Final exam, March 3, 2020

- Duration: 3 hours.
- Answers may be written in English or French.
- Allowed documents: lecture notes, personal notes. **Mobile phones must be switched off.** Electronic notes are allowed, but **all network connections must be switched off.**
- There are 7 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 sheet of paper.**

1 Minimum Excluded

In this exercise, we consider the problem of *minimum excluded* which consists, given an array a of integer, 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.*

Answer.

```
invariant { res ↔ (∃ j. 0 ⩽ j < i ∧ a[j] = x) }
```

It is trivially true at the loop entrance, it is preserved by a loop iteration in both cases of the conditional, and at loop exit it directly corresponds to the post-condition to prove.

Digression: Programs with `return` Statements

The code of `check_occur_simple` is inelegant because it uses a local mutable and 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 *not* initially 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.*

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

Answer.

$$\begin{aligned}
(assignment) \quad & \mathrm{WP}(x \leftarrow t, Q, R) = Q[result \leftarrow (); x \leftarrow t] \\
(assert) \quad & \mathrm{WP}(\texttt{assert } A, Q, R) = A \wedge Q \\
(let) \quad & \mathrm{WP}(\texttt{let } x\texttt{=}e_1 \texttt{ in } e_2, Q, R) = \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, R)[result \leftarrow x], R) \\
(return) \quad & \mathrm{WP}(\texttt{return } t, Q, R) = R[result \leftarrow t] \\
(if) \quad & \mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q, R) = \texttt{if } t \texttt{ then } \mathrm{WP}(e_1, Q, R) \texttt{ else } \mathrm{WP}(e_2, Q, R) \\
(function\ call) \quad & \mathrm{WP}(f(\vec{t_i}), Q, R) = Pre_f[x_i \leftarrow t_i] \wedge \forall \vec{v}.(Post_f[x_i \leftarrow t_i, w_j \leftarrow v_j] \rightarrow Q[w_j \leftarrow v_j])
\end{aligned}$$

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?*

Answer. $\forall \vec{x_i} \forall v_i. Pre \rightarrow \mathrm{WP}(body, Post, Post)[w_i@Old \leftarrow w_i][w_i \leftarrow v_i]$

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.*

Answer.

```
invariant { ∀ j. 0 ⩽ j < i → a[j] ≠ x }
```

It is trivially true at the loop entrance, it is preserved by a loop iteration when the conditional is false. At a normal loop exit it directly corresponds to the post-condition to prove. If the returned statement is executed then the post-condition is directly established thanks to the conditional.

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.*

Answer. The minimum excluded is non-negative by definition. If it is not strictly smaller than l, then it means all values between 0 and $l-1$ occurs in a, meaning that all cells of a are filled with these values. Hence l itself cannot occur in a.

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.*

Answer.

```
invariant { all_occur_below min a }
```

It is naturally valid at loop entrance and preserved by a loop iteration that does not return. In case of a return, the first post-condition is naturally true by the loop index constraint, the second is established with the conditional, and the third by the loop invariant. In case of a normal loop exit, the return value `a.length` is OK for the fist post-condition, the second post-condition is true thanks to the general remark above, and the third is again established by the loop invariant.

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.*

Answer. First loop:

```
invariant { ∀ x. 0 ≤ x < n ∧ used[x] → mem x a }
invariant { ∀ j. 0 ≤ j < i ∧ 0 ≤ a[j] < l → used[a[j]] }
```

At the end of that first loop the loop invariants tell us that the array `used` indicates, for each index i, if i occurs in a.
The second loop simply search for the first index of `used` where the value is False.

```
invariant { ∀ j. 0 ≤ j < r → mem j a }
```

---

2 Heap entailment

Reminders Exercises may involve mutable lists. Mutable lists are implemented using cells with a head field and a tail field, and the empty list is represented using the null pointer:

```
type 'a cell = { mutable hd : 'a; mutable tl : 'a cell }  (* or null *)
```

Recall that a mutable linked list is described by the heap predicate $p \rightsquigarrow \mathsf{Mlist}\, L$, where p denotes the location of the first cell (or null), and where L describes the values stored in the head fields of the cells:

$$p \rightsquigarrow \mathsf{Mlist}\, L \quad \equiv \quad \mathsf{match}\, L\, \mathsf{with} \;\; \begin{array}{l} \mid \mathsf{nil} \;\Rightarrow\; [p = \mathsf{null}] \\ \mid x :: L' \;\Rightarrow\; \exists p'.\; p \mapsto \{\mathrm{hd}{=}x;\, \mathrm{tl}{=}p'\} \star p' \rightsquigarrow \mathsf{Mlist}\, L' \end{array}$$

Notations When necessary, you may assume that the offsets for hd and tl are respectively 0 and 1. For conciseness, you may use the notation $p \mapsto \{\!|x; p'|\!\}$ as short for $p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\}$. When the fact that L is a list is absolutely unambiguous, you may write $p \rightsquigarrow L$ instead of $p \rightsquigarrow \mathsf{Mlist}\, L$.
Recall $\curlywedge$ and $\curlyvee$ and $\rhd$. also remind the rules for $\curlyvee$ for *triples*

Question 2.1. *For each heap entailment, 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. $[x = 2] \star [x = 4] \rhd [\mathsf{False}]$
11. $[x = 2] \curlywedge [x = 2] \rhd [\mathsf{False}]$

Question 2.2. *Same question for a second set of entailments:*

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

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 C arrays). We define the following representation predicate for more precision, where p is a pointer and s is a list of characters, s_i is the ith element of the list, $|s|$ is its length, and $\ast$ is the iteration of the separating conjunction ($*$):

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

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

Answer.

$$p \rightsquigarrow \mathsf{String}(\varepsilon) \quad \equiv \quad \mathop{\ast}_{i=0}^{0-1} (p + i) \mapsto s_i \quad \equiv \quad [\mathsf{True}] \quad \equiv \quad [\,]$$

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 it *excludes* s_j, in such a way 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, instead of "...", to complete the specification. Explain carefully how the case $p = \mathsf{null}$ is handled.*

Answer. $0 \leqslant i < n \wedge i+n \leqslant |s|$. If $p = \mathsf{null}$, that means that either $|s| = 0$, in which case it is impossible to satisfy the precondition, because $i + n$ must be at most 0 knowing that $i \geqslant 0$ and $n > 0$, or $|s| > 0$, in which case $p + 0 = \mathsf{null}$ so we have $\mathsf{null} \mapsto \_$, which is also impossible to satisfy.

Question 3.3. *How much time and memory running `substring p i n` takes?*

Answer. $\Theta(n)$ and $\Theta(n)$: it needs to allocate n characters, since there will be n new memory cells in any heap satisfying the postcondition, and copy n characters.

For some purposes, this is just too much for our text editor. Luckily there are much faster operations. The *constant-time and constant-memory* `split_string` function 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 and constant-memory implementation of `split_string`.*

Answer.
```
let split_string i p = p + i
```

Question 3.5. *Show, with details, that `split_string` satisfies its specification.*

Answer. We have p, s, i such that $0 \leqslant i \leqslant |s|$. We need to prove

$$\{p \rightsquigarrow \mathsf{String}(s)\}(\texttt{p + i})\{\lambda p'.p \rightsquigarrow \mathsf{String}(s_{0\ldots i}) * p' \rightsquigarrow \mathsf{String}(s_{i\ldots|s|})\}$$

By the rule for `+`, it is enough to consider that the term is in fact the value $p + i$, and in turn, by the VAL-FRAME rule we need to prove:

$$p \rightsquigarrow \mathsf{String}(s) \rhd p \rightsquigarrow \mathsf{String}(s_{0\ldots i}) * p + i \rightsquigarrow \mathsf{String}(s_{i\ldots|s|})$$

by unfolding we can in fact prove the equality

$$\overset{|s|-1}{\underset{j=0}{\circledast}} (p+j) \mapsto s_j \ = \ \overset{|s_{0\ldots i}|-1}{\underset{j=0}{\circledast}} (p+j) \mapsto (s_{0\ldots i})_j * \overset{|s_{i\ldots|s|}|-1}{\underset{j=0}{\circledast}} ((p+i)+j) \mapsto (s_{i\ldots|s|})_j$$

By rewriting $s_{i\ldots j} = j - i$, $(s_{i\ldots j})_k = s_{i+k}$, associativity of $+$ and index renaming, that gives:

$$\overset{|s|-1}{\underset{j=0}{\circledast}} (p+j) \mapsto s_j \ = \ \overset{i-1}{\underset{j=0}{\circledast}} (p+j) \mapsto s_j * \overset{i+(|s|-i)-1}{\underset{j=i}{\circledast}} (p+j) \mapsto s_j$$

We conclude by associativity of $*$.

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 cell. We write $p \rightsquigarrow \{\!|v_0; v_1; v_2; v_3|\!\}$ for $(p + 0 \mapsto v_0) \star (p + 1 \mapsto v_1) \star (p + 2 \mapsto v_2) \star (p + 3 \mapsto v_3)$ for and we will access those four values from a pointer `p` with the respective field access notations `p.l`, `p.r`, `p.n`, `p.m`. 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 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 $\mathsf{Rope}(s)$ for the representation predicate for a rope that represents a string s, in the way described above.

Question 3.6. *Define a recursive function* `get` *of type* `int → rope → char` *such that* `get i p` *returns the* i*th character of the string represented by the rope pointed by* p*. Write the specification of* `get`*.*

Answer.

```
let rec get (i : int) (p : rope) : char =
  if p.l == null then r.[i] else
    if i < p.m then get i p.l else get (i - p.m) p.r
```

$$\forall p\ i\ s,\ 0 \leqslant i < |s| \ \Rightarrow \{p \rightsquigarrow \mathsf{Rope(s)}\}\ (\texttt{get i p})\ \{\lambda x.[x = s_i] \star p \rightsquigarrow \mathsf{Rope(s)}\}$$

Question 3.7. *Give one equation for* Rope*, in terms of separation logic, of the form* $p \rightsquigarrow \mathsf{Rope}(s) = \ldots$*, that corresponds to the two cases (1) and (2).*

Answer. Forall p and s,

$$\begin{aligned} p \rightsquigarrow \mathsf{Rope(s)} = (\exists p_1 p_2 s_1 s_2.\ & p \rightsquigarrow \{\!|p_1; p_2; |s|; |s_1|\!|\} \\ & \star\ p_1 \rightsquigarrow \mathsf{Rope}(s_1) \\ & \star\ p_2 \rightsquigarrow \mathsf{Rope}(s_2) \\ & \star\ [s = s_1 +\!\!+ s_2]) \\ \curlyvee\ (\exists p'.\ & p \rightsquigarrow \{\!|\mathsf{null}; p'; |s|; -|\!\}\ \star\ p' \rightsquigarrow \mathsf{String}(s)) \end{aligned}$$

We can replace the $-$ with an existentially quantified v.

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?*

Answer. No, it cannot be a recursive definition, since there is nothing that decreases in what would be a recursive call to Rope: we would need to know $\max(|s_1|, |s_2|) < |s|$. It could, however, be an inductive predicate of type `string → Val → heap → Prop`, at the cost of breaking the abstraction of heap predicates. It could also be transformed into a recursive definition by transforming it to a Rope_n predicate, allowing at most n recursive calls, and then defining $p \rightsquigarrow \mathsf{Rope(s)} \equiv \exists n.\, p \rightsquigarrow \mathsf{Rope_n(s)}$. Equivalently, we can define a logical inductive object ropetree corresponding to the memory layout of the rope, and again do $\exists T : \mathsf{ropetree}.\, p \rightsquigarrow \mathsf{Rope}(T) \star [\mathsf{treeRepresentsString}(T, s)]$. A rather dirty solution is to require $\min(|s_1|, |s_2|) > 0$ in the equation, but that would add more edge cases to our functions and proofs (and would not be faithful to the two cases above).

The higher-order representation predicate Quadof, for quadruples, 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.*

Answer.

$$\begin{aligned} p \rightsquigarrow \mathsf{Rope(s)} = \exists s_1 s_2.\ & (\mathsf{Quadof\ String}\ s_1\ \mathsf{String}\ s_2\ \mathsf{Id}\ |s|\ \mathsf{Id}\ |s_1|) \star [s = s_1 +\!\!+ s_2] \\ & \curlyvee\ \mathsf{Quadof\ Id\ null\ String}\ s\ \mathsf{Id}\ |s|\ \mathsf{Any}\ () \end{aligned}$$

with $\mathsf{Id} \equiv \lambda V v.[V = v]$ and $\mathsf{Any} \equiv \lambda() v.[\,]$.

Question 3.10. *What is a useful postcondition for the following triple?*

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

Answer. Several possibilities, of which those two are the best:

- $\{\lambda v_1.(v_1 \rightsquigarrow R_1\ V_1) \star ((v_1 \rightsquigarrow R_1\ V_1) \mathrel{-\!\!\star} (p \rightsquigarrow \mathsf{Quadof}\ R_1\ V_1\ R_2\ V_2\ R_3\ V_3\ R_4\ V_4))\}$
- $\{\lambda v_1.(v_1 \rightsquigarrow R_1\ V_1) \star (p \rightsquigarrow \mathsf{Quadof}\ \mathsf{Id}\ v_1\ R_2\ V_2\ R_3\ V_3\ R_4\ V_4)\}$

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    ; n = i      ; m = 2019},
     {l = null; r = p + 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`*.*

Answer.

$$\forall p\ s\ i \quad 0 \leqslant i \leqslant |s| \ \Rightarrow\ \{p \rightsquigarrow \mathsf{Rope}(s)\}(\texttt{split\_rope i p})\{\lambda(p_1, p_2).p_1 \rightsquigarrow \mathsf{Rope}(s_{0\ldots i}) * p_2 \rightsquigarrow \mathsf{Rope}(s_{i\ldots|s|})\}$$

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.)*

Answer. We suppose we chose the Rope_n representation predicate and we do a strong induction on n. Suppose we have by induction

$$\{p \rightsquigarrow \mathsf{Rope}_n(s)\}(\texttt{split\_rope i p})\{\lambda(p_1, p_2).p_1 \rightsquigarrow \mathsf{Rope}(s_{0\ldots i}) * p_2 \rightsquigarrow \mathsf{Rope}(s_{i\ldots|s|})\}$$

for all $p, s, i \in \{0, \ldots, |s|\}$ and $n' < n$. Note that we do not need to refine Rope into a Rope_n on the right-hand side. We apply the rule for `if` twice, which results in three triples to prove, of which we prove the second:

$$\{p \rightsquigarrow \mathsf{Rope}_n(s)\}(\texttt{split\_rope i p})\{\lambda(p_1, p_2).p_1 \rightsquigarrow \mathsf{Rope}(s_{0\ldots i}) * p_2 \rightsquigarrow \mathsf{Rope}(s_{i\ldots|s|})\}$$

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*?*

Answer.

$$\begin{aligned} p \rightsquigarrow \mathsf{Quadof}'\ Q_1\ Q_2\ Q_3\ Q_4 \equiv \exists v_1 v_2 v_3 v_4.\ & p \rightsquigarrow \{\!|v_1; v_2; v_3; v_4|\!\} \\ & \star\ v_1 \rightsquigarrow Q_1 \\ & \star\ v_2 \rightsquigarrow Q_2 \\ & \star\ v_3 \rightsquigarrow Q_3 \\ & \star\ v_4 \rightsquigarrow Q_4 \end{aligned}$$

This would not work for Mlistof, since the Q would need to be different at every recursive call, since L is different every time. However, this would be fine for Mcellof.
