Final exam, March 10, 2021

- There are 4 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:

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

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.*

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.*

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

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

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

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

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.*

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* $|m| = 2$.

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* $|m| = 1$.

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.*

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

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

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

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

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

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

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

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

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

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?*

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:

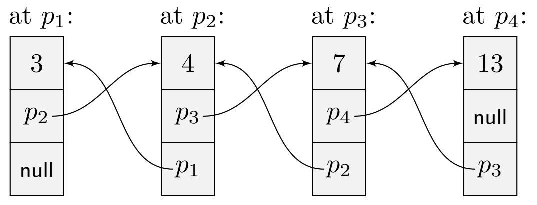

Figure 1: Doubly-linked list containing the list $[3; 4; 7; 13]$

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 $p_1, \ldots, p_n$.*

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 $p \overset{l}{\leftrightsquigarrow} q$.

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

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

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

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

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

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]

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 $l +\!\!+ [x]$.

[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'.

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?*

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:

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

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)*

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)*

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)*

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.*

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

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):

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

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.)*

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.)*

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.*