1 Circular lists

This section focuses on the formalization of circular lists and doubly linked circular lists in Separation Logic. For simplicity, we consider only lists of integers in this exercise.

Recall the type of mutable list cells.

```plaintext
type cell = { mutable hd : int; mutable tl : cell }
```

Recall also the definition of the representation predicates $MlistSeg$ and $Mlist$.

\[
p \Rightarrow MlistSeg q L \equiv \text{match } L \text{ with } \\
| \text{nil} \Rightarrow [p = q] \\
| x :: L' \Rightarrow \exists p'. p \Rightarrow \{\text{hd}=x; \text{tl}=p'\} * p' \Rightarrow MlistSeg q L'
\]

\[
p \Rightarrow Mlist L \equiv p \Rightarrow MlistSeg null L
\]

**Question 1.1.** When the last cell of the list, instead of pointing to null, points back to the first cell of the list, we have a circular list.

\[
\text{Answer.} \\
p \Rightarrow \text{CircList } L \equiv p \Rightarrow \text{MlistSeg } p L * [p = \text{null} \Leftrightarrow L = \text{nil}]
\]

**Question 1.2.** The function `read` takes as argument a pointer on a nonempty circular list, and returns the content of the item at this position.

```plaintext
let read p = p.hd
```

Give a specification for `read`, with a pre-condition of the form “$p \Rightarrow \text{CircList } (v :: L)$”.

\[
\text{Answer.} \\
\{p \Rightarrow \text{CircList } (v :: L)\} \{\text{read } p\} \{\lambda x. [x = v] * p \Rightarrow \text{CircList } (v :: L)\}
\]

**Question 1.3.** The function `forward` takes as argument a pointer on a nonempty circular list, and returns the pointer on the next item.

```plaintext
let forward p = p.tl
```

Specify `forward`, with a post-condition of the form “$\lambda q. q \Rightarrow \text{CircList } K$”, for some list $K$.

\[
\text{Answer.} \\
\{p \Rightarrow \text{CircList } (v :: L)\} \{\text{forward } p\} \{\lambda q. q \Rightarrow \text{CircList } (L & v)\}
\]
**Question 1.4.** Prove that the code of `forward` satisfies the specification that you proposed in the previous question (15 lines max). Clearly indicate where focus or defocus rules are involved in the proof (no need to prove these rules).

**Answer.** By definition of CircList, we have:
\[
p \rightsquigarrow \text{CircList } (v :: L) \quad \Rightarrow \quad p \rightsquigarrow \text{MlistSeg } p (v :: L) \circ (p = \text{null} \Leftrightarrow v :: L = \text{nil})
\]
So, \( p \neq \text{null} \). Let \( q \) be the value of \( p.tl \). By definition of MlistSeg,
\[
p \rightsquigarrow \text{MlistSeg } p (v :: L) \quad \Rightarrow \quad p \rightsquigarrow \{\text{hd}=v; \text{tl}=q\} \circ q \rightsquigarrow \text{MlistSeg } p \ L
\]
Since \( q \rightsquigarrow \text{MlistSeg } p \ L \) is part of the state, \( q \) cannot be null. (Otherwise \( p \) would have to be null as well, according to the definition of MlistSeg.) Furthermore, by definition of MlistSeg, we have:
\[
p \rightsquigarrow \{\text{hd}=v; \text{tl}=q\} \quad \Rightarrow \quad p \rightsquigarrow \text{MlistSeg } q (v :: \text{nil})
\]
By the concatenation rule for list segments, we have:
\[
q \rightsquigarrow \text{MlistSeg } p \ L \circ p \rightsquigarrow \text{MlistSeg } q (v :: \text{nil}) \quad \Rightarrow \quad q \rightsquigarrow \text{MlistSeg } q (L ++ (v :: \text{nil}))
\]
Using the fact that \( q \neq \text{null} \), we can fold the definition of CircList and deduce that the final state satisfies the heap predicate \( q \rightsquigarrow \text{CircList } (L \& v) \).

**Question 1.5.** We next consider a function to insert an item at the head of a circular list.

```plaintext
1  let insert v p =
2    if p == null then begin
3      let q = { hd = v; tl = null } in
4      q.tl <- q;
5      q
6    end else begin
7      let q = { hd = p.hd; tl = p.tl } in
8      p.hd <- v;
9      p.tl <- q;
10     p
11   end
```

Specify the above function, then prove its correctness (15 lines max).

**Answer.** Specification:
\[
\{p \rightsquigarrow \text{CircList } L\} (\text{insert } v \ p) \{\lambda q. \ q \rightsquigarrow \text{CircList } (v :: L)\}
\]

Case \( p = \text{null} \). By definition, \( L = \text{nil} \). At line 4, the state is \( q \rightsquigarrow \{\text{hd}=v; \text{tl}=q\} \), which is equivalent to \( q \rightsquigarrow \text{MlistSeg } q (v :: \text{nil}) \) and thus to \( q \rightsquigarrow \text{CircList } (v :: \text{nil}) \).

Case \( p \neq \text{null} \). By definition of CircList and MlistSeg, there exists \( p' \) and \( x \) and \( L' \) such that \( L = x :: L' \) and the state is described by:
\[
p \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \circ p' \rightsquigarrow \text{MlistSeg } p \ L'
\]
At line 10, the state is:
\[
p \rightsquigarrow \{\text{hd}=v; \text{tl}=q\} \circ q \rightsquigarrow \{\text{hd}=x; \text{tl}=p'\} \circ p' \rightsquigarrow \text{MlistSeg } p \ L'
\]
By folding twice the definition of MlistSeg, we conclude that the final state is \( p \rightsquigarrow \text{MlistSeg } p (v :: x' :: L') \), and therefore \( p \rightsquigarrow \text{CircList } (v :: L) \).

Remark: in the specification of `insert`, the post-condition may be strengthen to assert that if \( L \) is nonempty, then \( q \) is equal to \( p \).
**Question 1.6.** In order to be able to efficiently navigate the circular lists backwards, we extend it with reverse pointers. A cell is now represented as follows.

```plaintext
type cell = {
    mutable prev : cell;
    mutable item : int;
    mutable next : cell;
}
```

The goal is to define a predicate “\( p \rightsquigarrow \text{DbList} L \)” to characterize such a doubly linked list. The elements are described by the list \( L \), when starting at the cell at address \( p \) and following the next pointers until reaching \( p \) again. Note that the predicate should use the equivalence “\( p = \text{null} \Leftrightarrow L = \text{nil} \)” to handle the particular case where the list is empty.

**Hint:** Start by defining a predicate of the form “\( p \rightsquigarrow \text{DbListSeg} q r s L \)” that describes a segment of a doubly linked list storing the elements \( L \). As shown in the rightmost picture above, the first cell, at address \( p \), stores the address \( s \) in its prev field, and the last cell, at address \( r \), stores the address \( q \) in its next field. When \( L \) is empty, enforce \( p = q \) and \( r = s \).

**Answer.**

\[
p \rightsquigarrow \text{DbListSeg} q r s L \equiv \text{match} \, L \text{ with } \\
| \text{nil} \Rightarrow [p = q \land r = s] \\
| x :: L' \Rightarrow \exists p'. \{ \text{item} = x; \text{prev} = s; \text{next} = p' \}
\]

Then:

\[
p \rightsquigarrow \text{DbList} L \equiv \exists q. \ p \rightsquigarrow \text{DbListSeg} pqq L \ast [p = \text{null} \Leftrightarrow L = \text{nil}]
\]

**Remark:** if \( L = \text{nil} \), we have \( p = q = \text{null} \); and if \( L = v :: \text{nil} \), we have \( p = q \neq \text{null} \).

## 2 Filtering

In this exercise, we are interested in specifying and verifying, in Separation Logic, functions that filters particular elements from a list. We assume the existence of a predicate called Even, which characterizes even integers. We also assume the existence of a logical function Filter, which applies to a predicate of type “\( A \rightarrow \text{Prop} \)” and to a list of type “\( \text{list} \, A \)”.

For example, “\( \text{Filter Even} \, L \)” filters the even elements from a list \( L \), in the logic.

**Question 2.1.** We first consider a purely-functional implementation of filter on a pure list of integers, specialized to filtering even numbers.

```plaintext
let rec list_filter_even l = 
    match l with 
    | [] -> [] 
    | x::t -> let q = list_filter_even t in 
              if x mod 2 = 0 then x::q else q
```

Give a specification for this function, using a Separation Logic triple.
Answer.

\[
\begin{align*}
&\{[]\} (\text{list\_filter\_even } l) \ \{\lambda l'. \ [l' = \text{Filter Even } l]\}
\end{align*}
\]

Question 2.2. We now adapt the above function to mutable lists as follows:

\[
\begin{align*}
\text{let rec mlist\_filter\_even } p = \\
&\text{if } p == \text{null} \text{ then null else begin} \\
&\text{let } q = \text{mlist\_filter\_even } p.tl \text{ in} \\
&\text{if } p.hd \text{ mod } 2 = 0 \text{ then } (p.tl <- q; p) \\
&\text{else } q
\end{align*}
\]

Give a Separation Logic specification for this function, making use of the predicate Mlist.

Answer.

\[
\{p \rightsquigarrow \text{Mlist } L\} (\text{mlist\_filter\_even } p) \ \{\lambda q. \ q \rightsquigarrow \text{Mlist} (\text{Filter Even } L)\}
\]

Question 2.3. Prove that mlist\_filter\_even satisfies the specification that you claimed in the previous question (in 20 lines max).

Answer. Proof by induction on the length of \(L\).

- Case \(p = \text{null}\). From the pre-condition \(p \rightsquigarrow \text{Mlist } L\), we get \([L = \text{nil}]\). So, “Filter Even \(L = \text{nil}\)”.
  This validates the post-condition for \(q = \text{null}\), since we have \([\] \Rightarrow (\text{null} \rightsquigarrow \text{Mlist} \text{nil})\).

- Case \(p \neq \text{null}\). From the pre-condition, by the focus rule, there exists \(x\) and \(L'\) and \(t\) such that: \(L = x :: L'\) and the state is \(p \rightsquigarrow \{\text{hd}=x;\ \text{tl}=t\} \Rightarrow \text{Mlist } L'\). To reason about the recursive call, we apply the frame rule to keep only \(t \rightsquigarrow \text{Mlist } L'\), and invoke the induction hypothesis. We learn that \(q\) is such that “\(q \rightsquigarrow \text{Mlist} (\text{Filter Even } L')\)”.

  - Case \(\text{Even } x\). We have Filter Even \(L = x :: \text{Filter Even } L'\). The state is modified to: \(p \rightsquigarrow \{\text{hd}=x;\ \text{tl}=q\} \Rightarrow q \rightsquigarrow \text{Mlist} (\text{Filter Even } L')\). Applying the defocus rule gives \(p \rightsquigarrow \text{Mlist} (\text{Filter Even } L)\), which validates the post-condition since \(p\) is returned.

  - Case \(\neg\text{Even } x\). We have Filter Even \(L = \text{Filter Even } L'\). Applying the garbage collection rule to \(p \rightsquigarrow \{\text{hd}=x;\ \text{tl}=t\}\), we are left with “\(q \rightsquigarrow \text{Mlist} (\text{Filter Even } L')\)”, which validates the post-condition since \(q\) is returned.

Question 2.4. We generalize the function so that it may take as argument a filter function \(f\), of type \(\text{int} \rightarrow \text{bool}\).

\[
\begin{align*}
\text{let rec mlist\_filter } f \ p = \\
&\text{if } p == \text{null} \text{ then null else begin} \\
&\text{let } q = \text{mlist\_filter } p.tl \text{ in} \\
&\text{if } f \ p.hd \text{ then } (p.tl <- q; p) \\
&\text{else } q
\end{align*}
\]

Give the specification of this function, assuming that \(f\) does not perform any visible side effect. For this question, quantify explicitly all the variables involved.

Answer.

\[
\begin{align*}
\forall fP. \ \forall x. \ \{ [] \} (f \ x) \ \{ \lambda b. \ [b = \text{isTrue} \ (P x)]\} \Rightarrow \\
\{p \rightsquigarrow \text{Mlist } L\} (\text{mlist\_filter } f \ p) \ \{\lambda q. \ q \rightsquigarrow \text{Mlist} (\text{Filter } P \ L)\}
\end{align*}
\]

4
**Question 2.5.** We are now interested in the generalization to mutable lists containing items of arbitrary types. For this purpose, we need in particular to quantify over the representation predicate \( R \) associated with the items stored in the list, and use \( \text{Mlistof} \) instead of \( \text{Mlist} \).

Moreover, we assume that the function \( f \) may read (but not write) values in the heap; in particular, it may not alter any of the heap invariants. So, the function \( f \), in addition to being able to access the representation of the items that it receives as argument, should also be able to read in other parts of the heap. These pieces of heap should be described by an invariant, call it “\( I \)”, that should appear in both the pre- and the post-condition of \( \text{mlist_filter} \).

Give the corresponding specification of \( \text{mlist_filter} \).

**Answer.**

\[
(\forall x. \{x \rightsquigarrow RX \* I\} \ (f x) \{\lambda b. \ [b = \text{isTrue} (PX)] \ * \ x \rightsquigarrow RX \* I\}) \Rightarrow
\{p \rightsquigarrow \text{Mlistof} \ R \* I\} \ (\text{mlist_filter} \ f \ p) \{\lambda q. q \rightsquigarrow \text{Mlistof} \ R \ (\text{Filter} \ P \ L) \* I\}
\]

**Question 2.6.** Consider the application of \( \text{mlist_filter} \) to a mutable list “\( p \)” made of possibly-aliased reference cells, each of these reference storing an integer. The goal is to filter the reference cells that store an even number. Using the “\( \text{Group} \)” representation predicate to describe the possibly-aliased reference cells, state a triple that specifies the behavior of the following expression, which you may refer to as “\( \text{expr} \)”.

\( \text{mlist_filter} \ (\text{fun} \ r \rightarrow !r \ mod \ 2 = 0) \ p \)

**Answer.**

\[
\{p \rightsquigarrow \text{Mlist} \* \text{GroupOf Ref} \* \{\forall x \in L. x \in \text{dom}(G)\}\}
\]

\( \text{expr} \)

\[
\{\lambda q. q \rightsquigarrow \text{Mlist} \ (\text{Filter} \ (\lambda x. \text{Even}(G(x))) \ L) \* \text{GroupOf Ref} \ G\}
\]

where \( G \) has type “\( \text{fmap loc int} \)”.

**Question 2.7.** Prove that the function “\( \text{fun} \ r \rightarrow !r \ mod \ 2 = 0 \)” satisfies the hypothesis made on \( f \) in the specification of \( \text{mlist_filter} \) in question 2.5. Clearly state the focus rule that you exploit (max 12 lines for the proof).

**Answer.** Instantiate the specification of \( \text{mlist_filter} \) with \( R = \text{id} \) and \( I = \text{GroupOf Ref} \ G \) and \( P = \lambda x. \text{Even}(G(x)) \). Since \( x \rightsquigarrow RX \) is equivalent to \( [x = X] \), the hypothesis on \( f \) simplifies to:

\[
\forall r. \ \{\text{GroupOf Ref} \ G\} \ (f \ r) \ {[\lambda b. \ [b = \text{isTrue} (P \ r)] \ * \ \text{GroupOf Ref} \ G} \}
\]

We wish to relate the above specification to:

\[
\forall rn. \ \{r \rightsquigarrow \text{Ref} \ n\} \ (\lambda r. \ [r \ mod \ 2 = 0] \ {[\lambda b. \ [b = \text{isTrue} (\text{Even} \ n)] \ * \ r \rightsquigarrow \text{Ref} \ n}\}
\]

To that end, it suffices to exploit the following focus rule in both directions:

\[
\text{GroupOf Ref} \ (G' \ \text{ref } \ x) = (x \rightsquigarrow \text{Ref } X) \ * \ (\text{GroupOf Ref} \ G')
\]

and frame out \( (\text{GroupOf Ref} \ G') \) during the evaluation of “\( !r \ mod \ 2 = 0 \)”.

5