Final exam, March 1st, 2024

- 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 8 pages and **2** parts. Part 1 is about verification using weakest preconditions, part 2 is about separation logic.
- Write **your name** and **page numbers** under the form 1/7, 2/7, etc., on each piece of paper.
- **Please write your answers for Part 1 and Part 2 on separate pieces of paper.**

1 Takuzu Sequences

This exercise considers sequences of colors. To simplify we consider that there are only two different colors, abbreviated `B` for Black and `W` for White. Such a sequence of `B` and `W` is called *Takuzu* if it respects the following two rules.

1. No color should occur 3 times or more consecutively.
2. The number of occurrences of `B` and `W` should be equal.

For example, the sequence `WWBBBW` is not Takuzu since `B` appears three times consecutively. The sequence `WBBWBB` is not Takuzu since `B` appears more often than `W`. The sequence `WWBWBB` is Takuzu. We consider the following problem: given a *partial* sequence of `B` and `W`, that is a sequence with an arbitrary number of "holes", is it possible to fill the holes with colors to make the sequence Takuzu? Let us represent holes with the "`-`" character. For example, the partial sequence `-BBW-W` can be filled as Takuzu `WBBWBW`, but `BB-WW-` and `BBWBB-` cannot be made Takuzu, respectively because of the first rule and the second rule. To represent partial sequences we use arrays of elements of type `color` defined as

```
type color = B | W | H
```

where `H` denotes a hole. To formalize the first rule we introduce the following predicates.

```
predicate no_triple_sub (a:array color) (l:int) =
  ∀ i. 0 ⩽ i < l−2 → not (a[i] = a[i+1] = a[i+2] ⋀ a[i] ≠ H)
predicate no_triple (a:array color) = no_triple_sub a a.length
```

The predicate `no_triple` thus expresses that there is no already three consecutive `B` or `W` in the given partial sequence.
We now consider the program below which checks if a partial sequence respects the first rule.

```
exception TripleFound

let check_rule_1 (a:array color) : bool =
  let ref l2 = a[0] in let ref l1 = a[1] in let ref i = 2 in
  try
    while i < a.length do
      let v = a[i] in
      if v ≠ H && l1 = v && l2 = v then raise TripleFound;
      l2 ← l1; l1 ← v; i ← i+1
    done;
    True
  with TripleFound → False
```

Question 1.1. *Which annotations (e.g. pre-conditions, loop invariants, etc.) should be given to the program* `check_rule_1` *above so as to be able to prove it safe and terminating? Justify informally (10 lines max) why your annotations suffice.*

Answer. To prove safety we need to prove that the array accesses are within the bounds of the input array. Since we access to indices 0 and 1 we need the array to be of length at least 2. Moreover to be sure that the array access `a[i]` within the loop is safe we need to state a loop invariant on index `i`. Finally to prove termination we need a variant.

```
requires { a.length ⩾ 2 }
...
  invariant { 2 ⩽ i }
  variant { a.length − i }
```

Note: adding the invariant `i ⩽ a.length` is not necessary, but is not considered as a mistake. On the other hand, adding the invariant `i < a.length` is a mistake since it is not an invariant.

To express the intended behavior of program `check_rule_1`, we add the post-condition

```
ensures { (result = True) ↔ (no_triple a) }
```

Question 1.2. *Which additional annotations should be given to the program* `check_rule_1` *so as to be able to prove this post-condition? Justify informally (15 lines max) why your annotations suffice. Hint: distinguish the case when the loop exits with exception* `TripleFound` *and the case when it exits normally.*

Answer. When the loop exits with the exception `TripleFound`, we can prove the post-condition, because we just found three consecutive elements. This though requires to state loop invariants expressing that `l1` is always the element before index `i` and `l2` is always the element two cells before `i`.

```
  invariant { l2 = a[i−2] }
  invariant { l1 = a[i−1] }
```

When the loop exits normally, we need an additional loop invariant to express that at loop iteration `i` we know that there is no triple in the range `0..i`:

```
  invariant { no_triple_sub a i }
```

Thanks to the two other loop invariants given above, we can prove this invariant is preserved. We need the additional loop invariant

```
invariant { i ⩽ a.length}
```

to establish that `i = a.length` at loop exit, and thus prove the post-condition.

To formalize the second Takuzu rule, we introduce the following program which computes a pair of integers from a function of type `int -> color` and a range.

```
let rec diffs (f:int → color) (l h:int) : (int,int) =
  if h ⩽ l then (0,0) else let (d,e) = (diffs f (l+1) h) in
  match (f l) with
  | H → (d,e+1)
  | B → (d+1,e)
  | W → (d−1,e)
  end
```

Notice that if `(d,e) = (diffs f l h)`, then `e` is the number of holes in the sequence `(f l)`, `(f (l+1))`, ..., `(f (h−1))` and `d` is the number of `B` minus the number of `W` in the same sequence.

Question 1.3. *Justify informally (5 lines max) how one could consider the program* `diffs` *above as a logic function.*

Answer. to consider it as a logic function we need it to be pure (no side effects) which is trivially the case, and terminating, which can be proved by adding a variant

```
variant { h − l }
```

Assuming now we have the function `diffs` in the logic context, we pose the predicate

```
predicate balanced (a:array color) =
  let (d,e) = (diffs a.elts 0 a.length) in −e ⩽ d ⩽ e
```

and admit that when `(balanced a)` is false then the partial sequence `a` cannot be completed to a sequence satisfying the second Takuzu rule.
Towards a brute-force search algorithm to check if a partial sequence can be completed into a Takuzu, we consider the following program taking a partial sequence `a`, tries to fill one hole with a color `c`, and checks if the sequence keeps satisfying the `no_triple` and the `balanced` predicates. To simplify we assume the hole that we attempt to fill is at index 0.

```
1  exception NotBalanced
2
3  let update_first_cell (a:array color) (c:color) (d e:int) : (int,int)
4    requires { a.length ⩾ 3 ⋀ a[0] = H ⋀ c ≠ H }
5    requires { (no_triple a) }
6    requires { (d,e) = (diffs a.elts 0 a.length) }
7    writes { a }
8    ensures { a[0] = c ⋀ ∀ i. 1 ⩽ i < a.length → a[i] = old a[i] }
9    ensures { (no_triple a) }
10   ensures { result = (diffs a.elts 0 a.length) }
11   raises { TripleFound → not (no_triple a) }
12   raises { NotBalanced → not (balanced a) }
13 = a[0] ← c;
14   if a[1] = a[2] = c then raise TripleFound;
15   match c with
16   | H → absurd
17   | B → if d ⩾ e−1 then raise NotBalanced else (d+1,e−1)
18   | W → if d ⩽ −e+1 then raise NotBalanced else (d−1,e−1)
19   end
```

Question 1.4. *Explain informally (10 lines max) why the post-condition* `(no_triple a)` *at line 9 and the exceptional post-condition* `not (no_triple a)` *at line 11 are provable.*

Answer. When the program raises exception `TripleFound` at line 14 we just have checked that the three first elements are of color `c` so the exceptional post-condition at line 11 trivially holds. When the program exits normally we have checked that the first three elements are not of the same color, and the pre-condition at line 5 tells us that the initial array had no triple yet, hence there is no triple elsewhere in the final array too.

Question 1.5. *Explain informally (20 lines max) why the post-condition*
`result = (diffs a.elts 0 a.length)` *at line 10 and the exceptional post-condition* `not (balanced a)` *at line 12 are* ***not*** *provable by a simple reasoning. Propose a method to obtain their proofs.*

Answer. With both the normal exit and the exceptional exits at lines 17 and 18, to prove post-condition we need to relate `(diffs a.elts)` after the modification of `a[0]` with `(diffs a.elts)` before the modification. Since the definition of `diffs` is recursive, we need to know that `(diffs a.elts 1 a.length)` and `(diffs (old a).elts 1 (old a).length)` are equal, which is not a simple fact: it can be proved only by an induction. It is a classical case of the need of a *frame lemma* for `diffs`. Such a lemma can be stated and proved as

```
let rec lemma diffs_frame (f:int → color) (g:int → color) (l h:int) : unit
  requires { ∀ i. l ⩽ i < h → f i = g i }
  variant { h − l }
  ensures { diffs f l h = diffs g l h  }
= if h ⩽ l then () else diffs_frame f g (l+1) h
```

Note: there is no need to formally express the lemma to succeed this question, a sufficiently precise and convincing informal answer is enough.

---

2 Separation Logic

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

We recall the definition of a few separation logic connectives:

$$H_1 \mathbin{\wedge\!\!\!\wedge} H_2 \equiv \lambda m.\ H_1\, m \wedge H_2\, m \qquad l \mapsto v \equiv \lambda m.\ m = \{(l,v)\} \wedge l \neq \mathsf{null} \qquad l \mapsto \_ \equiv \exists v.\, l \mapsto v$$

$$\ulcorner P \urcorner \equiv \lambda m.\ m = \varnothing \wedge P \qquad H_1 * H_2 \equiv \lambda m.\ \exists m_1 m_2.\, m = m_1 \uplus m_2 \wedge H_1\, m_1 \wedge H_2\, m_2$$

$$\mathsf{GC} \equiv \lambda m.\ \mathsf{True} \qquad H_1 \mathrel{-\!\!*} H_2 \equiv \lambda m.\ \forall m_1\ (m_1 \perp m \wedge H_1\, m_1) \Rightarrow H_2(m_1 \uplus m)$$

Remember that list cells are records with mutable fields `hd` and `tl` such that $p \leadsto \{|\mathsf{hd} = x; \mathsf{tl} = q|\}$ is synonymous with $p + 0 \mapsto x * p + 1 \mapsto q$.

Question 2.1. *For each of the following heap predicates, say how many unique heaps satisfy it, and give examples of such heaps when applicable.*

1. $p \leadsto \mathsf{MlistSeg}\, q\, [1, 2]$
2. $p \leadsto \mathsf{MlistSeg}\, q\, [1] \mathbin{\wedge\!\!\!\wedge} \mathsf{GC}$
3. $(1 \mapsto 1) \mathrel{-\!\!*} \mathsf{GC}$
4. $(2 \mapsto 3 * \mathsf{GC}) \mathbin{\wedge\!\!\!\wedge} 1 \leadsto \mathsf{Mlist}\, [1, 2]$

Answer.

1. Infinitely many: exactly those of the form $\{(p, 1), (p + 1, r), (r, 2), (r + 1, q)\}$, with $r \notin \{p - 1, p, p + 1, \mathsf{null}, \mathsf{null} - 1\}$ (but 0 if $p \in \{\mathsf{null}, \mathsf{null} - 1\}$);
2. exactly one: $\{(p, 1), (p + 1, q)\}$ (0 if $p \in \{\mathsf{null}, \mathsf{null} - 1\}$);
3. infinitely many, in fact all heaps;
4. exactly one: $(2 \mapsto 3 * \mathsf{GC})m \Leftrightarrow m(2) = 3$ and $(1 \leadsto \mathsf{Mlist}\, [1, 2])m \Leftrightarrow \exists p, m = \{(1, 1), (2, p)\} \uplus \{(p, 2), (p + 1, \mathsf{null})\}$ so the conjunction of those is equivalent to $m = \{(1, 1), (2, 3), (3, 2), (4, \mathsf{null})\}$.

Let us define the (non-standard) symbol ▸ as a heap predicate implying a pure fact:

$$H \blacktriangleright P \quad \equiv \quad \forall m. H m \Rightarrow P$$

Note that $H \blacktriangleright P$ is equivalent to $H \rhd H * \ulcorner P \urcorner$. Here is an example: $p \leadsto \mathsf{Mlist}(x :: L) \ \blacktriangleright\ p \neq \mathsf{null}$.

Question 2.2. *Show that* $1 \leadsto \mathsf{Mlist}[2; 4; 6] \mathbin{\wedge\!\!\!\wedge} (\mathsf{GC} * 2 \leadsto \mathsf{MlistSeg}\ q\ [3; 5]) \blacktriangleright q = 6$. *(Approx. 10 lines)*

Answer. Let m satisfying the heap predicate; m must satisfy both operands of $\mathbin{\wedge\!\!\!\wedge}$. By unfolding Mlist's definition and some rewriting, we must have p, p', r, m' such that

$$m = \{(1, 2), (2, p)\} \uplus \{(p, 4), (p + 1, p')\} \uplus \{(p', 6), (p' + 1, \mathsf{null})\}$$

$$m = m' \uplus \{(2, 3), (3, r)\} \uplus \{(r, 5), (r + 1, q)\}$$

It must follow that $p = m(2) = 3$, so $p = 3$.
This means that $4 = m(p) = m(3) = r$, so $r = 4$.
That means that $p' = m(p + 1) = m(4) = m(r) = 5$, so $p' = 5$.
That means that $6 = m(p') = m(5) = m(r + 1) = q$, so $q = 6$. And $m' = \{(1, 2), (6, \mathsf{null})\}$.

Question 2.3. *Let* $P = \exists p.\, 1 \mapsto p * p \mapsto 2$. *Is it possible to find* R *such that* $P \rhd (1 \mapsto \_ \mathbin{\wedge\!\!\!\wedge} R) * (R \mathrel{-\!\!*} P)$*? If so, prove the entailment, and if not, the impossibility. (Approx. 5 lines)*

Answer. No. Suppose we have such an R. We have, for all $p \neq 1$, $\{(1, p), (p, 2)\}$ satisfying P. There is only one way to split this heap into two for the separating conjunction, which means that for all $p \neq 1$, $\{(1, p)\}$ satisfies R, and that for all $p \neq 1$, $\{(p, 2)\}$ satisfies $R \mathrel{-\!\!*} P$. For example, $\{(5, 2)\}$ satisfies $R \mathrel{-\!\!*} P$. That means that for every m such that $m \perp \{(5, 2)\}$ and Rm, we must have $P(m \uplus \{5, 2\})$. We have seen that $\{1, 6\}$ is such an m, and so we must have $P(\{1, 6\} \uplus \{(5, 2)\})$, which is a contradiction.

Consider the following function `go` on mutable trees. We also recall the following rule for while loops, which is enough, in combination with induction, to establish total correctness.

```
let go (p : 'a node) : 'a node =
  let x = ref p in
  let y = ref null in
  while !x <> null do
    let q = !x.right in
    !x.right <- !y;
    y := !x;
    x := q
  done;
  !y
```

```
type 'a node = {
    mutable item : 'a;
    mutable left : 'a node;
    mutable right : 'a node;
  }
```

While-step

$$\frac{\{H\}\ \texttt{if b then (c; while b do c) else ()}\ \{Q\}}{\{H\}\ \texttt{while b do c}\ \{Q\}}$$

Question 2.4. *Let $tree(p)$ be the heap predicate $\exists T.\, p \rightsquigarrow \mathsf{Mtree}\ T$. Show that $\{tree(p)\}$ `go` p $\{\lambda r. tree(r)\}$. (Provide a detailed proof of total correctness, and carefully state your inductions. Approx. 20 lines.)*

Answer. After allocating x and y we have the assertion $\{A\}$ with $A \equiv x \mapsto p * y \mapsto \mathsf{null} * tree(p)$. Looking at the desired postcondition in the function's specification, we suspect we want $B \equiv \exists r.\, x \mapsto \mathsf{null} * y \mapsto r * tree(r)$ as a postcondition for the while loop (then we would be done by the load and GC rules). Let us introduce the existentially quantified T be (by extrusion and $\exists$-L), so that we can prove the total correctness triple by induction on T (where `b` and `c` are the loop's condition and body):

$$\forall pr, \{x \mapsto p * p \rightsquigarrow \mathsf{Mtree}\ T * y \mapsto r * tree(r)\}\ \texttt{while b do c}\ \{\lambda\_.B\} \qquad (1)$$

In both cases we apply the rule for while, which only means that we need to prove the same pre&post for the program `if b then (c; while b do c) else ()`. If T is a leaf, then $p = \mathsf{null}$ and we need to prove the following triple, which holds by Val-Frame.

$$\{x \mapsto \mathsf{null} * \mathsf{null} \rightsquigarrow \mathsf{Mtree\ Leaf} * y \mapsto r * tree(r)\}\ \texttt{()}\ \{\lambda\_.B\}$$

If $T = \mathsf{Node}(v, T_1, T_2)$ then $p \rightsquigarrow \mathsf{Mtree}\ T$ unfolds, and we get successively:

	$\{x \mapsto p * p \rightsquigarrow \{v, p_1, p_2\} * p_1 \rightsquigarrow\ T_1 * p_2 \rightsquigarrow\ T_2 * y \mapsto r * tree(r)\}$
`let q = (!x).right in`	$\{x \mapsto p * p \rightsquigarrow \{v, p_1, q\} * p_1 \rightsquigarrow\ T_1 * q \rightsquigarrow\ T_2 * y \mapsto r * tree(r)\}$
`(!x).right ← !y`	$\{x \mapsto p * p \rightsquigarrow \{v, p_1, r\} * p_1 \rightsquigarrow\ T_1 * q \rightsquigarrow\ T_2 * y \mapsto r * tree(r)\}$
`y := !x`	$\{x \mapsto p * p \rightsquigarrow \{v, p_1, r\} * p_1 \rightsquigarrow\ T_1 * q \rightsquigarrow\ T_2 * y \mapsto p * tree(r)\}$
`x := q`	$\{x \mapsto q * p \rightsquigarrow \{v, p_1, r\} * p_1 \rightsquigarrow\ T_1 * q \rightsquigarrow\ T_2 * y \mapsto p * tree(r)\}$

Remains now the "`; while b do c`" part, for which we conclude by matching the precondition of (1) by taking $p = q$, $T = T_2$, $r = p$, and after framing, it remains to prove:

$$p \rightsquigarrow \{v, p_1, r\} * p_1 \rightsquigarrow\ T_1 * tree(r) \rhd tree(p) \qquad \text{, i.e. ,}$$

$$p \rightsquigarrow \{v, p_1, r\} * p_1 \rightsquigarrow\ T_1 * \exists T_r.\, r \rightsquigarrow T_r\ \rhd\ \exists T_p.\, p \rightsquigarrow \mathsf{Mtree}\ T_p\ ,$$

which can be established by extruding T_r and choosing $T_p = \mathsf{Node}(v, T_1, T_r)$.

We consider now a function `push_right : α cell → α → unit` with the following specification:

$$\forall Lxp, \{\ulcorner P \urcorner * p \rightsquigarrow \mathsf{Mlist}\, L\}\ \texttt{push\_right}\ p\ x\ \{\lambda\_.p \rightsquigarrow \mathsf{Mlist}\,(L +\!\!+ [x])\} \qquad (2)$$

where $+\!\!+$ is list concatenation and P is a proposition that can depend on L, x, p.

Question 2.5. *Why cannot P be simply True for a reasonable implementation? Give a most general proposition P such that (2) can be satisfied (no proof required).*

Answer. Because then with $p = \mathsf{null}$ and $L = \mathsf{Nil}$ we would have $\{\ulcorner \mathsf{null} = \mathsf{null} \urcorner\}$ `push_right` $p\ x$ $\{\lambda\_.\mathsf{null} \rightsquigarrow \mathsf{Mlist}\,[x]\}$, the second part implying that $\mathsf{null} \neq \mathsf{null}$, which implies that it does not terminate. Satisfactory answers for P include $p \neq \mathsf{null}$, $|L| > 0$, and $L \neq []$.

Question 2.6. *Implement* `push_right` *and prove that it satisfies* (2)*. (Be less detailed than in Question 2.4, but stay precise in induction hypotheses and give assertions between all steps. Approx. 15 lines.)*

Answer.

```
let rec push_right p x =
  if p.tl == null then
    let q = { hd = x; tl = null } in
    p.tl <- q
  else
    let q = p.tl in
    push_right q x
```

We choose $P = (L \neq [])$ and we prove, by induction on L, that (2) holds for all p. If $L = []$ then $\neg P$ so the triple holds vacuously. If $L = [y]$ then `p.tl` evaluates to null and we enter the if's first branch.

$$\begin{array}{ll}
 & \{p \rightsquigarrow \mathsf{Mlist}\,[y]\} \\
 & \{\exists p'.\, p \rightsquigarrow \{\!|y, p'|\!\} * \ulcorner p' = \mathsf{null} \urcorner\} \\
 & \{p \rightsquigarrow \{\!|y, \mathsf{null}|\!\}\} \\
\texttt{let q = \{ hd = x; tl = null \} in} & \{p \rightsquigarrow \{\!|y, \mathsf{null}|\!\} * q \rightsquigarrow \{\!|x, \mathsf{null}|\!\}\} \\
\texttt{p.tl} \leftarrow \texttt{q} & \{p \rightsquigarrow \{\!|y, q|\!\} * q \rightsquigarrow \{\!|x, \mathsf{null}|\!\}\} \\
 & \{p \rightsquigarrow \mathsf{Mlist}\,[y; x]\}
\end{array}$$

The last case if $L = y :: L'$ with $L' \neq []$, and entering the second branch

$$\begin{array}{lll}
 & \{p \rightsquigarrow \mathsf{Mlist}\, y :: L'\} & \\
 & \{\exists p'.\, p \rightsquigarrow \{\!|y, p'|\!\} * p' \rightsquigarrow \mathsf{Mlist}\, L'\} & \\
\texttt{let q = p.tl} & \{p \rightsquigarrow \{\!|y, q|\!\} * q \rightsquigarrow \mathsf{Mlist}\, L'\} & \\
\texttt{push_right q x} & \{p \rightsquigarrow \{\!|y, q|\!\} * q \rightsquigarrow \mathsf{Mlist}\,(L' +\!\!+ [x])\} & \text{by I.H. (with } q\text{) + framing out } p \rightsquigarrow \{\!|y, q|\!\} \\
 & \{p \rightsquigarrow \mathsf{Mlist}\,(y :: (L' +\!\!+ [x]))\} & \\
 & \{p \rightsquigarrow \mathsf{Mlist}\,((y :: L') +\!\!+ [x])\} &
\end{array}$$

Let us now consider the function `transfer_one`:

```
let transfer_one (p : 'a cell) (r : 'a cell ref) : unit =
  let x = !r.hd in
  r := !r.tl;
  push_right p x
```

Question 2.7. *Give a specification for* `transfer_one` *(no proof required).*

Answer.

$$\forall L_1 x L_2 q,\ L_1 \neq [] \Rightarrow \begin{array}{c} \{p \rightsquigarrow \mathsf{Mlist}\, L_1 * r \mapsto q * q \rightsquigarrow \mathsf{Mlist}\,(x :: L_2)\} \\ \texttt{transfer\_one}\ p\ r \\ \{\lambda\_.\, p \rightsquigarrow \mathsf{Mlist}\,(L_1 +\!\!+ [x]) * \exists q'.\, r \mapsto q' * q' \rightsquigarrow \mathsf{Mlist}\, L_2\} \end{array}$$

Assume now that our logic supports fractional permissions, *i.e.* it features heap predicates of the form $l \overset{\alpha}{\mapsto} v$ where $0 < \alpha \leqslant 1$ where $l \mapsto v$ is in fact short for $l \overset{1}{\mapsto} v$, and whenever $0 < \alpha, \beta \leqslant 1$,

$$l \overset{\alpha+\beta}{\mapsto} v \;=\; l \overset{\alpha}{\mapsto} v \,*\, l \overset{\beta}{\mapsto} v \qquad \text{and} \qquad l \overset{\alpha}{\mapsto} v \,*\, l \overset{\beta}{\mapsto} w \;\blacktriangleright\; v = w\ . \tag{3}$$

Representation predicates are redefined accordingly, e.g. $p \overset{\alpha}{\rightsquigarrow} \{\!|x; q|\!\}$ is $p \overset{\alpha}{\mapsto} x * p + 1 \overset{\alpha}{\mapsto} q$ and $p \overset{\alpha}{\rightsquigarrow} \mathsf{Mlist}(x :: L) = \exists p'.\, p \overset{\alpha}{\rightsquigarrow} \{\!|x; p'|\!\} * p' \overset{\alpha}{\rightsquigarrow} \mathsf{Mlist}\ L$. Note that $p \overset{\alpha}{\rightsquigarrow} \mathsf{Mlist}\ []$ is still $\ulcorner p = \mathsf{null} \urcorner$. Let us also define two additional representation predicates $list_\alpha$ and $rlist_\alpha$:

$$list_\alpha(p, n) \;\equiv\; \exists L.\, p \overset{\alpha}{\rightsquigarrow} \mathsf{Mlist} L * \ulcorner |L| = n \urcorner \qquad\qquad rlist_\alpha(r, n) \;\equiv\; \exists p.\, r \overset{\alpha}{\mapsto} p * list_\alpha(p, n)$$

Question 2.8. *How can we prove $rlist_\alpha(r, n) * rlist_\beta(r, m) \blacktriangleright n = m$? (Only state intermediate results, main steps, and inductions, do not provide a full proof. Approx. 5 lines.)*

Answer. One can prove, by induction on L_1, that $\forall p L_2,\ p \overset{\alpha}{\rightsquigarrow} \mathsf{Mlist}\ L_1 * p \overset{\beta}{\rightsquigarrow} \mathsf{Mlist}\ L_2 \blacktriangleright L_1 = L_2$ (*):

- if $L_1 = []$, then $p = \mathsf{null}$, then $L_2 = []$,
- if $L_1 = x :: L_1'$, then $p \neq \mathsf{null}$ then L_2 is of the form $x_2 :: L_2'$, and so

  $$\exists p_1, p_2.\, p \overset{\alpha}{\mapsto} x_1 * p + 1 \overset{\alpha}{\mapsto} p_1 * p_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Mlist}\ L_1' * p \overset{\beta}{\mapsto} x_2 * p + 1 \overset{\beta}{\mapsto} p_2 * p_2 \overset{\beta}{\rightsquigarrow} \mathsf{Mlist}\ L_2'$$

  by (3) we derive $x_1 = x_2$ and $p_1 = p_2$, the latter allowing us to use the I.H. to prove $L_1' = L_2'$.

Similarly, expanding $rlist_\alpha(r,n) * rlist_\beta(r,m) \blacktriangleright n = m$ and extruding existentials, we need to prove that for all p_1, L_1, p_2, L_2,

$$r \overset{\alpha}{\mapsto} p_1 * p_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Mlist}\ L_1 * \ulcorner |L_1| = n \urcorner * r \overset{\beta}{\mapsto} p_2 * p_2 \overset{\beta}{\rightsquigarrow} \mathsf{Mlist}\ L_2 * \ulcorner |L_2| = m \urcorner \blacktriangleright n = m$$

and similarly, we first get $p_1 = p_2$ by (3), then which helps us use (*) to get $L_1 = L_2$ hence $n = m$.

Question 2.9. *Give a specification for* `transfer_one` *in terms of $list_\alpha$ and $rlist_\alpha$ (no proof required).*

Answer.

$$\forall pnrm, n \geqslant 1 \wedge m \geqslant 1 \Rightarrow \begin{array}{c}\{list_1(p,n) * rlist_1(r,m)\} \\ \texttt{transfer\_one}\ p\ r \\ \{\lambda\_.list_1(p,n+1) * rlist_1(r,m-1)\}\end{array}$$

Consider now concurrent programs, where `e1 ||| e2` runs expressions `e1` and `e2` in parallel. Programs use locks (type `lock`) as a synchronization mechanism through the following primitives, where $l \rightsquigarrow \mathsf{Lock}\,R$ is a duplicable heap predicate (P is duplicable if $P \rhd P * P$):

`create_lock : unit → lock`	$\{\ulcorner\urcorner\}$	`create_lock ()`	$\{\lambda l.l \rightsquigarrow \mathsf{Lock}\,R\}$
`acquire : lock → unit`	$\{l \rightsquigarrow \mathsf{Lock}\,R\}$	`acquire l`	$\{\lambda\_.R * l \rightsquigarrow \mathsf{Lock}\,R\}$
`release : lock → unit`	$\{R * l \rightsquigarrow \mathsf{Lock}\,R\}$	`release l`	$\{\lambda\_.l \rightsquigarrow \mathsf{Lock}\,R\}$

The function `fill` uses locks and the function `zip` calls it twice in parallel:

```
let rec fill (l : lock) (p : 'a cell) (r : 'a cell ref) : unit =
  if !r <> null then
    (acquire l;
     transfer_one p r;
     release l;
     fill l p r)

let zip (p : 'a cell) (q : 'a cell ref) (r : 'a cell ref) =
  let l = create_lock () in
  fill l p q ||| fill l p r;
  acquire l
```

Question 2.10. *Give a proof sketch (approx. 20 lines) for:*

$$\forall pqr, \{list_1(p,1) * rlist_1(q,2) * rlist_1(r,3)\}\ \texttt{zip}\ p\ q\ r\ \{list_1(p,6)\}\ .$$

Answer. Let $R = \exists n,m,k.\,list_1(p,n) * rlist_{\frac{1}{2}}(q,m) * rlist_{\frac{1}{2}}(r,k) * \ulcorner n+m+k = 6 \wedge n \geqslant 1\urcorner$. After `let l create_lock ()` we have $\{l \rightsquigarrow \mathsf{Lock}\,R * rlist_{\frac{1}{2}}(q,2) * rlist_{\frac{1}{2}}(r,3)\}$ which we split into, by the rule for parallel composition `|||`: $\{l \rightsquigarrow \mathsf{Lock}\,R * rlist_{\frac{1}{2}}(q,2)\}$ and $\{l \rightsquigarrow \mathsf{Lock}\,R * rlist_{\frac{1}{2}}(r,3)\}$.

We prove now $\forall m, \{l \rightsquigarrow \mathsf{Lock}\,R * rlist_{\frac{1}{2}}(q,m)\}$ `fill l p q` $\{l \rightsquigarrow \mathsf{Lock}\,R * rlist_{\frac{1}{2}}(q,0)\}$ (the case for k and r being symmetric), by induction on m. The triple holds trivially for $m = 0$. For $m > 0$ we get successively:

	$\{l \rightsquigarrow \mathsf{Lock}\,R * rlist_{\frac{1}{2}}(q,m)\}$
`acquire l;`	$\{l \rightsquigarrow \mathsf{Lock}\,R * R * rlist_{\frac{1}{2}}(q,m)\}$
so $\exists n,k$ s.t. $n+m+k = 6 \wedge n \geqslant 1$ such that, using agreement from Question 2.8:	
	$\{l \rightsquigarrow \mathsf{Lock}\,R * list_1(p,n) * rlist_1(q,m) * rlist_{\frac{1}{2}}(r,k)\}$
By Question 2.9 + frame:	
`transfer_one p r;`	$\{l \rightsquigarrow \mathsf{Lock}\,R * list_1(p,n+1) * rlist_1(q,m-1) * rlist_{\frac{1}{2}}(r,k)\}$
choosing $n = n+1, k = k, m = m-1$:	$\{l \rightsquigarrow \mathsf{Lock}\,R * R * rlist_{\frac{1}{2}}(q,m-1)\}$
`release l;`	$\{l \rightsquigarrow \mathsf{Lock}\,R * rlist_{\frac{1}{2}}(q,m-1)\}$
`fill l p q` (by I.H.)	$\{l \rightsquigarrow \mathsf{Lock}\,R * rlist_{\frac{1}{2}}(q,0)\}$

In the postcondition of the rule for parallel composition we get $\{l \rightsquigarrow \mathsf{Lock}\,R * rlist_{\frac{1}{2}}(q,0) * rlist_{\frac{1}{2}}(r,0))\}$ to which `acquire l` adds R, from which we can conclude by the agreement rule that $m = 0$ and $k = 0$, and so we get

$$\{l \rightsquigarrow \mathsf{Lock}\,R * rlist_1(q,0) * rlist_1(r,0)) * \exists n.\,list_1(p,n) * \ulcorner n+0+0 = 6 \wedge n \geqslant 1\urcorner\}$$

from which we derive the desired postcondition $list_1(p,6)$ with the consequence and GC rules.

Question 2.11. *Above, q and r are references allocated before the creation of the lock. It would also make sense to create the reference after the creation of the lock (for example inside* `fill`*, to keep the reference close to the while loop). What difficulty could we then expect?*

Answer. The lock invariant R would not be able to relate the values pointed by p, q, and r, only by p, and because it is invariant, we would not be able to establish that there are more elements after calling. The common workaround is to use a logic featuring ghost state.

Question 2.12. *Give a lock invariant R that would let us prove the following triple (no proof required):*

$$\{p \rightsquigarrow \mathsf{Mlist}\,[1] * q \mapsto l_q * l_q \rightsquigarrow \mathsf{Mlist}\,[2;3] * r \mapsto l_r * l_r \rightsquigarrow \mathsf{Mlist}\,[4;5]\}$$
$$\texttt{zip}\ p\ q\ r$$
$$\{\exists L.\, p \rightsquigarrow \mathsf{Mlist}\, L * \ulcorner \mathsf{Filter}(\lambda x.x < 4)L = [1;2;3]\urcorner\}\ .$$

Answer. This time we need to be precise about the content of the list pointed by p, which is the interleaving of the elements that have been removed from the list pointed by q and r. To be succinct, let us define the ad-hoc ternary relation S representing the possible values taken by the different lists:

$$\frac{}{S([1],[2;3],[4;5])} \qquad \frac{S(L, x :: L_1, L_2)}{S(L +\!\!+ [x], L_1, L_2)} \qquad \frac{S(L, L_1, x :: L_2)}{S(L +\!\!+ [x], L_1, L_2)}$$

Then we define the lock invariant R as

$$\exists L, l_q, l_r, L_1, L_2.\quad \begin{array}{ll} & p \overset{1}{\rightsquigarrow} \mathsf{Mlist}(1 :: L) \\ * & q \overset{\frac{1}{2}}{\mapsto} l_q * l_q \overset{\frac{1}{2}}{\rightsquigarrow} \mathsf{Mlist}\ L_1' \\ * & r \overset{\frac{1}{2}}{\mapsto} l_r * l_r \overset{\frac{1}{2}}{\rightsquigarrow} \mathsf{Mlist}\ L_2' \\ * & \ulcorner S(L, L_1, L_2)\urcorner \end{array}$$

remark that to call `transfer_one` we need to know that $L \neq []$, which is implied by $S(L, -, -)$. Since S follows tightly the course of the algorithm, not much logical reasoning on lists is needed just until the end. Then, the logical implication $S(L, [], []) \Rightarrow \mathsf{Filter}(\lambda x.x < 4)L = [1;2;3]$ require some reasoning on lists, either by proving results about interleaving, or by enumerating the 19 possible cases for $S(L, L_1, L_2)$.

Question 2.13. *In class, heaps (or heaplets) were initially defined as finite maps from locations to values, and the combination of heaps, $h_1 \uplus h_2$, used to define separation (i.e. in the definition of $*$ and $-\!\!*$) was defined as $h_1 \cup h_2$ under the condition that $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \varnothing$. This definition does not support fractional permissions. Can you give a definition for heaps, for $\uplus$, and for $l \overset{\alpha}{\mapsto} v$, such that it is possible to establish formulas* (3)*? (No proof required, max. 5 lines.)*

Answer. Heaplets can be finite maps $h : L \rightharpoonup V \times [0,1]$ where L are locations and V values ("map" means that $\forall lxyz,\ (x,y) \in h \wedge (x,z) \in h \Rightarrow y = z$). Heaplet combination, $h_1 \uplus h_2$, is only defined under the condition that at the same location, the value is the same and the permission do not exceed one:

$$\forall l \in L,\ l \notin \mathsf{dom}(h_1) \vee l \notin \mathsf{dom}(h_2) \vee \exists x\alpha\beta, h_1(l) = (x,\alpha) \wedge h_2(l) = (x,\beta) \wedge \alpha + \beta \leqslant 1$$

Then $(h_1 \uplus h_2)(l)$ is defined as $(x, \alpha + \beta)$, and $l \overset{\alpha}{\mapsto} v\ \equiv\ \lambda m.m = \{(l,(v,\alpha))\}$.
