Final exam, March 12, 2025

- 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 **2** parts. Part 1 correspond to the first half of the course, Part 2 about the second. Section 2.1 should be done before §2.2, §2.3, and §2.4, which are independent.
- Part 1 is worth approximately **one third** of the points, Part 2 approximately **two thirds**.
- Number each piece of paper sequentially (e.g. 1/3, 2/3, 3/3) to keep track of all documents (you do not need to label each page, just each paper, each "copie double").
- Write your name on each piece of paper.
- **Please write your answers for Part 1 and Part 2 on separate pieces of paper.**

1 Deductive verification with Why3

1.1 Boolean results

The following function computes the index of the smallest element in a table.

```
let find_min (a: array int): int
  ensures { 0 ≤ result < length a }
  ensures { ∀ k: int. 0 ≤ k < length a → a[result] ≤ a[k] }
= let ref r = 0 in
  let ref i = length a in
  while i > 0 do
    invariant (* 1 *) { 0 ≤ r < length a }
    invariant (* 2 *) { 0 ≤ i ≤ length a }
    invariant (* 3 *) { ∀ k: int. i < k < length a → a[r] ≤ a[k] }
    variant   { i }
    i := i − 1;
    if a[i] < a[r] then r := i;
  done;
  r
```

Question 1.1. *For each of the following affirmations, determine if they are true or false.*

1. *The invariant 1 is true at the start of the loop*
2. *The invariant 2 is true at the start of the loop*
3. *The invariant 3 is true at the start of the loop*
4. *The invariant 1 is preserved by the loop*
5. *The invariant 2 is preserved by the loop*
6. *The invariant 3 is preserved by the loop*
7. *The given invariants are sufficient for proving the post-condition of the function* `find_min`*.*
8. *The variant is sufficient for showing the termination of the loop.*

1.2 ♪ As invariant goes by ♪

Type invariants are used to enforce some internal property in a data structure.

For example, rational numbers are often kept in irreducible form:

```
predicate prime_to_one_another (a b:int) (* = (gcd a b = 1) *)

type q = { num : int; den: int }
    invariant { prime_to_one_another num den }
    by { num = 1; den = 1 }
```

The by-clause is used to give a witness of the type invariant. A witness is an example of a record that verifies the type invariant. A verification condition is generated to check that it is the case.

Why3 compiles the type `q` in first order logic using an abstract type:

```
type q' = { num : int; den: int }

type q

function open_q (x:q) : q'

axiom inv_q : ∀ x:q. prime_to_one_another (open_q x).num (open_q x).den

goal ex_q : prime_to_one_another 1 1
```

Question 1.2. *Why is it important to check that there exists an example of the record that verifies the type invariant?*

Question 1.3. *Why is an abstract type needed? Why* `q` *can't have the definition given to* `q'` *?*

The by-clause is not restricted to be a constant record, it is a program expression. Let see why it is needed with an example of a trivial data structure that contains fresh ids.

```
val ref counter : int

type t

function id t : int

val new_id () : t
```

Question 1.4. *Give a contract for* `new_id` *such that it returns each time a value* `t` *with a fresh* `id` *using the mutable global variable* `counter`*.*

Question 1.5. *Define a record* `p` *with two fields* `a` *and* `b` *of type* `t`*, and which verifies that the* `id` *of the value in the field* `a` *is strictly smaller than the one in the field* `b`*.*

Question 1.6. *Consider a record type* `t` *with type invariant* `Inv`*, and a* `by`*-clause* `By`*.* `Inv` *and* `By` *are typed in an environment where* `t` *is a simple record type without type invariant. How do we compute the formula to check that* `By` *is a witness of the invariant?*

We want to implement the type `t` and the function `new_id` efficiently:

Question 1.7. *Give an implementation of* `t` *and the function* `new_id`*, using the type* `int`*.*

Please now use a piece of paper separate from the one you used for Part 1

2 Separation Logic

We recall a few definition of S.L. connectives, where P and Q are heap predicates and P_0 is a proposition:

$$P \mathbin{\mathbb{\wedge}} Q \;\equiv\; \lambda m.\, P\,m \wedge Q\,m \qquad P \mathbin{\mathbb{\vee}} Q \;\equiv\; \lambda m.\, P\,m \vee Q\,m \qquad l \mapsto v \;\equiv\; \lambda m.\, m = \{(l, v)\} \wedge l \neq \mathsf{null}$$

$$\ulcorner P_0 \urcorner \;\equiv\; \lambda m.\, m = \emptyset \wedge P_0 \qquad P * Q \;\equiv\; \lambda m.\, \exists m_1 m_2.\, m = m_1 \uplus m_2 \wedge P\,m_1 \wedge Q\,m_2 \qquad \mathsf{GC} \;\equiv\; \lambda m.\, \mathsf{True}$$

$$\exists x.\, P \;\equiv\; \lambda m. \exists x. P m \qquad P \mathrel{-\!\!*} Q \;\equiv\; \lambda m.\, \forall m_1\ (m_1 \perp m \wedge P\,m_1) \Rightarrow Q(m_1 \uplus m)$$

Recall also that $P \triangleright Q$ means $\forall m.\ Pm \Rightarrow Qm$, and that $(P \triangleright Q) \wedge (Q \triangleright P) \Rightarrow P = Q$. You are allowed to use the extension of Separation Logic that includes fractional permissions $- \overset{\alpha}{\mapsto} -$ for any real number $\alpha \in]0, 1]$ together with equations and rules below, when $0 < \alpha$, $0 < \beta$ and $\alpha + \beta \leq 1$:

$$l \mapsto v = l \overset{1}{\mapsto} v \qquad\qquad l \overset{\alpha+\beta}{\mapsto} v = l \overset{\alpha}{\mapsto} v * l \overset{\beta}{\mapsto} v$$

$$\frac{}{l \overset{\alpha}{\mapsto} v * l \overset{\beta}{\mapsto} w \;\triangleright\; l \overset{\alpha+\beta}{\mapsto} v * \ulcorner v = w \urcorner}\ \text{FRAC-AGREE} \qquad\qquad \frac{}{\{l \overset{\alpha}{\mapsto} v\}\ !l\ \{\lambda r. \ulcorner r = v \urcorner * l \overset{\alpha}{\mapsto} v\}}\ \text{FRAC-READ}$$

2.1 Preliminaries

Question 2.1. *Give two examples of a P such that $(1 \mapsto 1 * 2 \mapsto 2) = (P * P)$.*

Question 2.2. *Show that $\ulcorner \urcorner \triangleright P \mathrel{-\!\!*} P$ and that $P * (Q \mathrel{-\!\!*} R) \triangleright Q \mathrel{-\!\!*} P * R$.*

Question 2.3. *Derive a contradiction from the following rule:*

$$\frac{\{P\}\ c\ \{\lambda\_.\ Q\}}{\{P \mathbin{\mathbb{\wedge}} R\}\ c\ \{\lambda\_.\ Q \mathbin{\mathbb{\wedge}} R\}}\ \text{AND-FRAME-BAD}$$

*(Do not explain **why** it does not hold. The goal is to give a proof of false by using this rule. Be precise about the rules that you are using. Approx. 7 lines should suffice)*

Question 2.4. *Which of the following rules hold? Explain why. (Approx. 6 lines)*

$$\frac{\{P\}\ c\ \{\lambda\_.\ Q\}}{\{P \mathbin{\mathbb{\vee}} R\}\ c\ \{\lambda\_.\ Q \mathbin{\mathbb{\vee}} R\}}\ \text{R1} \qquad \frac{\{P\}\ c\ \{\lambda\_.\ Q\}}{\{R \mathrel{-\!\!*} P\}\ c\ \{\lambda\_.\ R \mathrel{-\!\!*} Q\}}\ \text{R2} \qquad \frac{\{P\}\ c\ \{\lambda\_.\ Q\}}{\{P \mathrel{-\!\!*} R\}\ c\ \{\lambda\_.\ Q \mathrel{-\!\!*} R\}}\ \text{R3}$$

2.2 Treeness

Let $\mathsf{tree}(p)$ be the heap predicate $\exists T.\ p \rightsquigarrow \mathsf{Mtree}\ T$. It states that p points to *some* tree, not specifying which one, just that the pointer structure makes up a valid tree at p. It is useful to prove the safety of programs that manipulate mutable trees without establishing full functional correctness. In particular we are interested in *treeness-preserving* functions, i.e. functions `f` on trees such that:

$$\mathsf{TP}(\texttt{f}) \quad\equiv\quad \forall p\ \{\mathsf{tree}(p)\}\ \texttt{f}\ p\ \{\lambda\_.\ \mathsf{tree}(p)\} \tag{1}$$

Suppose we have one such function `action : α node → unit` such that TP(`action`) that performs some unknown operation on trees, potentially modifying them.

Consider now the function `find` that finds a subtree according to the result of a function `f`, itself treeness-preserving. The function `find_and_act` simply applies `action` at that subtree.

```
let rec find (f : 'a node -> int) (p : 'a node) : 'a node =
  let n = f p in
  if n = 0 || p = null then
    p
  else
    let q = if n < 0 then p.left else p.right in
    find f q

let find_and_act f p = let q = find f p in action q
```

We want to establish that `find_and_act f` is itself treeness-preserving when `f` is, i.e.:

$$\forall f \quad \mathsf{TP}(f) \quad\Rightarrow\quad \mathsf{TP}(\texttt{find\_and\_act}\ f) \tag{2}$$

Question 2.5. *Give a specification for `find` that is sufficient to establish (2).*

Question 2.6. *Prove that `find` satisfies this specification.*

2.3 Sharing and copying

Recall from class the following `copy` function on trees, slightly rewritten with some `let`-expansions to make the proof steps more explicit:

```
let rec copy (p:node) : node =
  if p == null then null else
  let q1 = p.left in
  let p1' = copy q1 in
  let q2 = p.right in
  let p2' = copy q2 in
  let x = p.item in
  { item = x; left = p1'; right = p2' }
```

We have seen in class that `copy` function satisfies specification (3) below. But this is not the most general specification.

$$\forall pT\ \{p \rightsquigarrow \mathsf{Mtree}\ T\}\ \texttt{copy p}\ \{\lambda q.\ p \rightsquigarrow \mathsf{Mtree}\ T * q \rightsquigarrow \mathsf{Mtree}\ T\} \tag{3}$$

Indeed, `copy` also works when there is some amount of sharing in the source tree. In this case, the source "tree" does not have a proper tree structure in memory, but rather a form of Directed Acyclic Graph (DAG). In order to reflect that, let us define a new connective $\circledast$ that we call here *partially separating conjunction*, and an inductive definition of $p \rightsquigarrow \mathsf{Dag}\ T$, as follows:

$$\begin{aligned} P \circledast Q &\equiv (P * \mathsf{GC}) \mathbin{\mathbb{\wedge}} (Q * \mathsf{GC}) \\ p \rightsquigarrow \mathsf{Dag\ Leaf} &\equiv \ulcorner p = \mathsf{null} \urcorner \\ p \rightsquigarrow \mathsf{Dag\ (Node}\ x\ T_1\ T_2) &\equiv \exists p_1 p_2.\ p \rightsquigarrow \{\!|\mathsf{item}{=}c, \mathsf{left}{=}p_1, \mathsf{right}{=}p_2|\!\} \circledast p_1 \rightsquigarrow \mathsf{Dag}\ T_1 \circledast p_2 \rightsquigarrow \mathsf{Dag}\ T_2 \end{aligned}$$

Question 2.7. *Justify the triple:* $\forall H l v.\ \{l \mapsto v \circledast H\}\ !l\ \{\lambda x. \ulcorner x = v \urcorner * (l \mapsto v \circledast H)\}$.

Question 2.8. *There is no* $\circledast$*-"frame rule" in general, but intuitively, is there a class of programs on which this modified frame rule would hold? What can you say about the one for* $\mathbin{\mathbb{\wedge}}$*? (No proof required)*

You may use Question 2.7 in the following, but not Question 2.8, except if you have provided a proof. We are now ready to establish a stronger specification:

$$\forall pT\ \{p \rightsquigarrow \mathsf{Dag}\ T\}\ \texttt{copy}\ p\ \{\lambda q.\ p \rightsquigarrow \mathsf{Dag}\ T * q \rightsquigarrow \mathsf{Mtree}\ T\} \tag{4}$$

Question 2.9. *Prove* (4). *(Hint: the frame rule can be used multiple times, but it cannot be used with* $\circledast$*, so you need to generalize your induction. Do specify how and where the frame rule is applied.)*

2.4 Concurrent building blocks

Consider now a concurrent setting with parallelism and mutual-exclusion locks (mutex locks). The primitives on locks satisfy the following triples, where $l \rightsquigarrow \mathsf{Lock}\ R$ is a duplicable heap predicate, and R is called a lock invariant, or resource invariant:

$$\begin{array}{llll} \forall R & \{R\} & \texttt{create\_lock ()} & \{\lambda l.\ l \rightsquigarrow \mathsf{Lock}\ R\} \\ \forall l R & \{l \rightsquigarrow \mathsf{Lock}\ R\} & \texttt{acquire\_lock}\ l & \{\lambda \_.\ R * l \rightsquigarrow \mathsf{Lock}\ R\} \\ \forall l R & \{R * l \rightsquigarrow \mathsf{Lock}\ R\} & \texttt{release\_lock}\ l & \{\lambda \_.\ l \rightsquigarrow \mathsf{Lock}\ R\} \end{array} \tag{5}$$

It is common to acquire a lock, do some work, then release the same lock. This is called a critical section or critical region. Let us introduce the notation `with l do e done` (not standard OCaml syntax) for the following expression, where `e` is not supposed to use `l`.

```
acquire_lock l;
let x = e in
release_lock l;
x
```

Question 2.10. *Give a proof rule for* `with l do e done`*. Explain why it holds. (3 lines)*

Recall the rule for the *parallel composition* of two terms `e1` and `e2` (written `e1 ||| e2`), also called *fork-join parallelism*, running in parallel on different threads and discarding the results.

$$\frac{\{P_1\}\ \texttt{e1}\ \{\lambda \_. Q_1\} \qquad \{P_2\}\ \texttt{e2}\ \{\lambda \_. Q_2\}}{\{P_1 * P_2\}\ \texttt{e1 ||| e2}\ \{\lambda \_. Q_1 * Q_2\}}$$

Consider now the following program where two threads increment a common reference `r` in parallel critical sections, each remembering the resulting values into two different references.

```
let r = ref 0
let r1 = ref 0
let r2 = ref 0
let l = create_lock ()

let prog =
  (thread1 () ||| thread2 ());
  acquire_lock l;
  assert (!r1 + !r2 == 3)
```

```
let thread1 () =
  with l do
    r := !r + 1;
    r1 := !r
  done
```

```
let thread2 () =
  with l do
    r := !r + 1;
    r2 := !r
  done
```

Question 2.11. *Establish the safety of this program. (Write a proof sketch with the lock invariant and important steps; do not write all details of each rule instantiation. 20 lines should be enough.)*

Consider the program c below, given `c1` and `c2`

```
let l = create_lock () in
(with l do c1 done ||| with l do c2 done);
acquire_lock l
```

First, assume that `c1` and `c2` indeed do not mention `l`.

Question 2.12. *Intuitively, what are the possible executions of c? Give the premise of a rule with conclusion $\{P\}\ c\ \{\lambda\_.\ Q\}$ that would reflect this intuition. How would you annotate this program c with auxiliary variables in order to establish a rule of this form, and what would be the corresponding lock invariant?*

Question 2.13. *Does your proof assume that `c1` and `c2` do not use the lock `l`? Is it still valid if for example `c1` releases `l`, then acquires it again? Is there any other problem in this case?*