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 9 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.*

Answer.

1. Faux, car le tableau `a` peut être vide.
2. Vrai.
3. Vrai.
4. Vrai.
5. Vrai.
6. Faux, car on ne pourra pas prouver `a[r] <= a[i+1]` à la fin de l'itération. En effet, au début de l'itération l'invariant ne garantit pas `a[r] <= a[i]`.
7. Faux, car l'invariant ne couvre pas l'indice 0.
8. Vrai.

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?*

Answer. The logic of Why3 and the first order solver used suppose that every type are inhabited. So we can suppose that `e` of type `q` exists. Since it is of type `q` it should verify the invariant, which would lead to a contradiction if no elements of the type verify the invariant.

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

Answer. In first order logic well-typed term exists, so the term `{num=2;den=2}` is valid. If this record is of type `q`, the axiom `inv_q` would be false.

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`.

Answer.

```
val new_id () : t
  ensures { id result = counter }
  ensures { counter = (old counter) + 1 }
  writes { 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`.

Answer.

```
type p = { a: t; b : t}
invariant { id a < id b }
by
  let ida = new_id () in
  let idb = new_id () in
  { a = ida; b = idb }
```

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?*

Answer. We want to check the Hoare triple $\{true\}By\{Inv(result)\}$, for example by checking the formula $WP(By, Inv(result))$.

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`.

Answer.

```
type t = int

function id (x:t) : int = x

let new_id () : t
    ensures { id result = counter }
    ensures { counter = (old counter) + 1 }
    writes { counter } =
      counter ← counter + 1;
      counter
```

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 $(1 \mapsto 1 * 2 \mapsto 2) = (P * P)$.*

Answer. $1 \mapsto 1 \mathbin{\mathbb{\vee}} 2 \mapsto 2$ works, as well as $1 \overset{1/2}{\mapsto} 1 * 2 \overset{1/2}{\mapsto} 2$.

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

Answer. $\ulcorner \urcorner \triangleright P \mathrel{-\!\!*} P$: assume $\ulcorner \urcorner m$ i.e. $m = \emptyset$ let us prove $(P \mathrel{-\!\!*} P)m$. Let m_1 s.t. $m_1 \perp m$ and $P\,m_1$, let us prove $P\,(m_1 \uplus m)$, this holds since $m_1 \uplus m = m_1 \uplus \emptyset = m_1$.

Second entailment: assuming $(P * (Q \mathrel{-\!\!*} R))\,m$, we prove $(Q \mathrel{-\!\!*} P * R)\,m$. Let m_1, m_2 such that $m_1 \perp m_2$ and $m = m_1 \uplus m_2$ with $P\,m_1$ and $(Q \mathrel{-\!\!*} R)\,m_2$. Let m_0 such that $m_0 \perp m$ and $Q\,m_0$, we need to prove $(P * R)\,(m \uplus m_0)$. Because $m_0 \perp (m_1 \uplus m_2)$ we also have $m_0 \perp m_2$, so by $(Q \mathrel{-\!\!*} R)\,m_2$ we have $R\,(m_2 \uplus m_0)$. We also $m_0 \perp m_1$ and so together with $m_1 \perp m_2$ we have $m_1 \perp (m_2 \uplus m_0)$, and so finally we have $(P * R)(m_1 \uplus (m_2 \uplus m_0))$ i.e. $(P * R)\,(m \uplus m_0)$.

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)*

Answer. With $P = R = r \mapsto 0$, $c = (r := 1)$ and $Q = r \mapsto 1$, we can derive:

$$\dfrac{\dfrac{\dfrac{}{\{r \mapsto 0\}\ r := 1\ \{\lambda\_.\ r \mapsto 1\}}\ \text{ASSIGN}}{\{r \mapsto 0 \mathbin{\mathbb{\wedge}} r \mapsto 0\}\ r := 1\ \{\lambda\_.\ r \mapsto 1 \mathbin{\mathbb{\wedge}} r \mapsto 0\}}\ \text{AND-FRAME-BAD}}{\{r \mapsto 0\}\ r := 1\ \{\lambda\_.\ \mathsf{False}\}}\ \text{CONSEQ}$$

This implies that if $r := 1$ terminates in a heap satisfying $r \mapsto 0$, then it does so in a heap satisfying False. Since the heap $\{(r, 0)\}$ satisfies $r \mapsto 0$ and since the program $r := 1$ does terminate in this heap into heap $\{(r, 1)\}$, then we know that we have a heap satisfying false, contradiction.

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

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

Answer. None holds. Using $\{r \mapsto 0\}\ r := 1\ \{r \mapsto 1\}$ as a premise for each rule, we give R and a heap that satisfies the precondition of the conclusion, but cannot be used to run safely the program $r := 1$.

1. $(- \Vdash R)$: with $R = \ulcorner\urcorner$ the heap $\emptyset$ satisfies $r \mapsto 0 \Vdash R$ but cannot run $r := 1$.
2. $(R \mathrel{-\!\!*} -)$: with $R = \ulcorner \mathsf{False} \urcorner$ the heap $\emptyset$ satisfies $R \mathrel{-\!\!*} r \mapsto 0$ but cannot run $r := 1$.
3. $(- \mathrel{-\!\!*} R)$: with $R = r \mapsto 0$ the heap $\emptyset$ satisfies $r \mapsto 0 \mathrel{-\!\!*} R$ but cannot run $r := 1$.

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` $: \alpha$ `node` $\to$ `unit` such that $\mathsf{TP}(\texttt{action})$ 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).

Answer. The difficulty lies here in the fact that `find` returns a pointer that is not separated from p, so we can use $\mathrel{-\!\!*}$ to specify the result:

$$\forall f p.\ \ \mathsf{TP}(f) \ \Rightarrow\ \{\mathsf{tree}(p)\}\ \texttt{find}\ f\ p\ \{\lambda q.\ \mathsf{tree}(q) * (\mathsf{tree}(q) \mathrel{-\!\!*} \mathsf{tree}(p))\}$$

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

Answer. Let f s.t. $\mathsf{TP}(f)$, let us use the left-intro rule of $\exists$ we prove by induction on T that:

$$\{p \rightsquigarrow \mathsf{Mtree}\ T\}\ \texttt{find}\ f\ p\ \{\lambda q.\ \mathsf{tree}(q) * (\mathsf{tree}(q) \mathrel{-\!\!*} \mathsf{tree}(p))\}$$

The first 'then' branch leaves us to prove $\mathsf{tree}(p) \triangleright \mathsf{tree}(p) * (\mathsf{tree}(p) \mathrel{-\!\!*} \mathsf{tree}(p))$ which holds by the frame rule and the fact that $\ulcorner\urcorner \triangleright P \mathrel{-\!\!*} P$.

If $T = \mathsf{Node}\ x\ T_1\ T_2$, introducing p_1 and p_2, entering (say) the then-branch, after framing $p \rightsquigarrow \{|x, p_1, p_2|\} * p_2 \rightsquigarrow \mathsf{Mtree}\ T_2$ to use the induction on T_1 we get a q s.t.

$$(\mathsf{tree}(q) * (\mathsf{tree}(q) \mathrel{-\!\!*} \mathsf{tree}(p_1))) * p \rightsquigarrow \{|x, p_1, p_2|\} * p_2 \rightsquigarrow \mathsf{Mtree}\ T_2$$

By Question 2.2 and framing we get

$$\mathsf{tree}(q) * (\mathsf{tree}(q) \mathrel{-\!\!*} (p \rightsquigarrow \{|x, p_1, p_2|\} * p_2 \rightsquigarrow \mathsf{Mtree}\ T_2 * \mathsf{tree}(p_1)))$$

from which we get by unfolding $\mathsf{tree}(p_1)$ then folding back $\mathsf{tree}(p)$:

$$\mathsf{tree}(q) * (\mathsf{tree}(q) \mathrel{-\!\!*} \mathsf{tree}(p))$$

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 $\divideontimes$ that we call here *partially separating conjunction*, and an inductive definition of $p \rightsquigarrow \mathsf{Dag}\ T$, as follows:

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

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

Answer. The predicate $l \mapsto v \divideontimes H$ ensures that (l, v) is in the starting heap, so `!l` is safe to run and will return v, indeed. Because `!l` does not modify the heap, the same predicate will hold after.

Question 2.8. *There is no $\divideontimes$-"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 $\wedge$? (No proof required)*

Answer. The class of programs satisfying this rule includes read-only programs, but also programs that may allocate memory and may modify the heap as long as they restore the initial values.

In other words we have:

$$\frac{\{P\}\ c\ \{\lambda x.\ Q\,x\} \qquad c \text{ preserves-memory-at-} P}{\{P \divideontimes R\}\ c\ \{\lambda x.\ Q\,x \divideontimes R\}}\ \text{RO-MIXAND-FRAME}$$

with "c preserves-memory-at-P" defined as:

$$\forall h.\ P\,h \Rightarrow \forall v h' h_1.\ h \perp h_1 \wedge (c, h \uplus h_1) \rightarrow^* (v, h') \Rightarrow \exists h_2.\ h' = h \uplus h_1 \uplus h_2$$

The proof of the above rule is a little tedious and was not asked.

A "frame rule" for $\wedge$ would exclude allocating programs and so it would be quite limited, because in principle $Q = P$. (Allocating programs should be excluded otherwise we would have e.g. $\{\ulcorner\urcorner \wedge \ulcorner\urcorner\}$ `ref 0` $\{\lambda x.\ x \mapsto 0 \wedge \ulcorner\urcorner\}$, which implies false.)

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 $\divideontimes$, so you need to generalize your induction. Do specify how and where the frame rule is applied.)*

Answer. We generalize the statement by adding $P \divideontimes -$ in pre and post, and prove by induction on T:

$$\forall TPp\ \{P \divideontimes p \rightsquigarrow \mathsf{Dag}\ T\}\ \texttt{copy}\ p\ \{\lambda q.\ (P \divideontimes p \rightsquigarrow \mathsf{Dag}\ T) * q \rightsquigarrow \mathsf{Mtree}\ T\}$$

When T is a leaf we need to prove $P \circledast \ulcorner p = \mathsf{null} \urcorner \triangleright \ulcorner \mathsf{null} = \mathsf{null} \urcorner * (P \circledast \ulcorner p = \mathsf{null} \urcorner)$ which holds by rule PROP-R, since indeed $\mathsf{null} = \mathsf{null}$. When $T = \mathsf{Node}\ x\ T_1\ T_2$, let us introduce p_1, p_2. We use Question 2.7 for different field accesses:

$\{P \circledast p \rightsquigarrow \{|x, p_1, p_2|\} \circledast p_1 \rightsquigarrow \mathsf{Dag}\ T_1 \circledast p_2 \rightsquigarrow \mathsf{Dag}\ T_2\}$

`let q1 = p.left in`

we use Question 2.7 and use $\ulcorner q_1 = p_1 \urcorner$ to replace `q1` with p_1 in the following

$\{P \circledast p \rightsquigarrow \{|x, p_1, p_2|\} \circledast p_1 \rightsquigarrow \mathsf{Dag}\ T_1 \circledast p_2 \rightsquigarrow \mathsf{Dag}\ T_2\}$

`let p1' = copy` p_1 `in`

by induction hypothesis with $P =$ [everything but $p_1 \rightsquigarrow \mathsf{Dag}\ T_1$]

$\{(P \circledast p \rightsquigarrow \{|x, p_1, p_2|\} \circledast p_1 \rightsquigarrow \mathsf{Dag}\ T_1 \circledast p_2 \rightsquigarrow \mathsf{Dag}\ T_2) * p_1' \rightsquigarrow \mathsf{Mtree}\ T_1\}$

we frame out $p_1' \rightsquigarrow \mathsf{Mtree}\ T_1$ **and** use Question 2.7 again to substitude `q2` with p_2

`let q2 = p.right in`

$\{(P \circledast p \rightsquigarrow \{|x, p_1, p_2|\} \circledast p_1 \rightsquigarrow \mathsf{Dag}\ T_1 \circledast p_2 \rightsquigarrow \mathsf{Dag}\ T_2) * p_1' \rightsquigarrow \mathsf{Mtree}\ T_1\}$

we frame out $p_1' \rightsquigarrow \mathsf{Mtree}\ T_1$ **then** use the I.H. with P being $P \circledast p \rightsquigarrow \{|x, p_1, p_2|\} \circledast p_1 \rightsquigarrow \mathsf{Dag}\ T_1$

$\{(P \circledast p \rightsquigarrow \{|x, p_1, p_2|\} \circledast p_1 \rightsquigarrow \mathsf{Dag}\ T_1 \circledast p_2 \rightsquigarrow \mathsf{Dag}\ T_2) * p_2' \rightsquigarrow \mathsf{Mtree}\ T_2 * p_1' \rightsquigarrow \mathsf{Mtree}\ T_1\}$

`let x = p.item in`

we frame out $p_1' \rightsquigarrow \mathsf{Mtree}\ T_1 * p_2' \rightsquigarrow \mathsf{Mtree}\ T_2$ then use Question 2.7 again

$\{(P \circledast p \rightsquigarrow \{|x, p_1, p_2|\} \circledast p_1 \rightsquigarrow \mathsf{Dag}\ T_1 \circledast p_2 \rightsquigarrow \mathsf{Dag}\ T_2) * p_2' \rightsquigarrow \mathsf{Mtree}\ T_2 * p_1' \rightsquigarrow \mathsf{Mtree}\ T_1\}$

we use the rule for allocation by framing **everything** to get

`{ item = x; left = p1'; right = p2' }`

$\{\lambda q.\ (P \circledast p \rightsquigarrow \{|x, p_1, p_2|\} \circledast p_1 \rightsquigarrow \mathsf{Dag}\ T_1 \circledast p_2 \rightsquigarrow \mathsf{Dag}\ T_2) * p_2' \rightsquigarrow \mathsf{Mtree}\ T_2 * p_1' \rightsquigarrow \mathsf{Mtree}\ T_1 * q \rightsquigarrow \{|x, p_1', p_2'|\}\}$

$\{\lambda q.\ (P \circledast p \rightsquigarrow \mathsf{Dag}\ T) * q \rightsquigarrow \mathsf{Mtree}\ T\}$

Inspired from §6 of J. Reynolds's *Separation logic: a logic for shared mutable data structures*, LICS'02.

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)*

Answer. Because we typically do not use l in `e`, we can write:

$$\forall l e R P Q\ \frac{\{R * P\}\ e\ \{\lambda x.\ R * Q\,x\}}{\{l \rightsquigarrow \mathsf{Lock}\ R * P\}\ \texttt{with l do e done}\ \{\lambda x.\ l \rightsquigarrow \mathsf{Lock}\ R * Q\,x\}}\ \text{WITH-DO}$$

It can be derived by applying the frame rule, framing $l \rightsquigarrow \mathsf{Lock}\ R$, then applying the rule for sequence twice and the rules for acquire and release.

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 thread1 () =              let thread2 () =
let r1 = ref 0                     with l do                     with l do
let r2 = ref 0                       r := !r + 1;                  r := !r + 1;
let l = create_lock ()               r1 := !r                      r2 := !r
                                   done                          done

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

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.)*

Answer. The resource invariant must have full ownership of r because r is written to in both threads. In order to talk about the value of r_1 and r_2 it should have a partial ownership of them, so we choose the following invariant, where A is a pure predicate on $\mathbb{Z}^3$ describing all possible steps:

$$R = \exists n n_1 n_2.\ r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * \ulcorner (n, n_1, n_2) \in A \urcorner$$

$$A = \{(0,0,0), (1,1,0), (2,1,2), (1,0,1), (2,2,1)\}$$

After lock creation we reach state $l \rightsquigarrow \mathsf{Lock}\ R * r_1 \overset{1/2}{\mapsto} 0 * r_2 \overset{1/2}{\mapsto} 0$, since indeed $(0,0,0) \in A$. We now prove for each thread

$$\{l \rightsquigarrow \mathsf{Lock}\ R * r_i \overset{1/2}{\mapsto} 0\}\ \texttt{thread}i\ ()\ \{\lambda\_. l \rightsquigarrow \mathsf{Lock}\ R * \exists n_i\ r_i \overset{1/2}{\mapsto} n_i * \ulcorner n_i \neq 0 \urcorner\}$$

Using rule WITH-DO it is enough to prove:

$$\{R * r_i \overset{1/2}{\mapsto} 0\}\ \texttt{r := !r + 1; ri := !r}\ \{\lambda\_. R * \exists n_i\ r_i \overset{1/2}{\mapsto} n_i * \ulcorner n_i \neq 0 \urcorner\}$$

Let us do that for $i = 0$. Unfolding R, let $(n, n_1, n_2) \in A$, we proceed step by step

$\{r \mapsto n * r_1 \overset{1/2}{\mapsto} n_1 * r_2 \overset{1/2}{\mapsto} n_2 * r_1 \overset{1/2}{\mapsto} 0\}$
by FRAC-AGREE we get $n_1 = 0$ and

$\{r \mapsto n * r_1 \mapsto 0 * r_2 \overset{1/2}{\mapsto} n_2\}$
`r := !r + 1;`

$\{r \mapsto n+1 * r_1 \mapsto 0 * r_2 \overset{1/2}{\mapsto} n_2\}$
`r1 := !r;`

$\{r \mapsto n+1 * r_1 \mapsto n+1 * r_2 \overset{1/2}{\mapsto} n_2\}$

$\{r \mapsto n+1 * r_1 \overset{1/2}{\mapsto} n+1 * r_2 \overset{1/2}{\mapsto} n_2 * r_1 \overset{1/2}{\mapsto} n+1\}$
because $n_1 = 0$ we know that $n \geq 0$ so $n + 1 \neq 0$, so

$\{r \mapsto n+1 * r_1 \overset{1/2}{\mapsto} n+1 * r_2 \overset{1/2}{\mapsto} n_2 * \exists n_1\ r_1 \overset{1/2}{\mapsto} n_1 * \ulcorner n_1 \neq 0 \urcorner\}$
because $n_1 = 0$ and $(n, 0, n_2) \in A$ we know that $(n, n_2) \in \{(0,0), (1,1)\}$
and so $(n+1, n+1, n_2) \in \{(1,1,0), (2,2,1)\} \subseteq A$, so we can fold R:

$\{R * \exists n_1\ r_1 \overset{1/2}{\mapsto} n_1 * \ulcorner n_1 \neq 0 \urcorner\}$

The case for $i = 1$ is very similar, exchanging the roles of n_1 and n_2.

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?*

Answer. The executions are c_1 then c_2, or c_2 then c_1. We could expect the premises for $\{P\}\ c\ \{\lambda\_.\ Q\}$ to be the conjunction of those for $\{P\}\ c_1; c_2\ \{\lambda\_.\ Q\}$ and $\{P\}\ c_2; c_1\ \{\lambda\_.\ Q\}$, i.e. something of the form:

$$\frac{\{P\}\ c_1\ \{\lambda\_.\ P_1\} \qquad \{P_1\}\ c_2\ \{\lambda\_.\ Q\} \qquad \{P\}\ c_2\ \{\lambda\_.\ P_2\} \qquad \{P_2\}\ c_1\ \{\lambda\_.\ Q\}}{\{P\}\ c\ \{Q\}}$$

The annotation could look like something like this

```
let b1 = ref false in
let b2 = ref false in
let l = create_lock () in
with l do (c1; b1 := true) done ||| with l do (c2; b2 := true) done
acquire_lock l
```

And the lock invariant for l would be:

$$\begin{aligned} R = \quad & P \ * \ b_1 \overset{1/2}{\mapsto} \mathsf{false} \ * \ b_2 \overset{1/2}{\mapsto} \mathsf{false} \\ \mathbb{V} \quad & P_1 \ * \ b_1 \overset{1/2}{\mapsto} \mathsf{true} \ * \ b_2 \overset{1/2}{\mapsto} \mathsf{false} \\ \mathbb{V} \quad & P_2 \ * \ b_1 \overset{1/2}{\mapsto} \mathsf{false} \ * \ b_2 \overset{1/2}{\mapsto} \mathsf{true} \\ \mathbb{V} \quad & Q \ * \ b_1 \overset{1/2}{\mapsto} \mathsf{true} \ * \ b_2 \overset{1/2}{\mapsto} \mathsf{true} \end{aligned}$$

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?*

Answer. The proof holds, it assumes nothing on `c1` and `c2`. However it could be difficult to use this rule, not because $l \rightsquigarrow \mathsf{Lock}\ R$ is not present in the assumption (it can be duplicated), but because R is very specific. It would mean that before the internal release the precondition P (or P_2 if `c2` already has run) must still hold.
