15-122 : Principles of Imperative Computation

Fall 2012

Assignment 4 – Selected Solutions

(Theory Part)

The following solutions are provided to you to help you study this semester. They are not to be distributed to others outside of the class nor are they intended to be used by students in future semesters as a substitute for completing one's own homework assignments.

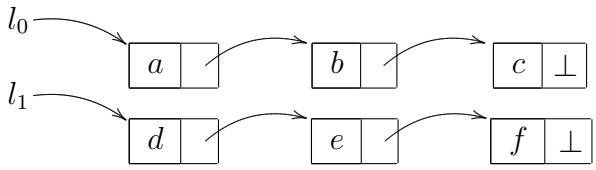
1. **Memory Contracts.** As lists are defined as `structs`, in C_0 we can only interact with them by reference—that is, with pointers. As a result, we must be careful that two lists which we would like to think of as separate are actually stored in distinct memory. For value types, like `int`s, this is automatically handled by the type system. For reference types (pointers), we can reason with contracts.

   For the purposes of this question, we will consider operations on non-circular, `NULL`-terminated lists. You may assume that you have a specification function:

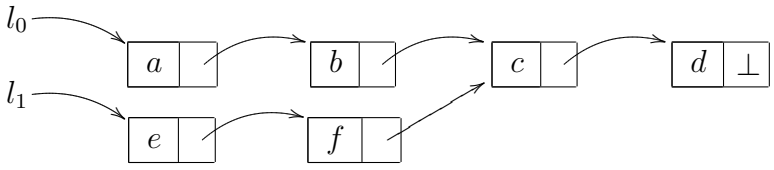
   ```
   bool is_nt_list(list *L)
   ```

   which correctly checks a list for these properties.

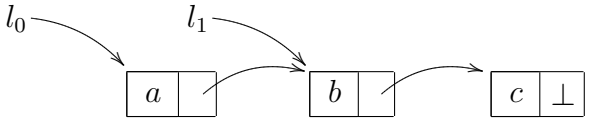
   Two lists can either be distinct,

   Or they can share some tail.

   Note that once the lists join, they cannot diverge. Also, note that one list may be contained within the other.

We can think of this as a special case of the lists sharing a tail—that tail just happens to be one of the lists in its entirety.

By convention, we treat `NULL` as the (valid) empty list. This means that `is_nt_list(NULL)` will return true, and a list that is `NULL` is distinct from everything, as it does not occupy memory.

(3) (a) Write the following function to check if two lists occupy different memory. Your function should return true if there is no list node that is common to both lists, false if the lists share some tail, or raise an annotation failure if either list is not a valid `NULL`-terminated list. There are several ways to do this; the fastest is $O(n)$, but an $O(n^2)$ solution is acceptable. Use the given contracts, and don't modify them or add other preconditions or postconditions.

Solution:

```
bool lists_pwise_distinct(list *L0, list *L1)
//@requires is_nt_list(L0) && is_nt_list(L1);
{
  if(L0 == NULL || L1 == NULL)
    return true;

  //@assert(L0 != NULL && L1 != NULL);

  list *L0_tail = L0;
  while(L0_tail->next != NULL)
    //@loop_invariant L0_tail != NULL;
    L0_tail = L0_tail->next;

  list *L1_tail = L1;
  while(L1_tail->next != NULL)
    //@loop_invariant L1_tail != NULL;
    L1_tail = L1_tail->next;

  return L0_tail != L1_tail;
}
```

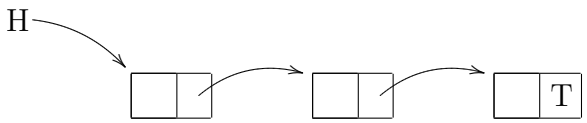
Notice that any pointer that is not explicitly required to be non-null must be checked for being NULL and handled appropriately. This solution works because if two *singly linked* lists begin overlapping at any point, they must also overlap from that point until the end of the list. Therefore, it is sufficient to check that the last elements occupying the same memory.

A list segment can be described by two list pointers:

```
bool is_segment(list *H, list *T)
{
  list *p;
  for (p = H; p != NULL && p != T; p = p->next) {}
```

```
  return p == T;
}
```

`is_segment(H,T)` is true if and only if `H` is connected to `T` via list nodes.



Notice that `is_segment(H,NULL)` is true if and only if `H` is a `NULL`-terminated list, assuming the list is not circular.

The functions above are useful in a function that copies a list:

```
list *nt_list_clone(list *L)
//@requires is_nt_list(L);
//@ensures is_nt_list(\result);
//@ensures lists_pwise_distinct(\result, L);
{
    list *new_head = NULL;
    list *new_tail = NULL;
    list *p = L;

    while(p != NULL)
    //@loop_invariant is_nt_list(new_head);
    //@loop_invariant lists_pwise_distinct(new_head, L);
    //@loop_invariant is_segment(new_head, new_tail);
    {
        // copy the node
        list *new_node = alloc(list);
        new_node->data = p->data;
        new_node->next = NULL;
        //@assert lists_pwise_distinct(new_node, L);

        // if this is the first node, set up the clone
        if(new_tail == NULL)
        {
          new_tail = new_node;
          new_head = new_node;
        }
        // otherwise add to the clone
        else
        {
          new_tail->next = new_node;
          new_tail = new_node;
        }

        // follow the original list
```

```
        p = p->next;
    }

    return new_head;
}
```

You will now prove the contracts for `nt_list_clone`. Don't worry about equality of the elements—for this problem, we're only concerned with the structure of the lists in memory. You may assume that memory returned by `alloc` is distinct from any memory already in use, per the `@assert` statement in the loop body.

(1) (b) Show that the loop invariants hold upon entering that the loop.

Solution: NULL is trivially a null-terminated list, and it is distinct from all lists. NULL to NULL is also trivially a segment.

(3) (c) Show that the loop invariants hold after an iteration of the loop, given that they held at the beginning of that iteration.

Solution: We handle this in two cases:
In the first case, if `new_tail` was NULL, then `new_tail` and `new_head` are both `new_node`.
The `next` pointer is NULL, so `new_head` is a NULL-terminated list.
`new_node` is fresh memory, so it must be distinct from `L`.
The span from `new_node` to itself is trivially a segment.

In the second case, we invoke the third loop invariant to claim that `new_head` to `new_tail` was a segment in the previous iteration of this loop (or upon loop initialization).
We append `new_node` onto the end of this segment. Since we then set `new_tail` to `new_node`, we can say that `new_head` to `new_tail` is still a segment.
Since `new_node`'s `next` pointer is NULL and `new_head` to `new_tail` is a segment, the list from `new_head` is NULL terminated.
Since `new_node` is fresh memory and the rest of the segment was distinct memory by L.I. 2, the list from `new_head` is still pwise distinct from L.

(1) (d) Show that the postconditions are satisfied. *HINT: look at the loop invariants!*

Solution: We simply observe that the are implied by L.I.s 1 and 2, should we ever exit the loop. *NOTE: Contrary to the usual pattern, we don't need the negated loop guard to show that the post-conditions hold for this function.*