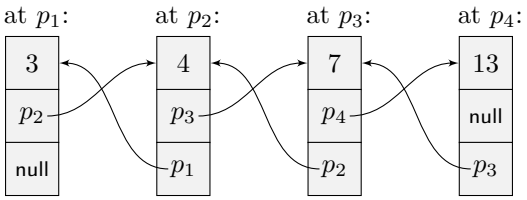


Figure 1: Doubly-linked list containing the list $[3;4;7;13]$

Question 3.1. *Let $n \in \mathbb{N}$. Given n values $a_1, \ldots, a_n$ and n memory addresses $p_1, \ldots, p_n$, write a separation logic formula that describes the fact that the list $[a_1; \ldots; a_n]$ is represented in memory as a doubly-linked list using the cells at exactly the addresses $p_1, \ldots, p_n$.*

Answer.

$$\mathop{\circledast}_{i=1}^{n} p_i \rightsquigarrow \{|a_i, p'_{i+1}, p'_{i-1}|\} \qquad \text{where} \qquad p'_i \equiv \begin{cases} \mathsf{null} & \text{if } i = 0 \text{ or } i = n+1 \\ p_i & \text{if } i \in \{1, \ldots, n\} \end{cases}$$

That formula is not perfect, since only the first and last pointers p_1 and p_n should be accessible to the programmer, while the others should be hidden behind existential quantifiers. We would like to define a representation predicate for a doubly-linked list containing the pure list l starting at a cell at address p and ending at a cell at address q, that we would write with the notation $p \overset{l}{\leftsquigarrow} q$.

Question 3.2. *Explain why it is difficult to define $p \overset{l}{\leftsquigarrow} q$ directly by recursion on l.*

Answer. If $l = x :: l'$, the part of the doubly-linked list that contains l' is not a doubly-linked list itself, since its first back pointer points to the first cell and hence is not null.

To solve this problem, we define the auxiliary predicate $(p, p') \overset{l}{\leftsquigarrow} (q, q')$ by induction on l, as follows:

$$\begin{array}{lll} (p,p') \overset{l}{\leftsquigarrow} (q,q') & \equiv & \mathsf{match}\ l\ \mathsf{with} \\ & & \mid \mathsf{nil} \Rightarrow [p = q \wedge p' = q'] \\ & & \mid x :: l' \Rightarrow \exists r.\, p \rightsquigarrow \{|\mathrm{data}{=}x;\ \mathrm{next}{=}r;\ \mathrm{prev}{=}p'|\} \star (r,p) \overset{l'}{\leftsquigarrow} (q,q') \end{array}$$

In the rest of this exercise, one can write $p \rightsquigarrow \{\!|x; r; p'|\!\}$ as a shortcut for $p \rightsquigarrow \{\!|\mathsf{data}{=}x;\ \mathsf{next}{=}r;\ \mathsf{prev}{=}p'|\!\}$.

Question 3.3. *Give a simpler separation logic formula for* $(p, p') \overset{[a]}{\leftrightsquigarrow} (q, q')$*, i.e. for lists of length 1.*

Answer. $\cdots \equiv (\exists r.\, p \rightsquigarrow \{\!|a; r; p'|\!\} \star (r, p) \overset{[]}{\leftrightsquigarrow} (q, q'))$
$\equiv (\exists r.\, p \rightsquigarrow \{\!|a; r; p'|\!\} \star [r = q \wedge p = q'])$
$\equiv (p \rightsquigarrow \{\!|a; q; p'|\!\} \star [p = q'])$

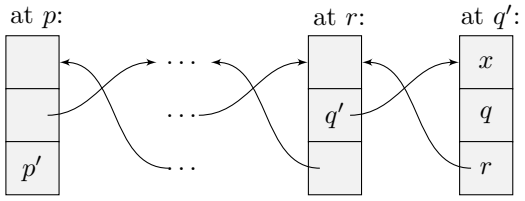
Question 3.4. *Suppose that* $(p, p') \overset{[3;4;7;13]}{\leftrightsquigarrow} (q, q')$ *describes the memory layout in the diagram of Figure 1, with the four cells at addresses* p_1, p_2, p_3, p_4*. What are* $p, p', q,$ *and* q'*? More generally, how to define* $p \overset{l}{\leftrightsquigarrow} q$ *in terms of* $(\cdot, \cdot) \overset{\cdot}{\leftrightsquigarrow} (\cdot, \cdot)$*?*[1]

Answer. $p = p_1$, $p' = \mathsf{null}$, $q' = p_4$, $q = \mathsf{null}$; and $p \overset{l}{\leftrightsquigarrow} q \equiv (p, \mathsf{null}) \overset{l}{\leftrightsquigarrow} (\mathsf{null}, q)$.

We recall that $l_1 +\!\!+ l_2$ is the concatenation of the pure lists l_1 and l_2, and that $l\&x$ is short for $l +\!\!+ [x]$.

Question 3.5. *Express* $(p, p') \overset{l\&x}{\leftrightsquigarrow} (q, q')$ *in terms of* $(\cdot, \cdot) \overset{l}{\leftrightsquigarrow} (\cdot, \cdot)$*. Draw a pointer diagram, similar to Figure 1, showing the last two cells. At which address is the cell containing* x*?*

Answer. $(p, p') \overset{l\&x}{\leftrightsquigarrow} (q, q') = \exists r.\, (p, p') \overset{l}{\leftrightsquigarrow} (q', r) \star q' \rightsquigarrow \{\!|x;\ q;\ r|\!\}$. x is in the cell at address q':



Doubly-linked lists make it easy to iterate backwards:

```
let rec iter_backwards f q =
  if q != null then (f q.data; iter_backwards f q.prev)
```

One specification for iteration on (non doubly-linked) mutable lists is Equation (1) below:

$$\forall f, p, I, l.\ \big(\forall x, k.\ \{I\ k\}\ (f\ x)\ \{\lambda\_.\ I\ (k\&x)\}\big) \Rightarrow \{p \rightsquigarrow \mathsf{Mlist}\ l \star I\ \mathsf{nil}\}\ (\texttt{miter}\ f\ p)\ \{\lambda\_.\ p \rightsquigarrow \mathsf{Mlist}\ l \star I\ l\} \quad (1)$$

Question 3.6. *Give a specification for* `iter_backwards` *that is similar in style to Equation* (1)*, and prove that the function satisfies its specification. State carefully your induction. (20 lines max)*

Answer.

$$\forall f, p, I, l.\ (\forall x, k.\{I\ k\}\ f x\ \{I(x :: k)\}) \Rightarrow \{p \overset{l}{\leftrightsquigarrow} q \star I\ \mathsf{nil}\}\ \texttt{iter\_backwards}\ f\ q\ \{\lambda\_. p \overset{l}{\leftrightsquigarrow} q \star I\ l\}$$

Let f, I such that $(\forall x, k.\{I\ k\}\ f x\ \{I(x :: k)\})$. We prove by induction on $n \in \mathbb{N}$ that for all l of length n, and for all p, q, q', k:

$$\{(p, \mathsf{null}) \overset{l}{\leftrightsquigarrow} (q', q) \star I\ k\}\ \texttt{iter\_backwards}\ f\ q\ \{(p, \mathsf{null}) \overset{l}{\leftrightsquigarrow} (q', q) \star I\ (l +\!\!+ k)\}$$

- When $n = 0$, $l = []$, the precondition *and* the postcondition are $[p = q' \wedge q = \mathsf{null}] \star I\ k$, which means the `if` returns the value `()`, which means we need to prove the the precondition entails the postcondition (applied to `()`), which holds by reflexivity of $\vartriangleright$.
- When $n > 0$, $l = l'\&x$ for some l' and x. Using the previous question, the precondition becomes, for some r,

  $$(p, \mathsf{null}) \overset{l'}{\leftrightsquigarrow} (q, r) \star q \rightsquigarrow \{\!|x; q'; r|\!\} \star I\ k$$

  the `if` enters its body, `q.data` evaluates to x so framing out everything but $I\ k$, the spec for f gives

  $$(p, \mathsf{null}) \overset{l'}{\leftrightsquigarrow} (q, r) \star q \rightsquigarrow \{\!|x; q'; r|\!\} \star I\ (x :: k)$$

[1] In case the notation with dots "$\cdot$" is unclear, you must give a formula containing $(r, r') \overset{l'}{\leftrightsquigarrow} (s, s')$ for some r, r', l', s, s'.

then `q.prev` evaluates to r, so framing out $q \rightsquigarrow \{\!|x; q'; r|\!\}$, the induction hypothesis gives:

$$(p, \mathsf{null}) \overset{l'}{\leftsquigarrow} (q, r) \star q \rightsquigarrow \{\!|x; q'; r|\!\} \star I\ (l' +\!\!+ x :: k)$$

which, using the previous question again, and rewriting inside I, implies our postcondition:

$$(p, \mathsf{null}) \overset{l'\&x}{\leftsquigarrow} (q', q) \star I\ ((l'\&x) +\!\!+ k)$$

We want to write a function `dllist_of_list` that, given a non-mutable (pure) list l, creates a (mutable) doubly-linked list containing l. We start by writing the following `allocate` function:

```
let rec allocate (l : 'a list) : 'a dllcell =
  match l with
  | [] -> null
  | x :: l -> { data = x; prev = null; next = allocate l }
```

Question 3.7. *What does* `allocate` *do? Write a usable specification for* `allocate` *(you may have to define a new representation predicate). (5 lines max)*

Answer. `allocate` creates and returns a linked list of doubly-linked list cells, with the back pointers all being null. This means it does *not* return a pointer to a valid doubly-linked list. One specification is:

$$\forall l.\ \{[]\}\ \texttt{allocate}\ l\ \{\lambda p.\ p \rightsquigarrow \mathsf{Mlist}'l\}$$

Where $p \rightsquigarrow \mathsf{Mlist}'\mathsf{nil} \equiv [p = \mathsf{null}]$ and $p \rightsquigarrow \mathsf{Mlist}'(x :: l) \equiv \exists r.\, p \rightsquigarrow \{\!|x; r; \mathsf{null}|\!\} \star r \rightsquigarrow \mathsf{Mlist}'l$. It is equally useful to replace $p \rightsquigarrow \{\!|x; r; \mathsf{null}|\!\}$ with $p \rightsquigarrow \{\!|x; r; \_|\!\}$.

Let us now define the `fill_in` function:

```
let rec fill_in (previous : 'a dllcell) (p : 'a dllcell) : 'a dllcell =
  if p = null then previous else
    (p.prev <- previous; fill_in p p.next)
```

Question 3.8. *Give a usable specification for* `fill_in` *and prove it correct. (15 lines max: state precisely your induction hypothesis, but try not to write too many other details)*

Answer.

$$\forall l, p.\ \{p \rightsquigarrow \mathsf{Mlist}'l\}\ \texttt{fill\_in}\ \mathsf{null}\ p\ \{\lambda q.\ p \overset{l}{\leftsquigarrow} q\}$$

We prove by induction on l that:

$$\forall r, p.\ \{p \rightsquigarrow \mathsf{Mlist}'l\}\ \texttt{fill\_in}\ r\ p\ \{\lambda q.\ (p, r) \overset{l}{\leftsquigarrow} (\mathsf{null}, q)\}$$

- if $l = []$, $p \rightsquigarrow \mathsf{Mlist}'l$ is $[p = \mathsf{null}]$, so the `if` returns r so we must prove $p \rightsquigarrow \mathsf{Mlist}'l = [p = \mathsf{null}]$ entails $(\texttt{<postcondition>})r = (p, r) \overset{[]}{\leftsquigarrow} (\mathsf{null}, r) = [p = \mathsf{null} \wedge r = r]$, which holds.
- if $l = x :: l'$, $p \rightsquigarrow \mathsf{Mlist}'(x :: l')$ unfolds to $p \rightsquigarrow \{\!|x; p'; \mathsf{null}|\!\} \star p' \rightsquigarrow \mathsf{Mlist}'l'$ for some p'. The `if` goes into the second branch, `p.prev` $\leftarrow$ `r` changes our heap to $p \rightsquigarrow \{\!|x; p'; r|\!\} \star p' \rightsquigarrow \mathsf{Mlist}'l'$, and since `p.next` evaluates to p', we can frame out $p \rightsquigarrow \{\!| \ldots |\!\}$ to apply our induction hypothesis to get, after `fill_in p p.next`, some q such that $p \rightsquigarrow \{\!|x; p'; r|\!\} \star (p', p) \overset{l'}{\leftsquigarrow} (\mathsf{null}, q)$. Since we return q, it is enough to prove that this entails $(p, r) \overset{l}{\leftsquigarrow} (\mathsf{null}, q)$ which holds by definition of the doubly-linked list predicate.

Question 3.9. *Using Questions 3.7 and 3.8, implement and specify the function* `dllist_of_list`*. Explain which separation logic rules you need to prove it correct.*

Answer. First we remark that the function must return two pointers; then it is easy to define:

```
let dllist_of_list l = let p = allocate l in (p, fill_in null p)
```

and to specify, with the triple: $\forall l.\ \{[]\}\ \texttt{allocate}\ l\ \{\lambda(p,q).\ p \overset{l}{\leftsquigarrow} q\}$ (it is also possible to return a pointer to a cell containing both p and q).
The proof is simple, we need to use the specifications for the two functions, and then combine them with the rule for `let`:

$$\text{LET}\ \frac{\{[]\}\ \texttt{allocate}\ l\{\lambda p.\ p \rightsquigarrow \mathsf{Mlist}' l\} \qquad \forall p.\ \{p \rightsquigarrow \mathsf{Mlist}' l\}\ \texttt{fill\_in null}\ p\ \{\lambda q.\ p \overset{l}{\leftsquigarrow} q\}}{\{[]\}\texttt{let p = allocate l in (p, fill\_in null p)}\{\lambda(p,q).\ p \overset{l}{\leftsquigarrow} q\}}$$

There is also the rewriting of `(p, fill_in null p)` into `let q = fill_in null p in (p, q)`, but that is only bonus.

In the rest of this exercise, suppose that any record predicate $p \rightsquigarrow \{|\text{field}_0{=}a_0; \ldots; \text{field}_{n-1}{=}v_{n-1};|\}$ is short for $(p.\text{field}_0 \mapsto v_0) \star \ldots \star (p.\text{field}_{n-1} \mapsto v_{n-1})$ and that the offset of a field is equal to its position, so that effectively $(p.\text{field}_i) \mapsto v_i$ is the same as $(p+i) \mapsto v_i$.

Question 3.10. *Prove that* $(p \overset{l}{\leftsquigarrow} q) \rhd (p \rightsquigarrow \mathsf{Mlist}\ l \star H)$ *for a formula* H *of your choosing.*

Answer. We choose $H = \square(|l|)$ but choosing $H = \mathsf{GC} = \exists H'.\, H'$ also works. We prove more generally by induction on l that for all p and p', $(p,p') \overset{l}{\leftsquigarrow} (\mathsf{null}, q) \rhd p \rightsquigarrow \mathsf{Mlist}\ l \star \square(|l|)$.

- $l = []$: the entailement becomes $[p = \mathsf{null} \wedge p' = q] \rhd [p = \mathsf{null}] \star \square(0)$, which holds.
- $l = x :: l'$, $(p,p') \overset{l}{\leftsquigarrow} (\mathsf{null}, q)$ is $\exists r.\, p \rightsquigarrow \{|x; r; p'|\} \star (r,p) \overset{l'}{\leftsquigarrow} (\mathsf{null}, q)$, so by the rule for the left introduction of existential quantifiers, we prove that for all r, $p \rightsquigarrow \{|x; r; p'|\} \star (r,p) \overset{l'}{\leftsquigarrow} (\mathsf{null}, q)$ entails $p \rightsquigarrow \mathsf{Mlist}\ l \star \square(|l|)$.

$$\begin{aligned}
& p \rightsquigarrow \{|x; r; p'|\} \star (r,p) \overset{l'}{\leftsquigarrow} (\mathsf{null}, q) \\
(\text{by I.H.+framing}) \quad \rhd\ & p \rightsquigarrow \{|x; r; p'|\} \star p' \rightsquigarrow \mathsf{Mlist} l' \star \square(|l'|) \\
=\ & p \rightsquigarrow \{|x; r|\} \star (p+2) \mapsto p' \star p \rightsquigarrow \mathsf{Mlist} l' \star \square(|l'|) \\
\rhd\ & p \rightsquigarrow \{|x; r|\} \star p' \rightsquigarrow \mathsf{Mlist} l' \star \square(1) \star \square(|l'|) \\
\rhd\ & p \rightsquigarrow \mathsf{Mlist}(x :: l') \star \square(1 + |l'|) \\
=\ & p \rightsquigarrow \mathsf{Mlist} l \star \square(|l|)
\end{aligned}$$

In the following, suppose that there is no typing enforcement, and that `p.hd` behaves as `p.data`, and that `p.tl` behaves as `p.next`, as those fields have the same respective offsets in the record. Then the iteration on mutable lists `miter p` would work not only on normal mutable lists, but also on doubly-linked lists. More precisely, we would have Equation (2):

$$\forall f, p, q, I, l.\ \big(\forall x, k.\ \{I\, k\}\ (f\, x)\ \{\lambda\_.\ I\, (k \& x)\}\big) \Rightarrow \{(p \overset{l}{\leftsquigarrow} q) \star I\, \mathsf{nil}\}\ (\texttt{miter}\, f\, p)\ \{\lambda\_.\ (p \overset{l}{\leftsquigarrow} q) \star I\, l\} \quad (2)$$

Question 3.11. *Could we use the heap entailment of Question 3.10 to derive Equation* (2) *from Equation* (1)*? Or from a variant of Equation* (1)*? (Hint: notice that* `miter` *does not modify the list.)*

Answer. Not using the traditional rules of separation logic, since we would need the reversed direction for entailment (which does not hold). There are at least two ways to circumvent this:

- with read-only permission, we can have $\{\mathsf{RO}(p \rightsquigarrow \mathsf{Mlist}\ l) \star I\ []\}\ \texttt{miter}\ p\ \{I\ l\}$, and so

$$\begin{array}{rl} \text{GC} & \dfrac{\{\mathsf{RO}(p \rightsquigarrow \mathsf{Mlist}\ l \star H) \star I\ []\}\ \texttt{miter}\ p\ \{I\ l\}}{} \\ \text{CONSEQ} & \dfrac{\{\mathsf{RO}(p \overset{l}{\leftsquigarrow} q) \star I\ []\}\ \texttt{miter}\ p\ \{I\ l\}}{} \\ \text{FRAME-RO} & \dfrac{}{\{p \overset{l}{\leftsquigarrow} q \star I\ []\}\ \texttt{miter}\ p\ \{p \overset{l}{\leftsquigarrow} q \star I\ l\}} \end{array}$$

- with fractional permissions, we can establish a two-way entailment (i.e. an equality): $(p \rightsquigarrow q) = D \star L$ where D is $p \overset{l}{\leftsquigarrow}_{\frac{1}{2},\frac{1}{2},1} q$ (i.e. all the first and second fields have permission $\frac{1}{2}$, all third fields have full 1 permission) and L is $p \rightsquigarrow_{\frac{1}{2},\frac{1}{2}} \mathsf{Mlist}\ l$ (both fields have $\frac{1}{2}$ permission). We can adapt the specification of `miter` to $\{L \star I\ []\}\ \texttt{miter}\ p\ \{L \star I\ l\}$, from which, framing with D, we can deduce our desired triple.

We are now interested in n-ary trees (i.e. trees that are not necessarily binary trees) and such that each node has pointers not only to its children, but also to its parent (a back pointer) and to its left and right siblings. We call those trees pointing trees. Pointing trees are mutable, and represent pure n-ary trees. We recall the inductive definition of pure n-ary trees:

```
Inductive tree (A : Type) : Type :=
 | Leaf : tree A
 | Node : A → list (tree A) → tree A.
```

Question 3.12 (Hard). *Can you think of a way to reuse exactly doubly-linked lists to approximate the features of pointing trees in memory? Draw a diagram on a simple example to illustrate your answer. (A partial answer, and an argumentation on whether it works or not, can count.)*

Answer. It is possible to have doubly-linked list cells of which the value points to nodes, and to define a higher-order version of doubly-linked lists written e.g. $(p, q) \rightsquigarrow \mathsf{dllistOf}\ R\ l$ and then define $p \rightsquigarrow \mathsf{PTree}\ \mathsf{T}$ as $p \rightsquigarrow \mathsf{PTree'}_{\mathsf{null}}\ \mathsf{T}$ where the auxiliary predicate $\mathsf{PTree'}_r$ also has a back pointer r for the parent, and the definition for non-leaves is $p \rightsquigarrow \mathsf{PTree'}_r(\mathsf{Node}(x, l)) \equiv \exists p' q'.\, p \rightsquigarrow \{\!|x; p'; r|\!\} \star (p', q') \rightsquigarrow \mathsf{dllistOf}\ (\mathsf{PTree'}_p)\ l$. However the nodes themselves cannot go left or right, only up or down: only the cells (pointing to the nodes) can point left or right. It is also possible to make the nodes point back to the cell, so that they can point left or right.

Question 3.13 (Hard). *Using the insights you gained working on doubly-linked lists, but not using them directly, represent pointing trees in memory the simplest way you can. This means: 1) defining a type of pointing tree nodes* `'a pnode`*, 2) defining a representation predicate $p \rightsquigarrow \mathsf{PTree}\ T$ where T is a pure n-ary tree, and 3) giving specifications for the navigation pointers (up, down, left, right). You will probably need to use one or two auxiliary representation predicates.*

Answer. The simplest way is to have directly the pointers in a record:

```
type 'a pnode = { data : 'a;
                  up   : mutable 'a pnode;
                  down : mutable 'a pnode;
                  left : mutable 'a pnode;
                  right: mutable 'a pnode }   (* or null *)
```

We have only one type of node, but since we still have a mix of list and trees we need to tree in the logic, we still need to distinguish them and define an auxiliary predicate for lists.

$$
\begin{aligned}
p \rightsquigarrow \mathsf{PTree}_{p_u,p_l,p_r}(\mathsf{Leaf}) &\equiv [p = \mathsf{null}] \\
p \rightsquigarrow \mathsf{PTree}_{p_u,p_l,p_r}(\mathsf{Node}(x, ts)) &\equiv \exists p'.\, p \rightsquigarrow \{\!|\mathrm{data}{=}x; \mathrm{up}{=}p_u; \mathrm{down}{=}p'; \mathrm{left}{=}p_l; \mathrm{right}{=}p_r|\!\}] \\
p \rightsquigarrow \mathsf{PTreeList}_{p_u,p_l}(\mathsf{nil}) &\equiv [p = \mathsf{null}] \\
p \rightsquigarrow \mathsf{PTreeList}_{p_u,p_l}(\mathsf{t} :: \mathsf{ts}) &\equiv \exists p'.\, p \rightsquigarrow \mathsf{PTree}_{p_u,p_l,p'} \star p' \rightsquigarrow \mathsf{PTreeList}_{p_u,p}(ts)
\end{aligned}
$$

Then $p \rightsquigarrow \mathsf{PTree}\ t$ is simply $p \rightsquigarrow \mathsf{PTree}_{\mathsf{null},\mathsf{null},\mathsf{null}}$. Navigating in a pointing tree is easy: follow a pointer in the direction you want, which is possible iff it is not null. The specification for navigation needs also some thinking, but it can be done using a tree path, i.e. a list l of indices of children, and defining yet another refinement $p@l \rightsquigarrow \mathsf{PTree}\ t$, which states that l is a valid path in t which is (entirely) represented in memory and that q is the address of the corresponding node. The navigation specs look like:

$$
\begin{aligned}
\{p@(l\&i) \rightsquigarrow \mathsf{PTree}\ t\}\ p\texttt{.up}\ \{\lambda r. r@l \rightsquigarrow \mathsf{PTree}\ t\} \\
\{p@l \rightsquigarrow \mathsf{PTree}\ t \star [l\&0 \in \mathsf{dom(t)}]\}\ p\texttt{.down}\ \{\lambda r. r@(l\&0) \rightsquigarrow \mathsf{PTree}\ t\} \\
\{p@(l\&i) \rightsquigarrow \mathsf{PTree}\ t \star [i \geqslant 1]\}\ p\texttt{.left}\ \{\lambda r. r@(l\&(i-1)) \rightsquigarrow \mathsf{PTree}\ t\} \\
\{p@(l\&i) \rightsquigarrow \mathsf{PTree}\ t \star [l\&(i+1) \in \mathsf{dom(t)}]\}\ p\texttt{.right}\ \{\lambda r. r@(l\&(i+1)) \rightsquigarrow \mathsf{PTree}\ t\}
\end{aligned}
$$

For invalid paths the post condition is $\lambda r.[r = \mathsf{null}] \star$ `<precondition>`.
