Final exam, March 3, 2020

- 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 5 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.
- **Please write your answers for Exercise 1 on a separate piece of paper.**

1 Minimum Excluded

In this exercise, we consider the problem of *minimum excluded* which consists, given an array a of integers, in finding the smallest non-negative integer that does not occur anywhere in a. For example, the minimum excluded of array $[45; -3; 1; -98; 6; 0; 42]$ is 2. We are given the following two logic predicates on arrays:

```
mem (x: int) (a: array int) = ∃ i. 0 ⩽ i < a.length ∧ a[i] = x

all_occur_below (m:int) (a:array int) = ∀ x. 0 ⩽ x < m → mem x a
```

The following program checks whether the given x occurs in the array a.

```
let check_occur_simple (x:int) (a:array int) : bool
  ensures { result ↔ mem x a }
= let ref res = False in
  for i=0 to a.length - 1 do
    if a[i] = x then res ← True;
  done;
  res
```

Question 1.1. *Propose a loop invariant that allows proving this program correct. Justify your answer in 2-3 lines of text.*

Digression: Programs with `return` Statements

The code of `check_occur_simple` is inelegant because it goes to the end of the loop instead of exiting as soon as x is found. A better solution would be to use a `return` statement that would act as a kind of exception that is implicitly captured at the end of the function body:

```
let check_occur (x:int) (a:array int) : bool
  ensures { result ↔ mem x a }
= for i=0 to a.length - 1 do
    if a[i] = x then return True;
  done;
  False
```

We assume our programming language is initially *not* equipped with an exception mechanism, and equipped with a weakest pre-condition calculus $\mathrm{WP}(e, Q)$ for any program expression e and formula Q. We want to extend it to support such a `return` statement, with syntax "`return` e" for any expression e. We assume a typing system that constrains e to have the return type of the enclosing function. We extend the weakest precondition calculus to $\mathrm{WP}(e, Q, R)$ where R is now the post-condition in case of a `return` statement is met. Both Q and R may contain the keyword `result` to respectively denote the returned value.

Question 1.2. *Complete the following rules for computing* WP*, where t is a pure term.*

$$\begin{array}{rl} (assignment) & \mathrm{WP}(x \leftarrow t, Q, R) = \\ (assert) & \mathrm{WP}(\texttt{assert } A, Q, R) = \\ (let) & \mathrm{WP}(\texttt{let } x\texttt{=}e_1 \texttt{ in } e_2, Q, R) = \\ (return) & \mathrm{WP}(\texttt{return } t, Q, R) = \\ (if) & \mathrm{WP}(\texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2, Q, R) = \\ (function\ call) & \mathrm{WP}(f(\vec{t_i}), Q, R) = \end{array}$$

Consider now the general shape of a program function

```
let f(x_1 : τ_1, ..., x_n : τ_n) : τ
  requires Pre
  writes w⃗
  ensures Post
= body
```

Question 1.3. *What is the formula one should prove to express safety of the execution of any calls to f?*

Question 1.4. *Considering again the program* `check_occur` *that uses a* `return` *statement, propose a loop invariant that allows to prove this program correct. Justify your answer in 2-3 lines of text.*

Back to the Minimum Excluded Problem

A first remark to notice is that the minimum excluded of an array of length l is necessary between 0 and l included.

Question 1.5. *Justify the remark above informally, with 2-3 lines of text.*

Any program that solves the minimum excluded problem can thus be specified by the following contract:

```
val min_excluded (a: array int) : int
  ensures { 0 ⩽ result ⩽ a.length }
  ensures { not (mem result a) }
  ensures { all_occur_below result a }
```

A naive algorithm is to iteratively check all values starting from 0, calling the previous program `check_occur`:

```
for min = 0 to a.length - 1 do
  if not (check_occur min a) then return min
done;
a.length
```

Question 1.6. *Propose a loop invariant that allows proving this program correct. Justify your answer in a few lines of text.*

To avoid the quadratic complexity of the naive algorithm above, we propose the algorithm below that makes use of an extra array of booleans:

```
let l = a.length in
let used = Array.make l false in
for i = 0 to l - 1 do
  let x = a[i] in if 0 ⩽ x < l then used[x] ← true
done;
for r = 0 to l - 1 do
  if not used[r] then return r;
done;
l
```

Question 1.7. *Propose proper loop invariants for the algorithm above. Justify your answer in a few lines of text.*

2 Heap entailment

Reminders Heaps are finite maps from locations to values, *i.e.* sets of pairs (l, v) where l is unique in each set. Separating conjunction $\star$ is defined by

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

where $m_1 \perp m_2$ means that the domains (the l's) of m_1 and m_2 are disjoint. We also recall the definition of regular conjunction $\curlywedge$, regular disjunction $\curlyvee$, single heap $\mapsto$, pure fact $[P]$, and heap entailment $\rhd$:

$$\begin{array}{rcl} H_1 \curlywedge H_2 & \equiv & \lambda m.\ H_1\, m \wedge H_2\, m \\ H_1 \curlyvee H_2 & \equiv & \lambda m.\ H_1\, m \vee H_2\, m \\ l \mapsto v & \equiv & \lambda m.\ m = \{(l, v)\} \wedge l \neq \mathsf{null} \\ [P] & \equiv & \lambda m.\ m = \varnothing \wedge P \\ H_1 \rhd H_2 & \equiv & \forall m.\, H_1\, m \Rightarrow H_2\, m \end{array}$$

Question 2.1. *For each heap entailment below, explain why it holds, or provide a counterexample if it does not (the counterexample should be a heap satisfying the left-hand side and not the right-hand side).*

1. $((x \mapsto 2) \star (y \mapsto 4)) \rhd [\mathsf{False}]$
2. $((x \mapsto 2) \star (x \mapsto 4)) \rhd [\mathsf{False}]$
3. $((x \mapsto 2) \star (x \mapsto 2)) \rhd [\mathsf{False}]$
4. $((x \mapsto 2) \curlywedge (y \mapsto 4)) \rhd [\mathsf{False}]$
5. $((x \mapsto 2) \curlywedge (y \mapsto 2)) \rhd [\mathsf{False}]$
6. $((x \mapsto 2) \curlywedge (x \mapsto 4)) \rhd [\mathsf{False}]$
7. $((x \mapsto 2) \curlywedge (x \mapsto 2)) \rhd [\mathsf{False}]$
8. $([x = 2] \star [x = 4]) \rhd [\mathsf{False}]$
9. $([x = 2] \star [x = 2]) \rhd [\mathsf{False}]$
10. $([x = 2] \curlywedge [x = 4]) \rhd [\mathsf{False}]$
11. $([x = 2] \curlywedge [x = 2]) \rhd [\mathsf{False}]$

Question 2.2. *Same question for a second set of entailments (remember that $\exists x.H \ \equiv \ \lambda m. \exists x.(H\ m)$)*

1. $(\exists x.([x = 2] \star [x = 2])) \rhd [\mathsf{False}]$
2. $(\exists x.([x = 2] \star [x = 4])) \rhd [\mathsf{False}]$
3. $(\exists x.((x \mapsto 2) \star (x \mapsto 2))) \rhd [\mathsf{False}]$
4. $(\exists x.((x \mapsto 2) \star (x \mapsto 4))) \rhd [\mathsf{False}]$
5. $((\exists x.x \mapsto 2) \star (\exists x.x \mapsto 2)) \rhd [\mathsf{False}]$
6. $((\exists x.x \mapsto 2) \star (\exists x.x \mapsto 4)) \rhd [\mathsf{False}]$
7. $((\exists x.x \mapsto 2) \curlywedge (\exists x.x \mapsto 2)) \rhd [\mathsf{False}]$
8. $((\exists x.x \mapsto 2) \curlywedge (\exists x.x \mapsto 4)) \rhd [\mathsf{False}]$

3 Ropes

Notations In the following, we omit field names in records. For example the notation $p \rightsquigarrow \{\!|x; p'|\!\}$ is short for $p \rightsquigarrow \{\!|\mathrm{hd}{=}x; \mathrm{tl}{=}p'|\!\}$.

There are many text editors so let us try to make yet another one. A fast one. We start with the problem of representing text in memory and will not consider any other problem. Text is often represented using strings. In this exercise, strings are arrays of characters with no additional information such as length (in this sense, they are arrays as in the C language). More precisely, we define the following representation predicate, where p is a pointer and s is a pure list of characters, $|s|$ is its length, s_i is the ith element of the list, and $\circledast$ is the iteration of the separating conjunction ($*$):

$$p \rightsquigarrow \mathsf{String}(s) \quad \equiv \quad \circledast_{i=0}^{|s|-1} (p+i) \mapsto s_i$$

Question 3.1. *Let ε be the empty string, of length 0. Give a simplified form of $p \rightsquigarrow \mathsf{String}(\varepsilon)$.*

We write `string` for the type corresponding to those strings, which is in fact a pointer to the first character. It is possible to define a function `substring : string → int → int → string` that takes a string representing s, and two integers i and n and returns the string representing $s_{i\ldots i+n}$, where $s_{i\ldots j}$ denotes the string composed of the characters $s_i, s_{i+1}, \ldots, s_{j-1}$. Note that $s_{i\ldots j}$ *excludes* s_j, and that $s_{0\ldots|s|} = s$.
In other words, `substring` has the following (incomplete) specification:

$$\forall s\ p\ i\ n \quad \ldots \ \Rightarrow\ \{p \rightsquigarrow \mathsf{String}(s)\}(\texttt{substring p i n})\{\lambda p'. p \rightsquigarrow \mathsf{String}(s) * p' \rightsquigarrow \mathsf{String}(s_{i\ldots i+n})\}$$

Question 3.2. *Give a reasonable precondition to complete the specification. Explain carefully how the case $p = \mathsf{null}$ is handled.*

Question 3.3. *How much time and memory, at least, is using* `substring p i n`*, whatever the implementation?*

For some purposes, this is just too much for our text editor. Luckily there are much faster operations. The function `split_string` returns a pointer such that:

$$\forall p\ s\ i \quad 0 \leqslant i \leqslant |s| \ \Rightarrow\ \{p \rightsquigarrow \mathsf{String}(s)\}(\texttt{split\_string i p})\{\lambda p'. p \rightsquigarrow \mathsf{String}(s_{0\ldots i}) * p' \rightsquigarrow \mathsf{String}(s_{i\ldots|s|})\}$$

Question 3.4. *Give a constant-time implementation of* `split_string`*.*

Question 3.5. *Show, with details, that* `split_string` *satisfies its specification. (less than 10 lines expected)*

Even if `split_string` is fast, our text editor should be able to do other operations on text, such as inserting characters. Many data structures support this operation alongside many others, but converting a very long string into almost any of those data structures is slow. We need one that is more robust than strings but can made from strings easily, such as a *rope*.
A rope is a four-value record with fields `l`, `r`, `n`, and `m`. One can define such a record in a program with `let p = { l = v0; r = v1; n = v2; m = v3 }` and we would then write $p \rightsquigarrow \{\!|v_0; v_1; v_2; v_3|\!\}$ for the resulting heap predicate, which is formally defined as $(p \mapsto v_0) \star (p+1 \mapsto v_1) \star (p+2 \mapsto v_2) \star (p+3 \mapsto v_3)$. We access those four values from a pointer `p` with the respective field access notations `p.l`, `p.r`, `p.n`, and `p.m`. Ropes are represented in memory in such a way that if `p` points to a rope that represents a string s, then either:

(1)
- `p.l` points to a rope that represents some string s_1,
- `p.r` points to a separate rope that represents some string s_2,
- `p.n` is the integer $|s|$,
- `p.m` is the integer $|s_1|$, and
- $s = s_1 +\!\!+ s_2$;

or

(2)
- `p.l` is null,
- `p.r` is a pointer to the string s,
- `p.n` is the integer $|s|$, and
- `p.m` is some unspecified value.

Let us write Rope for some representation predicate for ropes, such that $\mathsf{Rope}(s)$ represents a string s as described above.

Question 3.6. *Define a function* `concat_rope` *of type* `rope → rope → rope` *that satisfies the following specification (no proof is necessary).*

$$\forall p\ p'\ s\ s',\ \{p \rightsquigarrow \mathsf{Rope}(\mathsf{s}) \star p' \rightsquigarrow \mathsf{Rope}(\mathsf{s}')\}\ (\texttt{concat\_rope p p'})\ \{\lambda r.r \rightsquigarrow \mathsf{Rope}(\mathsf{s} +\!\!+ \mathsf{s}')\}$$

Question 3.7. *Give an equation that corresponds to the cases (1) and (2), of the form* $p \rightsquigarrow \mathsf{Rope}(s) = \ldots$.

Question 3.8. *Can this equation be a recursive definition of the representation predicate* Rope*? Be precise on why or why not. If not, how can this be fixed?*

The higher-order representation predicate Quadof, for quadruples (or rope records), is defined as follows:

$$\begin{aligned} p \rightsquigarrow \mathsf{Quadof}\ R_1\ V_1\ R_2\ V_2\ R_3\ V_3\ R_4\ V_4 \equiv \\ \exists v_1 v_2 v_3 v_4.\ p \rightsquigarrow \{\!|v_1; v_2; v_3; v_4|\!\} \ \star\ v_1 \rightsquigarrow R_1\ V_1\ \star\ v_2 \rightsquigarrow R_2\ V_2 \\ \star\ v_3 \rightsquigarrow R_3\ V_3\ \star\ v_4 \rightsquigarrow R_4\ V_4 \end{aligned}$$

Question 3.9. *Rewrite the previous equation using* Quadof *instead of the* $\cdot \rightsquigarrow \{\!|\cdot|\!\}$ *notation.*

Question 3.10. *The following triple is missing a postcondition. Give one that relates the return value and* V_1*. It must also entail the precondition.*

$$\{p \rightsquigarrow \mathsf{Quadof}\ R_1\ V_1\ R_2\ V_2\ R_3\ V_3\ R_4\ V_4\}\ (\texttt{p.l})\ \{\quad \ldots \quad\}$$

We define the function `split_rope` of type `int → rope → rope * rope` as follows:

```
let rec split_rope (i : int) (p : rope) : rope * rope =
  if p.l == null then
    ({l = null; r = p.r     ; n = i       ; m = 2019},
     {l = null; r = p.r + i; n = p.n - i; m = 2020})
  else
     if i < p.m then
       let (l1, l2) = split_rope i p.l in
       (l1, {l = l2; r = p.r; n = p.n - i; m = p.m - i})
     else
       let (r1, r2) = split_rope (i - p.m) p.r in
       ({l = p.l; r = r1; n = i; m = p.m}, r2)
```

Question 3.11. *Give a specification for* `split_rope`*.*

Question 3.12. *Prove that* `split_rope` *satisfies its specification, but omit the* `then` *case of the first* `if` *and the* `else` *case of the second* `if`*. (You should not, however, omit to state your induction hypothesis and which rules you are applying.) (approx. 15 lines expected)*

Question 3.13. *How to define a variant of* Quadof *that takes only four arguments instead of eight? Can this reasoning be applied to the higher-order representation predicate* Mlistof*? For* Mcellof*?*

We want to add the `cut` and `paste` features to our editor, that we would implement with functions that have following specifications:

$$\{p \rightsquigarrow \mathsf{Rope}(s)\}\ (\texttt{cut p i j})\ \{\lambda x.x \rightsquigarrow \mathsf{Rope}(s_{i\ldots j}) \star p \rightsquigarrow \mathsf{Rope}(s_{0\ldots i} +\!\!+ s_{j\ldots|s|})\}$$

$$\{p \rightsquigarrow \mathsf{Rope}(s) \star x \rightsquigarrow \mathsf{Rope}(s')\}\ (\texttt{paste x p i})\ \{\lambda\_.p \rightsquigarrow \mathsf{Rope}(s_{0\ldots i} +\!\!+ s' +\!\!+ s_{i\ldots|s|})\}$$

Question 3.14. *Can we implement* `cut` *and* `paste` *with* `concat_rope` *and* `split_rope` *directly so that they satisfy these specifications? If not, how can we implement them?*

Question 3.15. *Why are the above specifications for* `cut` *and* `paste` *not enough to implement the usual features of a cut and paste system? What can we change to correct this? Give some advantages and disadvantages to your solution(s).*