Final exam, March 12th, 2015

- Duration: 3 hours. There are **3** independent exercises.
- Allowed documents: lecture notes, personal notes, **but no electronic devices.**
- **Mobile phones 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.

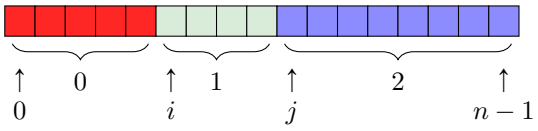
1 Sorting a 3-valued Array

In this exercise, we consider arrays of integers that are *3-valued*: the values stored are either 0, 1 or 2. We propose a *linear* algorithm to sort 3-valued arrays in increasing order. Recall that array indices start from 0 and that `a.length` denotes the length of the array `a`.

Question 1.1. Give a definition to the following predicate stating that an array is 3-valued.

```
predicate three_valued (a:array int) = ...
```

Our sorting algorithm should have the signature "`sort (a:array int) : unit`" and its expected behavior is to permute the elements of `a` so that at the end it has the following shape, for some indices i and j:



Question 1.2. Give a definition for the following predicate stating that in an array `a`, all values stored between index `i` (included) and `j` (excluded) are equal to a given value `v`.

```
predicate one_valued (a:array int) (i j:int) (v:int) = ...
```

Recall the predicate "`permute a1 a2`", stating that the arrays `a1` and `a2` are permutations of each other, defined as the reflexive-transitive closure of the exchange of two elements.

Question 1.3. Give a postcondition for the sorting algorithm expressed in terms of the predicate `one_valued`, formalizing the expected behavior described above.

We assume given procedures to increment and decrement integer references, and a procedure for swapping two elements in an array, specified as follows:

```
val incr (x:ref int) : unit
  writes { x }
  ensures { !x = old !x + 1 }

val decr (x:ref int) : unit
  writes { x }
  ensures { !x = old !x - 1 }

val swap (a:array int) (i j:int) : unit
  requires { 0 ≤ i < a.length ∧ 0 ≤ j < a.length }
  writes { a }
  ensures { a[i] = old a[j] ∧ a[j] = old a[i] }
  ensures { forall k. 0 ≤ k < a.length ∧ k ≠ i ∧ k ≠ j → a[k] = old a[k] }
  ensures { permute (old a) a }
```

The code of our sorting algorithm is as follows.

```
let sort (a:array int) : unit =
  let k0 = ref 0 in let k1 = ref 0 in let k2 = ref a.length in
  while !k1 < !k2 do
    let v = a[!k1] in
    if v = 0 then (swap a !k0 !k1; incr k0; incr k1) else
    if v = 2 then (decr k2; swap a !k2 !k1) else
    (assert { v = 1 }; incr k1)
  done
```

Question 1.4. Assume for now the postcondition to be empty, and suppose that we are interested in proving *partial correctness* for this algorithm. What are the program points at which safety conditions need to be checked, and what are these conditions? Propose a precondition and a set of loop invariants for the code, then explain how they allow proving partial correctness (max 5 lines, focusing on the key arguments).

Question 1.5. Assume again the postcondition to be empty, and suppose that we are interested in proving the *total correctness* for this algorithm. What is the additional property to prove? Propose additional annotations to prove it, and justify (max 3 lines).

Question 1.6. We now wish to establish the postcondition stated in question 1.3. Give the additional loop invariants required, and explain how they allow proving the algorithm (max 12 lines, focusing on the key arguments).

2 Operations on binary search trees

Recall the representation and the Separation Logic definitions associated with binary search trees.

```
type node = { mutable item : int; mutable left : node; mutable right : node; } (* or null *)
```

$$p \rightsquigarrow \mathsf{Mtree}\, T \;\equiv\; \begin{array}{l}\mathsf{match}\, T\, \mathsf{with} \\ \mid \mathsf{Leaf} \Rightarrow [p = \mathsf{null}] \\ \mid \mathsf{Node}\, x\, T_1\, T_2 \Rightarrow \exists p_1 p_2.\ p \mapsto \{|\mathrm{item}{=}x;\ \mathrm{left}{=}p_1;\ \mathrm{right}{=}p_2|\} * p_1 \rightsquigarrow \mathsf{Mtree}\, T_1 * p_2 \rightsquigarrow \mathsf{Mtree}\, T_2\end{array}$$

$$p \rightsquigarrow \mathsf{Msearchtree}\, E \;\equiv\; \exists T.\ p \rightsquigarrow \mathsf{Mtree}\, T * [\mathsf{search}\, T\, E]$$

$$\frac{}{\mathsf{search}\, \mathsf{Leaf}\, \varnothing} \qquad \frac{\mathsf{search}\, T_1\, E_1 \qquad \mathsf{search}\, T_2\, E_2 \qquad \forall y \in E_1.\ y < x \qquad \forall y \in E_2.\ y > x}{\mathsf{search}\, (\mathsf{Node}\, x\, T_1\, T_2)\, (\{x\} \cup E_1 \cup E_2)}$$

Initialization The program below builds a binary search tree representing the set $[i, j)$, that is $\{i, \ldots, j-1\}$.

```
1  let rec seq i j =
2    if i >= j then null else
3    let m = (i+j)/2 in
4    let p1 = seq i m in
5    let p2 = seq (m+1) j in
6    { item = m; left = p1; right = p2 }
```

Question 2.1. Give a Separation Logic specification for the function `seq` in terms of Msearchtree.

Question 2.2. Prove the correctness of the function `seq`, following the steps below. For each step, you should write one line of explanations in English (or in French), then one line with the formulae describing the state or its evolution.
(a) Unfold the definition of Msearchtree in the specification of `seq`.
This unfolded specification is the one on which the correctness proof is carried out by induction.
(b) Explain how to establish the postcondition when `null` is returned on line 2.
(c) Explain the evolution of the state on lines 4.
(d) Explain the evolution of the state on lines 5. Explain which structural rule is used and how it is instantiated.
(e) Describe the piece of state that corresponds to the record allocated on line 6.
(f) Explain how to establish the postcondition on line 6.

Conversion to list The program below converts a binary search tree into a mutable list.

```
let rec aux r p =
  if p != null then begin
    aux r p.right;
    r := { head = p.item; tail = !r };
    aux r p.left;
  end
```

```
let tolist p =
   let r = ref null in
   aux r p;
   !r
```

Question 2.3. Give a specification for the function `tolist` in terms of Msearchtree, with a postcondition that specifies that the output list is sorted ($\mathsf{sorted}\, L$) and that the list contains the values from the set represented by the tree ($\mathsf{SetOfList}\, L = E$).

Question 2.4. Consider the logical function ToList which converts a tree into a list:

$$\begin{array}{rcl} \mathsf{ToList}\, T & \equiv & \mathsf{match}\, T\, \mathsf{with} \mid \mathsf{Leaf} \Rightarrow \mathsf{nil} \\ & & \quad \mid \mathsf{Node}\, x\, T_1\, T_2 \Rightarrow \mathsf{ToList}\, T_1 +\!\!+ x :: \mathsf{ToList}\, T_2 \end{array}$$

Give a specification for the function `aux` in terms of Mtree and ToList. This specification should be general enough to enable proving by induction the correctness of the code of `aux`, i.e. it should *not* assume r to contain the null pointer.

3 Chunked and bootstrapped chunked bags

A bag data structure is a representation of a multiset, that is, a set in which an item may occur multiple times. The size of a multiset E, written $|E|$, corresponds to the total number of items stored in E, including duplicates. The union of two multisets is written $E_1 \uplus E_2$. In this section, we study two bag data structures. The first one, called "chunked bag", is based on arrays and can be used to represent bags of bounded size. The second one, called "bootstrapped chunked bag", can be used to represent efficiently very large bags.

Chunked bags The chunked bag data structure can be used to represent bags storing up to `k` items, where `k` is a fixed parameter. A chunk is represented as a partially-filled array of size `k`, as shown below. For simplicity, we use a function `Array.make_default` which is like `Array.make` except that it does not require providing a value for initialization.

```
module Chunk = struct
  type 'a chunk = { mutable size : int; mutable data : 'a array }
  let create () = { size = 0; data = Array.make_default k }
  let size c = c.size
  let pop c = c.size <- c.cize - 1; c.data.(c.size)
  let push x c = c.data.(c.size) <- x; c.size <- c.cize + 1
  let transfer c1 c2 = while size c1 > 0 do push (pop c1) c2 done
  let iter_pop f c = while size c > 0 do f (pop c) done
end
```

Recall that $t \rightsquigarrow \mathsf{Arrayof}\, R\, M$ describes an array at address t whose contents is represented in the logic by the map M, with R being the representation predicate for the items. Let "$\mathsf{Values}\, M\, i\, j$" denote the multiset made of the values stored in the map M at the indices between i (inclusive) and j (exclusive).

Question 3.1. Define a predicate of the form $c \rightsquigarrow \mathsf{Chunkof}\, R\, E$ asserting that, at the memory location c, there exists a chunked bag data structure representing the multiset E, when R is the representation predicate for the items. Hint: use a predicate "$c \mapsto \{|\mathrm{size}{=}n;\ \mathrm{data}{=}t|\}$", use Arrayof and Values, and enforce the appropriate constraints on sizes.

Question 3.2. Give a (Separation Logic) specification to the function `Chunk.push`, expressed in terms of Chunkof. Hint: include the predicate "$x \mapsto RX$" in the precondition, and recall that a chunk cannot store more than `k` values.

Question 3.3. Give a specification to the function `Chunk.transfer`, expressed in terms of Chunkof. Hint: describe both chunks in the postcondition, and recall that a chunk cannot store more than `k` values.

Question 3.4. Give the loop invariant involved in the implementation of `transfer`, and justify termination.

Question 3.5 (Difficult)**.** Specify the function `Chunk.iter_pop`, using an invariant I that takes a multiset as argument. For this question, explicit all universal quantifiers.

Bootstrapped chunked bags The "bootstrapped chunked bag" data structure represents bags using several *layers*. The first layer contains a chunk of items. The second layer contains a chunk of chunks of items. The third layer contains a chunk of chunks of chunks of items... and so on. All chunks are represented using the data structure presented above and have the same maximal size `k`. Thereafter, we assume $\mathtt{k} = 10$ for all chunks. For example, a bag storing 162 items may be represented with a first-layer chunk storing 2 items, a second-layer chunk storing 6 chunks of 10 items each, and a third layer chunk that stores a single chunk, which itself contains 10 chunks of 10 items each. Note that it is also possible for some of the chunks involved to contain less than 10 items.

The bootstrapped chunked bag data structure admits a rather compact representation, thanks to its use of chunks. It supports push and pop operations in constant time, and supports merge and arbitrary splitting operations in logarithmic time. These operations are notably useful in the design of parallel algorithms.

The bag data structure is represented by the data type shown below. The empty bag is represented as the null pointer. A nonempty bag is represented as a record made of a chunk of elements (the current layer) and a bag of chunks of elements (the next layers). Elements stored at the first layer are base items, whereas elements stored in the deeper layers are pointers on chunks.

```
type 'a bag = { mutable head : 'a chunk; mutable next : ('a chunk) bag }
```

The polymorphic representation predicate associated with the record is shown below.

$$p \rightsquigarrow \mathsf{Cellof}\, R_1\, X_1\, R_2\, X_2 \;\equiv\; \exists x_1 x_2.\; p \mapsto \{|\text{head}{=}x_1;\, \text{next}{=}x_2|\} \,*\, x_1 \rightsquigarrow R_1\, X_1 \,*\, x_2 \rightsquigarrow R_2\, X_2$$

The layer structure is represented in the logic as shown below: a layer is either "Empty", or it is of the form "Layer $E\,T$", where E is a multiset of elements of type A, and T is a layer structure storing elements of type "multiset A".

```
Inductive layer : Type → Type :=
 | Empty : ∀ A, layer A
 | Layer : ∀ A, multiset A → layer (multiset A) → layer A.
```

Question 3.6 (Difficult)**.** Define a predicate $p \rightsquigarrow \mathsf{Layersof}\, R\, T$ to state that, at the memory location p, there exists a bootstrapped chunked bag data structure that corresponds to the logical layer structure T, when R is the representation predicate for the items. Hint: follow the standard pattern for defining representation predicate for tree structures and, to represent the elements of the next field, instantiate Layersof with "Chunkof$R$" instead of R.

Question 3.7. Define an inductive predicate $\mathsf{Layerbag}\, T\, E$ to assert that the logical tree T is a representation of the multiset E. Define it using the operation flatten, which takes as argument a multiset of multisets of items, and returns a multiset containing all the items. Present your answer in the form of derivation rules (or using Coq syntax).

Question 3.8. Define a predicate $p \rightsquigarrow \mathsf{Bag}\, E$ to assert that, at the memory location p, there exists a bag data structure that represents the multiset E of basic items, i.e. items represented using the representation predicate Id.

Question 3.9. Consider the function `push` on boostrapped chunked bags.

```
1    let push x p =
2      let c = p.head in
3      if Chunk.size c = k then begin
4        p.head <- Chunk.create();
5        if p.next == null then
6          p.next <- { head = Chunk.create(); next = null };
7        push c p.next
8      end;
9      Chunk.push x p.head
```

This function admits precondition "$p \rightsquigarrow \mathsf{Layersof}\, R\, T * x \rightsquigarrow R\, X * [\mathsf{Layerbag}\, T\, E] * [p \neq \mathsf{null}]$". State the corresponding postcondition.

Question 3.10 (Bonus). Consider an execution of the function `push` starting from the state "$p \rightsquigarrow \mathsf{Layersof}\, R\, T * x \rightsquigarrow R\, X * [\mathsf{Layerbag}\, T\, E] * [p \neq \mathsf{null}]$". Specify the state at the end of line 8 in the code of `push`. Note that your specification should allow safely executing line 9 and then establishing the postcondition that you gave in the previous question. Hint: you need to unfold the definition of "$p \rightsquigarrow \mathsf{Layersof}\, R\, T$".

Question 3.11 (Bonus). Consider the function `transfer` on bootstrapped chunked bags, shown below. The function `swap_contents`, not detailed here, swaps the contents of the fields of the two records that it takes as arguments.

```
1    let transfer p1 p2 = (* from p2 into p1; assumes p1 != null *)
2      if p2 != null then begin
3        Chunk.iter_pop (fun x -> push x p1) p2.head;
4        if p1.next == null then begin
5          p1.next <- p2.next;
6          p2.next <- null;
7        end else
8          transfer p1.next p2.next
9      end
```

Explain how the invariant I of function `Chunk.iter_pop`, given in question 3.5, needs to be instantiated in order to reason about the call on line 5 in the implementation of `transfer`. Then, prove that the hypothesis on f required by the specification of `Chunk.iter_pop`, when specialized to your invariant and to the function `(fun x → push x p1)`, is derivable from the specification of `push` given in question 3.9.