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 $p \rightsquigarrow \mathsf{MlistSeg}\, q\, L$.

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

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.

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.

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.

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

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 ...` ?

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.

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

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

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

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

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

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

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

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
```

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 $p \rightsquigarrow \mathsf{Mlist}\, L$, 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.

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

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.

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

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

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.

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 "$\forall ....,\ \{...\}\ t\ \{\lambda\_ .\ ...\}$" 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.

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.

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.

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.

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?

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.

Question 4.5. Give a generalized specification for `mlist_cmp`, with a pre-condition expressed using the higher-order representation predicate $p \rightsquigarrow \mathsf{Mlistof}\, R\, L$, 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.