Exercises on Lists and Arrays in Separation Logic

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

2 May, 2023

Separation logic is designed to reason about data structures and algorithms that manipulate pointers (i.e., dynamically allocated memory).

1 Linked lists in Separation Logic

We can define a predicate in separation logic that allows us to characterize the *shape* of a (singly) linked list in memory. Since we want to reason about the list contents as well, we will index the predicate with the underlying *sequence* of elements that the list encodes. In the following, we use ϵ to denote the empty sequence, $\alpha, \beta, \tau, \sigma$ to range over sequences, $\alpha^\dagger$ to denote the sequence obtained by reversing the order of the elements of sequence α, and, $a \cdot \alpha$ (respectively, $\alpha \cdot a$) to denote the sequence obtained by adding the element a to the start (respectively, the end) of sequence α. As a slight abuse of notation we will use $\alpha \cdot \beta$ to denote the sequence made up of the elements of α followed by those in β (i.e., the concatenation of α and β). Our list predicate is defined as follows, stating that variable i refers to a (singly) linked list encoding sequence α:

$$\begin{array}{lcl} \mathsf{list}\,\epsilon\,i & \triangleq & \mathbf{emp} \wedge i = \mathbf{null} \\ \mathsf{list}\,(a \cdot \alpha)\,i & \triangleq & \exists j. i \mapsto a, j * \mathsf{list}\,\alpha\,j \end{array}$$

The predicate is defined by structural induction on the given sequence, using the small footprint principle of separation logic. For the empty sequence ϵ, we state that the memory footprint of the list is characterized by the empty head and that i is the **null** reference. For a non-empty sequence of the form $a \cdot \alpha$, we state that i *points to* an object in memory made up of two fields, one containing the list element a and the other j (encoded using the points-to assertion $i \mapsto a, j$), which is itself (separately) a list encoding sequence α.

We can now use this definition of a list to prove the functional correctness of an in-place list reversal procedure that reverses the pointer structure of the list. The Java code for the procedure is as follows:

```
public class List {

    Node head;

    void reverse()
    {
        Node n = null;

        while (head != null)
        {
            Node k = head.next;
            head.next = n;
            n = head;
            head = k;
        }
        head = n;
    }
}
```

and we will show that $\{\mathsf{List}\,\alpha\,\mathtt{head}\}\,\mathtt{reverse}()\,\{\mathsf{List}\,\alpha^{\dagger}\,\mathtt{head}\}$.

The list implementation is made up of a chain of `Node` objects, as seen in the last lecture, each with a `next` field. The `reverse` procedure is deceptively succint: variable `n` maintains a pointer to the previously edited node in the list and the loop variable `k` maintains the next element in the list. The loop then proceeds by storing the pointer to the next element in the list in `k`,"reversing" the `next` pointer of the head of the list, storing the current `head` in `n` so that `n` stores the new "previous" element and then advancing the `head` pointer to `k`. After the loop terminates, the head of the list is updated to `n`, the last element in the initial list.

It is not too challenging to observe that at any given point during the execution of the loop, the chain of `Node`s starting at `n` is precisely the reversed subsequence of the original list, whereas the chain of `Node`s starting from `head` denote the elements that have yet to be reversed. If α is the sequence originally encoded by the list, σ the sequence starting at `head` and τ the sequence starting at `n`, we can observe that $\alpha = \tau^{\dagger} \cdot \sigma$:

$$\{\mathsf{List}\,\alpha\,\mathtt{head}\}$$
$$\{\mathsf{List}\,\alpha\,\mathtt{head} * (\mathbf{emp} \wedge \mathtt{null} = \mathtt{null})\}$$
`Node n = null`
$$\{\mathsf{List}\,\alpha\,\mathtt{head} * (\mathbf{emp} \wedge \mathtt{n} = \mathtt{null})\}$$
$$\{\mathsf{List}\,\alpha\,\mathtt{head} * \mathsf{List}\,\epsilon\,\mathtt{n}\}$$
$$\{\exists \sigma, \tau. \mathsf{List}\,\sigma\,\mathtt{head} * \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger} \cdot \sigma\}$$

On method entry we have that $\mathsf{List}\,\alpha\,\mathtt{head}$. Before execution reaches the loop we can establish straightforwardly that, *separately*, $\mathsf{List}\,\epsilon\,\mathtt{n}$. At this point, simple logical manipulation allows us to generalize this assertion by quantifying existentially over the two sequences, and singling out that the original sequence α can be obtained by concatenating the reversal of τ (for now, the empty sequence) with σ (which consists of α itself, for now).

We can now check that the last assertion above can indeed be used as a loop invariant:

$$\{\exists \sigma, \tau. \mathsf{List}\,\sigma\,\mathtt{head} * \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger} \cdot \sigma\}$$
`while (head != null) {`
$$\{\exists \sigma, \tau. \mathsf{List}\,\sigma\,\mathtt{head} * \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger} \cdot \sigma\}$$
$$\{\exists \sigma, \tau. \mathsf{List}\,(a \cdot \sigma)\,\mathtt{head} * \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger} \cdot (a \cdot \sigma)\}$$
$$\{\exists \sigma, \tau, k. \mathtt{head} \mapsto a, k * \mathsf{List}\,\sigma\,\mathtt{k} * \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger} \cdot (a \cdot \sigma)\}$$
`Node k = head.next;`
$$\{\exists \sigma, \tau. \mathtt{head} \mapsto a, k * \mathsf{List}\,\sigma\,\mathtt{k} * \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger} \cdot (a \cdot \sigma)\}$$
`...`
`} //end of loop body`

Up to the first assignment of the loop body, all we do is leverage the loop condition to determine that the list denoted by `head` must be non-empty, thus warranting us to obtain its `next` pointer and storing it in local variable `k`.

```
Node k = head.next;
```
$\{\exists \sigma, \tau . \mathtt{head} \mapsto a, k \, * \, \mathsf{List}\,\sigma\,\mathtt{k} \, * \, \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger} \cdot (a \cdot \sigma)\}$
```
head.next = n;
```
$\{\exists \sigma, \tau . \mathtt{head} \mapsto a, \mathtt{n} \, * \, \mathsf{List}\,\sigma\,\mathtt{k} \, * \, \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger} \cdot (a \cdot \sigma)\}$
```
...
```

After the first mutation performed by the loop, the next pointer of `head` is now `n`.

```
head.next = n;
```
$\{\exists \sigma, \tau . \mathtt{head} \mapsto a, \mathtt{n} \, * \, \mathsf{List}\,\sigma\,\mathtt{k} \, * \, \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger} \cdot (a \cdot \sigma)\}$
$\{\exists \sigma, \tau . \mathsf{List}\,(a \cdot \tau)\,\mathtt{head} \, * \, \mathsf{List}\,\sigma\,\mathtt{k} \wedge \alpha = \tau^{\dagger} \cdot (a \cdot \sigma)\}$
$\{\exists \sigma, \tau . \mathsf{List}\,(a \cdot \tau)\,\mathtt{head} \, * \, \mathsf{List}\,\sigma\,\mathtt{k} \wedge \alpha = (a \cdot \tau)^{\dagger} \cdot \sigma\}$
$\{\exists \sigma, \tau . \mathsf{List}\,\tau\,\mathtt{head} \, * \, \mathsf{List}\,\sigma\,\mathtt{k} \wedge \alpha = \tau^{\dagger} \cdot \sigma\}$
```
...
```

We now observe that after the assignment, we have that the predicate $\mathsf{List}\,(a \cdot \tau)\,\mathtt{head}$ holds since we had that $\mathsf{List}\,\tau\,\mathtt{n}$. We can then apply some basic properties of sequences to see that $\tau^{\dagger} \cdot (a \cdot \sigma)$ is equivalent to $(a \cdot \tau)^{\dagger} \cdot \sigma$. In the last step, we simply existentially quantify over the sequences again. We then simply replace `head` with `n` and `k` with `head`:

$\{\exists \sigma, \tau . \mathsf{List}\,\tau\,\mathtt{head} \, * \, \mathsf{List}\,\sigma\,\mathtt{k} \wedge \alpha = \tau^{\dagger} \cdot \sigma\}$
```
n = head;
head = k;
```
$\{\exists \sigma, \tau . \mathsf{List}\,\tau\,\mathtt{n} \, * \, \mathsf{List}\,\sigma\,\mathtt{head} \wedge \alpha = \tau^{\dagger} \cdot \sigma\}$
```
} //end of loop body
```

We have thus re-established the loop invariant! After the loop, since the invariant holds *and* the loop condition must be false we have that the sequence that starts at node `head` must be empty and so:

$\{\exists \tau . \mathsf{List}\,\tau\,\mathtt{n} \, * \, \mathsf{List}\,\epsilon\,\mathtt{head} \wedge \alpha = \tau^{\dagger} \cdot \epsilon\}$
$\{\exists \tau . \mathsf{List}\,\tau\,\mathtt{n} \, * \, \mathsf{List}\,\epsilon\,\mathtt{head} \wedge \alpha = \tau^{\dagger}\}$
$\{\exists \tau . \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha = \tau^{\dagger}\}$
$\{\exists \tau . \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha^{\dagger} = (\tau^{\dagger})^{\dagger}\}$
$\{\exists \tau . \mathsf{List}\,\tau\,\mathtt{n} \wedge \alpha^{\dagger} = \tau\}$
$\{\mathsf{List}\,\alpha^{\dagger}\,\mathtt{n}\}$
```
head = n;
```
$\{\mathsf{List}\,\alpha^{\dagger}\,\mathtt{head}\}$

Since the node chain starting at `head` is `null`, we can drop it. We then conclude that $\alpha^{\dagger} = \tau$ since the reversing a sequence twice produces the original sequence. The final assignment produces the desired post-condition, establishing that `head` is now a chain of nodes encoding the reversal of the original sequence α.

1.1 List Reversal in Verifast

Even though our reasoning above is rigorous, the procedure and the reasoning itself is intricate enough that the reader may wonder if all the steps are indeed logically justified. A way to potentially improve our trust in our argument is to attempt to check our reasoning in a tool such as Verifast.

We begin by encoding the list predicate(s):

```
/*@
  predicate Node(Node n; Node nn, int v) = n.next |-> nn &*&
    n.val |-> v;
  predicate List(Node n; list<int> elems) = n == null ? emp &*&
   elems == nil :
    Node(n,?nn,?v) &*& List(nn,?tail) &*& elems == cons(v,tail);
  predicate ListInv(List l; list<int> elems) = l.head |-> ?h &*&
    List(h,elems);
@*/
```

The `Node` predicate characterizes a list node as an object with a next and value fields. The `List` predicate coincides with the previous definition of list, using the logical type `list<int>` to encode sequences. Finally, the `ListInv` predicate characterizes a list object as having a `head` field that satisfies the `List` predicate.

The code for list nodes is:

```
class Node {
Node next;
int val;


Node()
//@requires true;
//@ensures Node(this,null,0);
{
next = null;
val = 0;
}


Node(int v, Node next)
//@requires true;
//@ensures Node(this,next,v);
 {
this.next = next;
val = v;
}

void setNext(Node next)
//@ requires Node(this,_,?v) &*& Node(next,_,_);
//@ ensures Node(this,next,v);
{
this.next = next;
}



void setVal(int v)
//@ requires Node(this,?nn,_);
//@ ensures Node(this,nn,v);
{
val = v;
}

}
```

The code for the linked list implementation is:

```
class List {


Node head;

public List()
//@ requires true;
//@ ensures ListInv(this,nil);
{
head = null;
}

void reverseList()
//@ requires ListInv(this,?l);
//@ ensures ListInv(this,reverse(l));
{
  Node n = null;
  //@open ListInv(this,l);

  while (head != null)
  /*@ invariant head |-> ?h &*& List(h,?l1) &*& List(n,?l2)
         &*&  l == append(reverse(l2),l1); *@/
  {
   Node k = head.next;
   head.next = n;
   n = head;
   head = k;
   /*@assert l1 == cons(?v,?tail0) &*&
           l == append(reverse(l2),cons(v,tail0));@*/
   //@reverse_reverse(cons(v,tail0));
   //@reverse_append( reverse(cons(v,tail0))  , l2 );
   //@append_assoc(reverse(tail0),cons(v,nil),l2);
   //@reverse_append(reverse(tail0),cons(v,l2));
   //@ reverse_reverse(tail0);
   }
   //@open List(h,l1);
   head = n;
   //@append_nil(reverse(l2));

  }
 }
```

The code for the reverse method requires the use of several lemmas that essentially correspond to the reasoning about sequence done in our "pen and paper" proof.

2 Arrays in Verifast

Separation logic has no primitive notion of array. In pen and paper proofs, arrays are easily modelled as contiguous chunks of memory via the *points-to* assertion. However, in a system like Verifast, arrays are encoded via a series of built-in predicates that aim to characterize an array and its contents:

```
predicate array_element<T>(T[] array, int index; T value);
predicate array_slice<T>(T[] array, int start, int end; list<T> elements);
predicate array_slice_deep<T, A, V>(T[] array, int start, int end,
          predicate(A, T; V) p, A info; list<T> elements, list<V> values);
```

The `array_element` characterizes a single element at position `index` of the given **`array`**, which must be equal to `value`. The `array_slice` predicate indicates that the given **`array`** has valid indices from `start` to `end`, which consist exactly of the elements in the (logical) list `elements`. The `array_slice_deep` predicate generalizes `array_slice` by further allowing the array elements to satisfy a supplied predicate `P`. For instance, the following snippet:

```
predicate Positive(unit a, int v; unit n) = v >= 0 &*& n == unit;

array_slice_deep(s,0,n,Positive,unit,elems,_)
```

Indicates that array `s` has valid indices from positions `0` to `n`, all satisfying the `Positive` predicate (i.e., they are non-negative integers).

2.1 Iterating over arrays

The following piece of code specifies a method that sums the elements of a given array of integers:

```
fixpoint int sum(list<int> vs) {
  switch(vs) {
    case nil: return 0;
    case cons(h, t): return h + sum(t);
  }
}

lemma_auto(sum(append(xs, ys))) void sum_append(list<int> xs, list<int> ys)
  requires true;
  ensures sum(append(xs, ys)) == sum(xs) + sum(ys);
{
  switch(xs) {
    case nil:
    case cons(h, t): sum_append(t, ys);
  }
}
```

```
  public static int sum(int[] a)
  //@ requires array_slice(a, 0, a.length, ?vs);
  //@ ensures array_slice(a, 0, a.length, vs) &*& result == sum(vs);
  {
    int total = 0; int i = 0;
    while(i < a.length)
      //@ invariant 0 <= i &*& i <= a.length &*& array_slice(a, 0, a.length, vs)
          &*& total == sum(take(i, vs));
    {
      int tmp = a[i]; total = total + tmp;
      //@ length_drop(i, vs);
      //@ take_one_more(i, vs);
      i++;
   }
    return total;
  }
```

Note how the method's pre-condition specifies the given array `a` via the `array_slice` predicate. Moreover, note how the use of `array_slice` in the loop invariant does not rely on `i` but rather on `0` and `a.length` (i.e., we do not use `array_slice(a,0,i,vs)` as an invariant), since otherwise we would not be able to "grow" the valid range of the array.

To prove the functional correctness of the loop, we observe that the value of variable `total` tracks the sum of the first `i` elements of the array, which is noted in the loop invariant via the assertion `total =sum(take(i, vs))`, where `take(i,vs)` is the list composed of the first `i` elements of `vs`. The reasoning then relies on the following lemmas (part of Verifast's standard library of lemmas):

```
lemma void length_drop<t>(int n, list<t> xs);
    requires 0 <= n && n <= length(xs);
    ensures length(drop(n, xs)) == length(xs) - n;

lemma void take_one_more<t>(int i, list<t> xs);
  requires 0 <= i &*& i < length(xs);
  ensures take(i + 1, xs) == append(take(i, xs),
         cons(nth(i, xs), nil));
```

The first characterizes the length of a list obtained via `drop` and the second characterizes the relationship between `take`-ing `i+1` and `i` elements.

Exercises

1. Write and verify a method that, given an array and an index into the array, returns the element stored in that position.

2. Write and verify a method that, given an array, an index into the array and a value to place in the array, updates the index of the array to contain the given value. **Hint:** you will need to use the lemma given below, and the list specification function `store`:

```
lemma void store_take_drop<t>(list<t> xs, int index, t v)
  requires 0 <= index && index < length(xs);
  ensures store(xs, index, v) == append(take(index, xs), cons(v, drop(index + 1, xs)));
```

```
{
  switch(xs) {
    case nil:
    case cons(h, t):
      if(index == 0) {
      } else {
        store_take_drop(t, index - 1, v);
      }
  }
}
```

3. [⋆⋆] Write and verify a method that sums the elements of a given array, iterating over the array from the higher to the lower indices of the array.

4. [⋆⋆⋆] Write and verify a method that returns the minimum element of a (non-empty) array. It will be helpful to factor your implementation into two methods, one that computes the index of the minimum element of the array, starting at a given index, and another that calls this method.

```
  public static int min(int[] a)
    //@ requires a != null &*& array_slice(a, 0, a.length, ?vs) &*& vs != nil;
    //@ ensures ...
  {

    int tmp = indexOfMin(a, 0);
    //@ length_drop(tmp, vs);
    //@ nth_drop(vs, tmp);
    //@ mem_nth(tmp, vs);
    return a[tmp];
  }

  public static int indexOfMin(int[] a, int start)
    //@ requires a != null &*& array_slice(a, start, a.length, ?vs) &*& vs != nil &*& length(
         vs) != 0;
    //@ ensures ...
  {
    ...
  }
```

The following auxiliary definition will likely be helpful:

```
fixpoint boolean forall_le(list<int> vs, int v) {
  switch(vs) {
    case nil: return true;
    case cons(h, t): return v <= h && forall_le(t, v);
  }
}
```