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 4 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.*

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.*

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.*

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.*

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.*

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:

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

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

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

Remember that list cells are records with mutable fields `hd` and `tl` such that $p \rightsquigarrow \{|\mathsf{hd} = x; \mathsf{tl} = q|\}$ 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. $p \rightsquigarrow \mathsf{MlistSeg}\, q\, [1, 2]$
2. $p \rightsquigarrow \mathsf{MlistSeg}\, q\, [1] \barwedge \mathsf{GC}$
3. $(1 \mapsto 1) \mathrel{-\!\!*} \mathsf{GC}$
4. $(2 \mapsto 3 * \mathsf{GC}) \barwedge 1 \rightsquigarrow \mathsf{Mlist}\, [1, 2]$

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

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

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

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

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

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.)*

We consider now a function `push_right` : α `cell` $\to \alpha \to$ `unit` with the following specification:

$$\forall L x p, \{\ulcorner P \urcorner * p \rightsquigarrow \mathsf{Mlist}\, L\}\ \texttt{push\_right}\ p\ x\ \{\lambda \_. p \rightsquigarrow \mathsf{Mlist}\,(L +\!\!+ [x])\} \tag{1}$$

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* (1) *can be satisfied (no proof required).*

Question 2.6. *Implement* `push_right` *and prove that it satisfies* (1). *(Be less detailed than in Question 2.4, but stay precise in induction hypotheses and give assertions between all steps. Approx. 15 lines.)*

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).*

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 $0 < \alpha, \beta \leqslant 1$,

$$l \overset{\alpha+\beta}{\mapsto} v \;=\; l \overset{\alpha}{\mapsto} v \;*\; l \overset{\beta}{\mapsto} v \qquad \text{and} \qquad l \overset{\alpha}{\mapsto} v \;*\; l \overset{\beta}{\mapsto} w \;\blacktriangleright\; v = w \;. \tag{2}$$

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.)*

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

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:*

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

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?*

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

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

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* (2)*? (No proof required, max. 5 lines.)*