# Final exam, March 1st, 2017

- Duration: 3 hours. There are **4** independent exercises.
- Allowed documents: lecture notes, personal notes. **Mobile phones must be switched off.** Electronic notes are allowed, but **all network connections must be switched off.**
- Answers may be written in English or French.
- Unless specified otherwise, universal quantification of free variables at top-level may be left implicit. However, existential quantification should always be explicit.

**Reminder** Exercises 2, 3 and 4 involve mutable lists. Recall that mutable lists are implemented using cells with a head and a tail field, and that 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. A list segment from $p$ (inclusive) to $q$ (exclusive) is described by the predicate $p \rightsquigarrow \mathsf{MlistSeg}\,q\,L$.

$$\begin{array}{lcl} p \rightsquigarrow \mathsf{Mlist}\,L & \equiv & p \rightsquigarrow \mathsf{MlistSeg}\,\mathsf{null}\,L \\ p \rightsquigarrow \mathsf{MlistSeg}\,q\,L & \equiv & \mathsf{match}\,L\,\mathsf{with}\ \mid \mathsf{nil} \Rightarrow [p = q] \\ & & \phantom{\mathsf{match}\,L\,\mathsf{with}}\ \mid x :: L' \Rightarrow \exists p'.\ p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\} \star p' \rightsquigarrow \mathsf{MlistSeg}\,q\,L' \end{array}$$

For conciseness, you may use the notation $p \mapsto \{\!|x; p'|\!\}$ as short for $p \mapsto \{\!|\mathrm{hd}{=}x;\ \mathrm{tl}{=}p'|\!\}$.

## 1 Polynomials

A polynomial of degree $n$ with real coefficients is of the form $\sum_{i=0}^{n} c_i X^i$. Consider a representation of such a polynomial as an array of real numbers. The array has length $n+1$, and the $i$-th cell of the array stores the coefficient $c_i$ associated with the term $X^i$. For example, the polynomial $X^3 + 4X - 7$ is represented as the array $[-7; 4; 0; 1]$. The function `eval` formally interprets an array of reals as a polynomial. It is defined as follows.

```
function eval (p:array real) (x:real) : real = eval_aux p x 0 p.length
```

where `eval_aux` is axiomatized by:

- $\forall\ p\ x\ i\ j.\ j \leqslant i \rightarrow \texttt{eval\_aux}\ p\ x\ i\ j = 0.0$
- $\forall\ p\ x\ i\ j.\ i < j \rightarrow \texttt{eval\_aux}\ p\ x\ i\ j = p[i] + x \times \texttt{eval\_aux}\ p\ x\ (i+1)\ j$

**Adding a constant to a polynomial** The function `add_const` adds a constant to a polynomial. It is specified and implemented as follows.

```
let add_const (p:array real) (c:real) : unit
  requires { p.length ⩾ 1 }
  writes { p }
  ensures { forall x. eval p x = eval (old p) x + c }
= p[0] ← p[0] + c
```

**Question 1.1.** As such, this program cannot be proved automatically: because a lemma is needed as a hint for automatic provers. State this lemma and explain why it is needed.

**Answer.** Let $p'$ denote the array after the assignment. Proving the post-condition requires to establish:

```
eval p' x = eval p x + c
```

By unfolding `eval`:

```
eval_aux p' x 0 l = eval_aux p x 0 l + c
```

By expanding using the second axiom of `eval_aux`:

```
p'[0] + eval_aux p' x 1 l = p[0] + eval_aux p x 1 l + c
```

After simplification:

```
eval_aux p' x 1 l = eval_aux p x 1 l
```

To prove that `p'` is equal to `p` on the range `1..l`, a frame lemma is needed: for any arrays $p$ and $q$, if $\forall k.\ i \leqslant k < j \rightarrow p[k] = q[k]$, then $\texttt{eval\_aux}\, p\, x\, i\, j = \texttt{eval\_aux}\, q\, x\, i\, j$.

**Question 1.2.** State the lemma above as a lemma function, and propose a implementation of that function that allows to automatically prove the lemma.

**Answer.** A lemma function can be stated as follows to enforce a proof by induction on $j - i$:

```
let rec lemma eval_frame (p q:array real) (x:real) (i j:int)
  requires { forall k. i ⩽ k < j → p[k] = q[k] }
  variant { j - i }
  ensures { eval_aux p x i j = eval_aux q x i j }
= if j > i then eval_frame p q x (i+1) j
```

**Addition of two polynomials** The addition of polynomials is specified and implemented as follows, by a function that stores the result in place in the first argument. For simplicity, we assume that the polynomials given as arguments have the same degree, thus the two arrays have the same length.

```
let add (p q:array real)
  requires { p.length = q.length }
  writes { p }
  ensures { forall x. eval p x = eval (old p) x + eval q x }
= for i = p.length - 1 downto 0 do
     p[i] ← p[i] + q[i]
  done
```

**Question 1.3.** Propose a loop invariant for this code. Explain why this loop invariant is sufficient to prove the post-condition.

**Answer.** A natural invariant is as follows:

```
invariant { forall x. eval_aux p x (i+1) p.length =
                 eval_aux (at p 'L) x (i+1) p.length + eval_aux q x (i+1) p.length }
```

where the label `L` denotes the entry of the program. When the loop terminates, the invariant is assumed true for the next index, that is, for -1. Thus, the post-condition follows from the definition of eval.

**Question 1.4.** Prove that the loop invariant proposed is preserved by the loop body. Explain carefully the reasoning steps.

**Answer.** To prove preservation, the loop invariant needs to be strengthened to assert that elements in the array $p$ at index less than $i$ are not modified.

```
invariant { forall k. 0 ⩽ k ⩽ i → p[k] = (at p 'L)[k] }
```

Proving this second invariant is trivial, since the loop body modifies only `p[i]`. Proving the first loop invariant can then be done, using again the frame lemma. For any $x$, we have:

```
eval_aux p x i p.length
= p[i] + x * eval_aux p x (i+1) p.length (* axiom for eval_aux *)
= (at p 'M)[i] + q[i]
  + x * (eval_aux p x (i+1) p.length)  (* effect of the loop body *)
= (at p 'M)[i] + q[i]
  + x * (eval_aux (at p 'M) x (i+1) p.length)  (* frame lemma *)
= (at p 'M)[i] + q[i]
  + x * (eval_aux (at p 'L) x (i+1) p.length + eval_aux q x (i+1) p.length)
                                        (* the assumed loop invariant *)
= (at p 'L)[i] + q[i]
  + x * (eval_aux (at p 'L) x (i+1) p.length + eval_aux q x (i+1) p.length)
                                        (* the assumed second loop invariant *)
= (at p 'L)[i] + x * eval_aux (at p 'L) x (i+1) p.length
  + (q[i] + x * eval_aux q x (i+1) p.length)
```

```
                    (* distributivity      *)
   = eval_aux (at p ’L) x i p.length
                  + eval_aux q x i p.length
                    (* axiom for eval_aux *)
```

where label M denotes the entry of the loop body

**Question 1.5.** The given code for addition uses a downward loop. What would happen if it was implemented with a forward loop `for i = 0 to p.length - 1 do ...` ?

**Answer.** The code would be correct, with the loop invariants

```
  invariant { forall x. eval_aux p x 0 i =
                eval_aux (at p ’L) x 0 i + eval_aux q x 0 i }
  invariant { forall k. i ⩽ k < p.length → p[k] = (at p ’L)[k] }
```

but to establish preservation of the first invariant, an additional lemma is needed:

```
   eval p x i (j+1) = eval p x i j + x^{j-i} * p[j]
```

## 2 Mutable iterators in Separation Logic

We consider here a particular representation of a mutable iterator over mutable lists. Such a mutable iterator keeps a pointer on a particular cell from the list. It provides direct read and write access to this cell. It also provides a method for stepping to the next item in the list, and a method to test if the end of the list has been reached. An iterator is initially constructed from the pointer on the head of the list. We will consider here a specification in which the iterator owns the entire list while traversing it.

### Implementation of mutable iterators

```
   type ’a iter = (’a cell) ref
   let mk_iter (p:’a cell) : ’a iter = ref p
   let get (i:’a iter) : ’a = (!i.hd)
   let set (i:’a iter) (v:’a) : unit = (!i.hd <- v)
   let is_done (i:’a iter) : bool = (!i == null)
   let next (i:’a iter) : unit = (i := !i.tl)
```

**Question 2.1.** Define a representation predicate of the form $i \rightsquigarrow \mathsf{Miter}\, p\, L_1\, L_2$ where $L_1$ describes the contents of the cells already traversed by an iterator $i$ that started from a pointer $p$, and where $L_2$ describes the cells remaining to be traversed by the iterator. Note that $L_2$ is empty when the iterator has reached the end of the list. The predicate $i \rightsquigarrow \mathsf{Miter}\, p\, L_1\, L_2$ should capture the fact that the iterator owns not just the reference cell that implements it, but also owns all the cells in the list. Hint: a list segment is needed.

**Answer.**

$$i \rightsquigarrow \mathsf{Miter}\, p\, L_1\, L_2 \quad \equiv \quad \exists q.\ i \mapsto q \,\star\, p \rightsquigarrow \mathsf{MlistSeg}\, q\, L_1 \,\star\, q \rightsquigarrow \mathsf{Mlist}\, L_2$$

**Question 2.2.** Prove the following specification below. (Expected answer: 4 lines.)

$$\{p \rightsquigarrow \mathsf{Mlist}\, L\}\ (\texttt{mk\_iter})\ \{\lambda i.\ i \rightsquigarrow \mathsf{Miter}\, p\, \mathsf{nil}\, L\}.$$

**Answer.** After a call to `mk_iter`, we have the state: $i \mapsto p \star p \rightsquigarrow \mathsf{Mlist}\, L$. We obtain the post-condition by taking $q = p$ and exploiting $(p \rightsquigarrow \mathsf{MlistSeg}\, p\, \mathsf{nil}) = [\,]$.

**Question 2.3.** Prove the following heap entailment. What is its purpose? (Expected answer: 5 lines.)

$$i \rightsquigarrow \mathsf{Miter}\, p\, L_1\, L_2 \ \ \rhd\ \ p \rightsquigarrow \mathsf{Mlist}\, (L_1 +\!\!+ L_2)$$

**Answer.** For the heap entailment, we exploit the concatenation of list segments, and we discard the iterator cell $i \mapsto q$. This rule is useful for interrupting the iteration process, possibly before completion, and reclaiming the ownership of the list.

**Question 2.4.** Give a specification for `get` and one for `set`, stated in terms of Miter.

**Answer.**

$$\{i \rightsquigarrow \mathsf{Miter}\, p\, L_1\, (v :: L_2)\}\ (\texttt{get i})\ \{\lambda x.\ [x = v] \star i \rightsquigarrow \mathsf{Miter}\, p\, L_1\, (v :: L_2)\}$$

$$\{i \rightsquigarrow \mathsf{Miter}\, p\, L_1\, (w :: L_2)\}\ (\texttt{set i v})\ \{\lambda\_.\ i \rightsquigarrow \mathsf{Miter}\, p\, L_1\, (v :: L_2)\}$$

**Question 2.5.** Give a specification for `is_done` stated in terms of $\mathsf{Miter}$. Argue in two 3 lines of english why the code satisfies your specification.

**Answer.**

$$\{i \leadsto \mathsf{Miter}\, p\, L_1\, L_2\}\ (\texttt{is\_done i})\ \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow L_2 = \mathsf{nil}] \star i \leadsto \mathsf{Miter}\, p\, L_1\, L_2\}$$

The code of `is_done` tests whether the contents of $i$, namely $q$ from the invariant, is null. Given the heap predicate $q \leadsto \mathsf{Mlist}\, L_2$, this test determines whether $L_2 = \mathsf{nil}$.

**Question 2.6.** Give a specification for `next` stated in terms of $\mathsf{Miter}$. Prove that the code satisfies your specification. (Expected answer: 6 lines.)

**Answer.**

$$\{i \leadsto \mathsf{Miter}\, p\, L_1\, (x :: L_2)\}\ (\texttt{next i})\ \{\lambda\_.\ i \leadsto \mathsf{Miter}\, p\, (L_1 \& x)\, L_2\}$$

Unfolding the pre-condition: $i \mapsto q \star p \leadsto \mathsf{MlistSeg}\, q\, L_1 \star q \leadsto \mathsf{Mlist}\, (x :: L_2)$, for some $q$.
Unfolding the second list: $i \mapsto q \star p \leadsto \mathsf{MlistSeg}\, q\, L_1 \star q \mapsto \{|\mathrm{hd}{=}x;\ \mathrm{tl}{=}q'|\} \star q' \leadsto \mathsf{Mlist}\, L_2$.
Folding the first list and executing the assignment: $i \mapsto q' \star p \leadsto \mathsf{MlistSeg}\, q\, (L_1 \& x) \star q' \leadsto \mathsf{Mlist}\, L_2$.
Folding-the post-condition by instantiating $q$ with $q'$ concludes the proof.

Remark: the following specification is also correct:

$$\{[L_2 \neq \mathsf{nil}] \star i \leadsto \mathsf{Miter}\, p\, L_1\, L_2\}\ (\texttt{next i})\ \{\lambda\_.\ \exists L_2'.\, [L_2 = x :: L_2'] \star i \leadsto \mathsf{Miter}\, p\, (L_1 \& x)\, L_2'\}$$

**Question 2.7** (Difficult)**.** Give a specification to the function `reach` defined below, assuming $f$ to be a pure function.

```
let reach (i:'a iter) (f:'a->bool) : unit =
  while !i != null && not (f !i.hd) do
    i := !i.tl;
  done
```

**Answer.**

$$\begin{array}{ll} \forall f i p L_1 L_2 P. & (\forall x.\ \{[\,]\}\ (f\, x)\ \{\lambda b.\, [b = \mathsf{true} \Leftrightarrow P\, x]\}) \\ & \{i \leadsto \mathsf{Miter}\, p\, L_1\, L_2\}\ (\texttt{reach i f})\ \{\lambda\_.\ \exists L_3 L_4.\quad \begin{array}[t]{l} i \leadsto \mathsf{Miter}\, p\, (L_1 \mathbin{+\!\!+} L_3)\, L_4 \\ \star\ [L_2 = L_3 \mathbin{+\!\!+} L_4] \\ \star\ [\mathsf{Forall}\, (\neg P)\, L_3] \\ \star\ [\forall x L_5.\ L_4 = x :: L_5 \Rightarrow P\, x] \end{array} \quad \} \end{array}$$

# 3 List concatenation in Separation Logic

In this exercise, we study three possible implementations of a concatenation function for mutable lists.

### Recursive implementation

```
let mappend (p1:'a cell) (p2:'a cell) : unit =
  if p1.tl == null
    then p1.tl <- p2
    else mappend p1.tl p2
```

**Question 3.1.** Give a formal specification to `mappend` stated using the representation predicate $p \leadsto \mathsf{Mlist}\, L$, expressing the fact that the function performs in-place concatenation of two mutable lists, updating its first argument and consuming its second argument. Make sure to include the pre-condition required to ensure safety.

**Answer.**

$$\begin{array}{ll} \forall p_1 p_2 L_1 L_2. & \{[p_1 \neq \mathsf{null}] \star p_1 \leadsto \mathsf{Mlist}\, L_1 \star p_2 \leadsto \mathsf{Mlist}\, L_2\} \\ & (\texttt{mappend p1 p2}) \\ & \{\lambda\_.\ p_1 \leadsto \mathsf{Mlist}\, (L_1 \mathbin{+\!\!+} L_2)\} \end{array}$$

Remark: it is equivalent to assert $[L_1 \neq \mathsf{null}]$ as pre-condition, or to consider a pre-condition of the form $p_1 \leadsto \mathsf{Mlist}\, (x :: L_1) \star p_2 \leadsto \mathsf{Mlist}\, L_2$ with the post-condition $p_1 \leadsto \mathsf{Mlist}\, (x :: L_1 \mathbin{+\!\!+} L_2)$.

**Question 3.2.** Prove that the implementation of `mappend` satisfies the specification claimed in the previous question. (Expected answer: 15 lines.)

**Answer.** By induction on $L_1$. Since $p_1 \neq \mathsf{null}$, we can decompose $L_1 = x :: L'_1$ and rewrite the pre-condition as:

$$p_1 \mapsto \{\!|\text{hd}{=}x;\ \text{tl}{=}p'_1|\!\} \star p'_1 \rightsquigarrow \mathsf{Mlist}\, L'_1 \star p_2 \rightsquigarrow \mathsf{Mlist}\, L_2.$$

Applying the reasoning rules for if-statements, we distinguish two cases.

- Case $p'_1 = \mathsf{null}$. The heap predicate $p'_1 \rightsquigarrow \mathsf{Mlist}\, L'_1$ is equivalent to $[\,]$, and indicates $L'_1 = \mathsf{nil}$. The assignment on the tail of $p_1$ leads to the state:

  $$p_1 \mapsto \{\!|\text{hd}{=}x;\ \text{tl}{=}p_2|\!\} \star p_2 \rightsquigarrow \mathsf{Mlist}\, L_2.$$

  By folding the representation predicate for lists, we obtain $p_1 \rightsquigarrow \mathsf{Mlist}\,(x :: L_2)$. This matches the post-condition $p_1 \rightsquigarrow \mathsf{Mlist}\,(L_1 \mathbin{+\!\!+} L_2)$, since $L_1 \mathbin{+\!\!+} L_2 = (x :: L'_1) \mathbin{+\!\!+} L_2 = (x :: \mathsf{nil}) \mathbin{+\!\!+} L_2 = x :: L_2$.

- Case $p'_1 \neq \mathsf{null}$. We frame $p_1 \mapsto \{\!|\text{hd}{=}x;\ \text{tl}{=}p'_1|\!\}$, and apply the induction hypothesis, which is:

  $$\{[p'_1 \neq \mathsf{null}] \star p'_1 \rightsquigarrow \mathsf{Mlist}\, L'_1 \star p_2 \rightsquigarrow \mathsf{Mlist}\, L_2\}\ (\texttt{mappend p1' p2})\ \{\lambda\_.\ p'_1 \rightsquigarrow \mathsf{Mlist}\,(L'_1 \mathbin{+\!\!+} L_2)\}.$$

  The final state is: $p_1 \mapsto \{\!|\text{hd}{=}x;\ \text{tl}{=}p'_1|\!\} \star p'_1 \rightsquigarrow \mathsf{Mlist}\,(L'_1 \mathbin{+\!\!+} L_2)$. By folding the representation predicate for lists, we obtain $p_1 \rightsquigarrow \mathsf{Mlist}\,(x :: (L_1 \mathbin{+\!\!+} L_2))$. This matches the post-condition $p_1 \rightsquigarrow \mathsf{Mlist}\,(L_1 \mathbin{+\!\!+} L_2)$, since $L_1 \mathbin{+\!\!+} L_2 = (x :: L'_1) \mathbin{+\!\!+} L_2 = x :: (L'_1 \mathbin{+\!\!+} L_2)$.

**Iterative implementation**

```
let mappend' (p1:'a cell) (p2:'a cell) : unit =
  let f = ref p1 in
  while !f.tl != null do
    f := !f.tl;
  done;
  !f.tl <- p2
```

**Question 3.3.** Give the loop invariant that holds at the beginning of every iteration of the while loop above. Discuss whether the description of the list $p_2$ needs to be mentioned or not in the loop invariant. Make sure to quantify all variables appropriately in the invariant.

**Answer.**

$$\exists q L_a L_b.\ f \mapsto q\ \ \star p_1 \rightsquigarrow \mathsf{MlistSeg}\, q\, L_a \star q \rightsquigarrow \mathsf{Mlist}\, L_b \star [L_1 = L_a \mathbin{+\!\!+} L_b] \star [q \neq \mathsf{null}]$$

The list at address $p_2$ needs not be part of the invariant because the corresponding list is not manipulated by the loop. The heap predicate $p_2 \rightsquigarrow \mathsf{Mlist}\, L_2$ may be framed during the reasoning on the while loop. Note that $[q \neq \mathsf{null}]$ could be replaced with $[L_b \neq \mathsf{nil}]$. Furthermore, using the predicate $i \rightsquigarrow \mathsf{Miter}\, p\, L_1\, L_2$ from the previous exercise, one could reformulate more concisely the invariant as follows:

$$\exists L_a L_b.\ f \rightsquigarrow \mathsf{Miter}\, p_1\, L_a\, L_b \star [L_1 = L_a \mathbin{+\!\!+} L_b] \star [L_b \neq \mathsf{nil}].$$

**Question 3.4.** Give the state before the loop, and show that it entails the invariant.

**Answer.** The state before the loop is:

$$f \mapsto p_1\ \ \star p_1 \rightsquigarrow \mathsf{Mlist}\, L_1 \star [L_1 = L_a \mathbin{+\!\!+} L_b] \star [p_1 \neq \mathsf{null}] \star p_2 \rightsquigarrow \mathsf{Mlist}\, L_2$$

To establish the invariant: $q = p_1$ and $L_a = \mathsf{nil}$ and $L_b = L_1$, and exploit $(p_1 \rightsquigarrow \mathsf{MlistSeg}\, p_1\, \mathsf{nil}) = [\,]$. The list $p_2$ is framed.

**Question 3.5.** Explain how the loop invariant is preserved at each iteration.

**Answer.** Assume the existence of $q$, $L_a$, and $L_b$ such that the state is:

$$f \mapsto q\ \ \star p_1 \rightsquigarrow \mathsf{MlistSeg}\, q\, L_a \star q \rightsquigarrow \mathsf{Mlist}\, L_b \star [L_1 = L_a \mathbin{+\!\!+} L_b] \star [q \neq \mathsf{null}].$$

Since $q \neq \mathsf{null}$, we may decompose $L_b$ as $x :: L'_b$ and:

$$f \mapsto q\ \ \star p_1 \rightsquigarrow \mathsf{MlistSeg}\, q\, L_a \star q \mapsto \{\!|x; q'|\!\} \star q' \rightsquigarrow \mathsf{Mlist}\, L'_b \star [L_1 = L_a \mathbin{+\!\!+} (x :: L'_b)].$$

If the loop condition succeeds, then $q'$ is not null, so we assign $f$ to $q'$ and may rearrange the state as:

$$f \mapsto q'\ \ \star p_1 \rightsquigarrow \mathsf{MlistSeg}\, q'\, (L_a \& x) \star q' \rightsquigarrow \mathsf{Mlist}\, L'_b \star [L_1 = (L_a \& x) \mathbin{+\!\!+} L'_b] \star [q' \neq \mathsf{null}].$$

This establishes the invariant with $q$ set as $q'$ and $L_a$ set as $(L_a \& x)$ and $L_b$ set as $L'_b$.

**Question 3.6.** At the end of the loop, the invariant holds and `!f.tl` is null; give the state just after performing the operation `!f.tl` $\leftarrow$ `p2`, to show that the post-condition can be derived from this state.

**Answer.** The state is:

$$f \mapsto q \;\; \star \; p_1 \rightsquigarrow \mathsf{MlistSeg}\, q\, L_a \; \star \; q \rightsquigarrow \mathsf{Mlist}\, L_b \; \star \; [L_1 = L_a +\!\!+ L_b] \; \star \; [q \neq \mathsf{null}] \; \star \; p_2 \rightsquigarrow \mathsf{Mlist}\, L_2$$

From $p_1 \neq \mathsf{null}$ and $\texttt{q.tl} = \mathsf{null}$, we deduce that $L_b$ is of the form $x :: \mathsf{nil}$. Thus the state is:

$$f \mapsto q \;\; \star \; p_1 \rightsquigarrow \mathsf{MlistSeg}\, q\, L_a \; \star \; q \mapsto \{\!|x; \mathsf{null}|\!\} \; \star \; [L_1 = (L_a \& x)] \; \star \; p_2 \rightsquigarrow \mathsf{Mlist}\, L_2.$$

Assigning the tail of $q$ to $p_2$ gives:

$$f \mapsto q \;\; \star \; p_1 \rightsquigarrow \mathsf{MlistSeg}\, q\, L_a \; \star \; q \mapsto \{\!|x; p_2|\!\} \; \star \; [L_1 = (L_a \& x)] \; \star \; p_2 \rightsquigarrow \mathsf{Mlist}\, L_2.$$

Discarding $f \mapsto q$ using rule GC, and concatenating the 3 pieces of lists indeed gives the post-condition:

$$p_1 \rightsquigarrow \mathsf{Mlist}\, (L_1 +\!\!+ L_2).$$

### Alternative iterative implementation

```
let mappend'' (p1:'a cell) (p2:'a cell) : unit =
  let f = ref p1 in
  while true do
    if !f.tl == null then begin
      !f.tl <- p2;
      break;
    end;
    f := !f.tl;
  done
```

**Question 3.7.** Interestingly, the alternative implementation can be proved correct without involving any list segments, thanks to the frame rule. State a judgement of the form "$\forall \ldots.,\ \{\ldots\}\ t\ \{\lambda\_.\ \ldots\}$" that one could use to prove by induction the correctness of the function, where $t$ denotes the entire while loop, from `while` to `done`, inclusive.

**Answer.** We may prove by induction on $L_b$ that:

$$\begin{array}{ll} \forall q L_b. & \{f \mapsto q \star [q \neq \mathsf{null}] \star q \rightsquigarrow \mathsf{Mlist}\, L_b \star p_2 \rightsquigarrow \mathsf{Mlist}\, L_2\} \\ & (\texttt{while true do ... done}) \\ & \{\lambda\_.\ q \rightsquigarrow \mathsf{Mlist}\, (L_b +\!\!+ L_2)\} \end{array}$$

**Question 3.8.** The code of `mappend''` has exactly the same semantics as the code of `mappend'`. But, thanks to its slightly different presentation, the code of `mappend''` may be proved correct without involving list segments, whereas `mappend'` does not offer this possibility. So, during the verification proof of `mappend'`, we could rewrite its code using a program transformation rule that would allow to change it on-the-fly to `mappend''`, and thereby complete the proof in a simpler way. State the corresponding program transformation rule in the form of a general equation between two terms.

**Answer.**

$$((\mathsf{while}\, t_1\, \mathsf{do}\, t_2)\, ;\, t_3) \;=\; (\mathsf{while\, true\, do}\, (\mathsf{if}\, t_1\, \mathsf{then}\, t_2\, \mathsf{else}\, (t_3\, ;\, \mathsf{break})))$$

## 4 List comparison in Separation Logic

This exercise investigates the specification of comparison functions over mutable lists.

### Implementation for mutable lists of integers

```
let rec mlist_cmp_int (p1:int cell) (p2:int cell) : bool =
  if (p1 == null) then (p2 == null)
  else if (p2 == null) then false
  else if (p1.hd <> p2.hd) then false
  else mlist_cmp_int p1.tl p2.tl
```

**Question 4.1.** Give a specification `mlist_cmp_int` expressing the fact that this function expects as arguments two disjoint mutable lists, which are preserved by the function, and returns a boolean value indicating whether the two structures describe exactly the same list of integers.

**Answer.**

$$\begin{array}{l} \forall p_1 p_2 L_1 L_2.\ \{p_1 \leadsto \mathsf{Mlist}\, L_1 \star p_2 \leadsto \mathsf{Mlist}\, L_2\} \\ \qquad (\texttt{mlist\_cmp\_int}\, p_1\, p_2) \\ \qquad \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow L_1 = L_2] \star p_1 \leadsto \mathsf{Mlist}\, L_1 \star p_2 \leadsto \mathsf{Mlist}\, L_2\} \end{array}$$

**Question 4.2.** Argue for the correctness of last line of `mlist_cmp_int`, i.e. in the case where the two lists are nonempty and have the same head. values. To that end, describe precisely the frame process associated with the recursive call, by stating the states before and after unfolding the representation predicate for lists, and stating the state before and after the recursive call.

**Answer.**

| | |
|---|---|
| $p_1 \leadsto \mathsf{Mlist}\, L_1 \star p_2 \leadsto \mathsf{Mlist}\, L_2$ | pre-condition |
| $p_1 \mapsto \{\!\lvert x; p_1' \rvert\!\} \star p_2 \mapsto \{\!\lvert x; p_2' \rvert\!\} \star p_1' \leadsto \mathsf{Mlist}\, L_1' \star p_2' \leadsto \mathsf{Mlist}\, L_2'$ | by unfolding |
| $p_1' \leadsto \mathsf{Mlist}\, L_1' \star p_2' \leadsto \mathsf{Mlist}\, L_2'$ | frame begins |
| $p_1' \leadsto \mathsf{Mlist}\, L_1' \star p_2' \leadsto \mathsf{Mlist}\, L_2' \star [b = \mathsf{true} \Leftrightarrow L_1' = L_2']$ | by induction |
| $p_1 \mapsto \{\!\lvert x; p_1' \rvert\!\} \star p_2 \mapsto \{\!\lvert x; p_2' \rvert\!\} \star p_1' \leadsto \mathsf{Mlist}\, L_1' \star p_2' \leadsto \mathsf{Mlist}\, L_2' \star [b = \mathsf{true} \Leftrightarrow L_1' = L_2']$ | frame ends |
| $p_1 \leadsto \mathsf{Mlist}\, L_1 \star p_2 \leadsto \mathsf{Mlist}\, L_2 \star [b = \mathsf{true} \Leftrightarrow x :: L_1' = x :: L_2']$ | post-condition |

**Question 4.3.** In this question, we consider the extension of Separation Logic with the read-only construct $\mathsf{RO}(H)$. In that setting, give a specification of `mlist_cmp_int`. In addition to improved conciseness, what is the major benefits of this specification over the previous one?

**Answer.**

$$\begin{array}{l} \forall p_1 p_2 L_1 L_2.\ \{\mathsf{RO}(p_1 \leadsto \mathsf{Mlist}\, L_1) \star \mathsf{RO}(p_2 \leadsto \mathsf{Mlist}\, L_2)\} \\ \qquad (\texttt{mlist\_cmp\_int}\, p_1\, p_2) \\ \qquad \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow L_1 = L_2]\} \end{array}$$

This specification supports reasoning about calls to `mlist_cmp_int` for comparing a list with itself.

### Implementation for polymorphic lists

```
let rec mlist_cmp (f:'a->'a->bool) (p1:'a cell) (p2:'a cell) : bool =
  if (p1 == null) then (p2 == null)
  else if (p2 == null) then false
  else if (not (f p1.hd p2.hd)) then false
  else mlist_cmp p1.tl p2.tl
```

**Question 4.4.** Give a specification for `mlist_cmp`. You may assume that the comparison function $f$ is pure and that it returns a boolean indicating whether its arguments are equal. Make sure to quantify all variables precisely.

**Answer.**

$$\begin{array}{lll} \forall p_1 p_2 L_1 L_2. & & \big(\forall x_1 x_2.\ \{[\,]\}\ (f\, x_1\, x_2)\ \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow x_1 = x_2]\}\big) \\ & \Rightarrow & \{p_1 \leadsto \mathsf{Mlist}\, L_1 \star p_2 \leadsto \mathsf{Mlist}\, L_2\} \\ & & (\texttt{mlist\_cmp}\, f\, p_1\, p_2) \\ & & \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow L_1 = L_2] \star p_1 \leadsto \mathsf{Mlist}\, L_1 \star p_2 \leadsto \mathsf{Mlist}\, L_2\} \end{array}$$

**Question 4.5.** Give a generalized specification for `mlist_cmp`, with a pre-condition expressed using the higher-order representation predicate $p \leadsto \mathsf{Mlistof}\, R\, L$, thereby specifying the comparison of two mutable lists storing mutable elements.
Remark: to avoid duplicating pre-conditions inside post-conditions, you may use, if you wish to, the read-only construct $\mathsf{RO}(H)$ from the extension of Separation Logic with read-only permissions.

**Answer.**

$$\begin{array}{l} \forall R p_1 p_2 L_1 L_2. \\ \quad \big(\forall x_1 x_2 X_1 X_2.\ \{\mathsf{RO}(x_1 \leadsto R\, X_1) \star \mathsf{RO}(x_2 \leadsto R\, X_2)\}\ (f\, x_1\, x_2)\ \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow X_1 = X_2]\}\big) \\ \Rightarrow \{\mathsf{RO}(p_1 \leadsto \mathsf{Mlistof}\, R\, L_1) \star \mathsf{RO}(p_2 \leadsto \mathsf{Mlistof}\, R\, L_2)\}\ (\texttt{mlist\_cmp}\, f\, p_1\, p_2)\ \{\lambda b.\ [b = \mathsf{true} \Leftrightarrow L_1 = L_2]\} \end{array}$$