IMPLEMENTATION AND PROOF OF A LONGEST COMMON SUBSTRING ALGORITHM

DOMINIK GABI

Contents

1. Introduction

The longest common substring problem [4] is a special case of the longest common subsequence problem [3]. The latter is defined as follows: given n sequences $s_1, ..., s_n$ find the longest sequence common to all sequences. The substring problem restricts this definition to consecutive parts of sequences (i.e. substrings). Both problems are textbook examples for dynamic programming.
The initial intention was therefore to implement and prove the dynamic programming solution. I started out with a brute-force prototype and soon came to realize that proving takes a lot more time than expected. While the prototype code was written within an afternoon, I spent a full week proving its correctness. Time restrictions finally lead me to abandon the initial plan and settle for the brute-force algorithm that will be described in this report.
I chose to write the proof using Verifast [2] for the simple reason that it supports C. I currently write most of my code in C++ and considering the complexity of the language, I have always longed for some support for formal methods[1].

2. Implementation and Proof

Before I go into details about the longest common substring implementation I will discuss the linked list implementation and the basic list operations. The code listings are taken from the *sublist.c* file accompanying this document. The line numbers mentioned also refer to this file. Please note that this document is not meant as a comprehensive documentation, nonetheless I will try to provide a good overview of the code and the proofs. For more details, I have found that the pre- and postconditions of the functions and lemmas provide excellent documentation. I

[1]Admittedly, Verifast is not what I had in mind.

have thought about including these signatures in this document but decided against it due to the sheer amount of space they would require.

2.1. **Linked List.** The linked list implementation is based on the *iter.c* example which can be found at [1]. Note that since I started from a list example, I will use the terms string and list (substring, sublist respectively) synonymously. The list can be described as a sequence of node `struct`s. Each node has two fields: one is a pointer to the next node, the other is the value of the element. Since the list is supposed to represent a string, this value is of type `char`. A wrapper `struct` for the list contains the first node of the list as well as a dummy node which terminates the list. Listing 1 shows the corresponding code.

Listing 1. Data Structures

```
struct node {
        struct node *next;
        char value;
};

struct list {
        struct node *first;
        struct node *after;
};
```

The four important predicates defined on these two structures follow in Listing 2. The `node` predicate describes a single node `struct`. It states that the node is allocated and that its fields point to the respective arguments of the predicate.
The segment describes a sequence of nodes and its values using the inductive list data type (defined in *list.h*). Again, the `last` parameter does not indicate an actual element of the list but a dummy node that terminates the list.
The next is the `list` predicate which describes a list that is allocated, has at least one node (the terminating node) and describes a `segment` of values between `first` and `after`.
The last of the four important predicates is the `segment2` predicate which also takes into account the `after` node. With these predicates it is possible to iterate in various ways over lists.

Listing 2. Predicates

```
predicate node(struct node *node; struct node *next,
               char value)
        requires malloc_block_node(node)
        &*& node->next |-> next &*& node->value |-> value;

predicate segment(struct node *first, struct node *last;
                list<char> values)
        requires first == last ?
                        values == nil
                :
                        node(first, ?next, ?value)
                        &*& segment(next, last, ?tail)
                        &*& values == cons(value, tail);

predicate list(struct list *list; list<char> values)
        requires malloc_block_list(list)
        &*& list->first |-> ?first
        &*& list->after |-> ?after
        &*& segment(first, after, values)
        &*& node(after, _, _);

predicate segment2(struct node *first,
                struct node *last,
        struct node *after, list<char> values)
        requires
                switch (values) {
                        case nil: return first == last;
                        case cons(head, tail):
                                return first != after
                                &*& node(first, ?next, head)
                                &*& segment2(next, last, after,
                                                tail);
                };
```

2.2. **Basic Operations.** The example I started with already contained a few functions to manipulate lists, such as the `list_add`, `list_append` as well as the `list_length` function. I have extended this selection with functions to copy and reverse lists (`list_copy`, `list_reverse`). Furthermore, I have added the functions `list_drop_n` and `list_take_n` which implement the corresponding fixpoint functions defined in *list.h.* Note that my focus was on verifiability and not performance. A prime example of what kind of code this lead to, can be seen in Listing 3.

LISTING 3. list_take_n

```
void list_take_n(struct list *list, int n)
        //@ requires list(list, ?values) &*& 0 <= n
                &*& n <= length(values);
        //@ ensures list(list, take(n, values));
{
        //@ open list(list, values);
        list_reverse(list);
        //@ reverse_length(values);
        int drop = list_length(list);
        drop -= n;
        list_drop_n(list, drop);
        list_reverse(list);
        //@ reverse_drop_reverse(values, n);
}
```

This operation could be done in a single pass over the linked list: take n elements and free the rest of the nodes. Here, however, I reverse the list, then use the already implemented and verified drop function to cut off the beginning, and then reverse it again. This results in a total running time of $O(l + 2n)$ where l is the length of the list.
In hindsight, the single pass version might have produced less code since this implementation meant I had to prove that this actually implements the `drop` fixpoint function (lemma `reverse_drop_reverse`).

2.2.1. *Sublist.* Having written the take and drop functions, I could move on to implement the sublist method for arbitrary starting and endpoints by simply combining the two. The implementation was straightforward and resulted in no additional proofs. The signature is shown in Listing 4.

Listing 4. list_sublist

```
struct list *list_sublist(struct list *list, int i, int j)
        //@ requires list(list, ?v) &*& 0 <= i &*& i <= j
                &*& j <= length(v);
        //@ ensures list(list, v) &*& list(result,
                take(j-i, drop(i, v)));
```

2.2.2. *Equality.* To be able to compare strings, a comparison operation on the lists had to be defined. Unfortunately, I didn't find a way to do inductive proofs with the `==` relation. Therefore, I was forced to define my own recursive equality fixpoint function (`list_equals`) as well as the corresponding lemmas to convert to and from the `==` relation (lemmas `list_equals_to_equals` and `equals_to_list_equals`). This relation was then used in the `list_equals` function who's signature is shown in Listing 5.

Listing 5. list_equals

```
bool list_equal(struct list *l1, struct list *l2)
        //@ requires list(l1, ?v1) &*& list(l2, ?v2);
        //@ ensures list(l1, v1) &*& list(l2, v2) &*& result ==
            list_equals(v1, v2);
```

2.3. **Maximum Substring.** With the previously presented list functions implemented and proven, it was time to tackle the main problem. The first function I wrote and verified was a function that returns the maximum common substring starting at position 0 in both arguments.

Listing 6. list_maximum_sublist_0

```
struct list *list_maximum_common_sublist_0(struct list *list1,
                struct list *list2)
        //@ requires list(list1, ?v1) &*& list(list2, ?v2);
        //@ ensures list(list1, v1) &*& list(list2, v2)
                &*& list(result, ?v3)
                &*& list_maximum_common_sublist_0(v3, v1, v2);
```

That this function indeed returns the maximum can be seen if we take a closer look at the `list_maximum_common_sublist_0` predicate.

Listing 7. list_maximum_common_sublist_0 predicate

```
predicate list_maximum_common_sublist_0<t>(list<t> sublist,
        list<t> list1, list<t> list2)
        requires true == list_common_sublist(sublist,
                        length(sublist), list1, list2)
                &*& (false == list_common_sublist(
                        take(length(sublist)+1, list1),
                        length(sublist)+1, list1, list2)
                || length(sublist) == length(list1)
                || length(sublist) == length(list2));
```

The predicate takes the sublist as well as the two lists as argument. First of all, it ensures that the sublist is in fact a sublist of the two lists. This is implemented using the fixpoint function `list_common_sublist`. This predicate in turn, compares the three lists using the previously defined equality function over the length of the sublist.
To make sure that the sublist is maximal, the predicate also states that if the sublist were one element longer, it would no longer be a sublist. There are special cases when the sublist covers one of the arguments fully which are taken care of in the or clause.
The implementation of the function is not very efficient. It iterates over an integer $i \in 1, ..., n$ where n is the minimum length of the argument lists. For each i, it takes takes the sublists over the range $[0, i]$ and compares the two. If they are not equal the function aborts, otherwise it continues to the next i.

The `list_maximum_common_sublist` function computes, as its name suggests, the maximum common sublist. It does so by applying the `list_maximum_common_sublist_0` function to every possible pair of starting points in the two argument lists.

Listing 8. list_maximum_common_sublist

```
struct list *list_maximum_common_sublist(struct list *list1,
                struct list *list2)
        //@ requires list(list1, ?v1) &*& list(list2, ?v2);
        //@ ensures list(list1, v1) &*& list(list2, v2)
                &*& list(result, ?v3)
                &*& list_maximum_common_sublist(v3, v1, v2);
```

The corresponding predicate defines the maximum common sublist to be the maximum of three possible solutions. One is the solution where the sublist starts at position 0, in which case the result is the result of the `list_maximum_common_sublist_0` function. The other two solutions are the recursive application of the predicate onto the tail of one argument list, and its unchanged counterpart.

Listing 9. list_maximum_common_sublist predicate

```
predicate list_maximum_common_sublist<t>(list<t> sublist,
                list<t> list1, list<t> list2)
        requires list1 == nil || list2 == nil ?
        sublist == nil
:
        list_maximum_common_sublist(?vl, tail(list1), list2)
        &*& list_maximum_common_sublist(?vr, list1, tail(list2))
        &*& list_maximum_common_sublist_0(?vm, list1, list2)
        &*& length(vm) >= length(vl) && length(vm) >= length(vr)
        ?
                sublist == vm
        :
                length(vl) >= length(vr) ?
                        sublist == vl
                :
                        sublist == vr;
```

The implementation of the function follows the predicate definition closely.

3. Discussion

As far as I can see, my predicates are both sufficient and necessary for the longest common substring problem and since my implementation passes the verification[2] I am confident that it is correct as well.
I have not verified the code with the arithmetic overflow checks. However, the changes to the code to pass these tests as well would be minimal. Simply adding bound checks to all indices would probably be enough.

3.1. **Alternative Approaches.**

I doubt that the dynamic programming algorithm could have been implemented with the predicates I have provided. Although I have spent quite some time thinking about it, I still do not have a concrete idea about how one would prove the high level properties of such an algorithm.
The wikipedia article on the substring problem [4] mentions another algorithm using syntax trees which seems to be more suited to implement using Verifast but I have not had time to look further into the topic.

3.2. **Performance.**

As already hinted at in Section 2, the implementation is highly inefficient. Most of the functions, though, would be fairly easy to optimize. Verifast, however, makes this step problematic. As an example, consider the length function which iterates over the list while it counts the elements. A far more efficient implementation would simply store the current length in an integer field in the list `struct` and add (subtract) one when adding (removing) an element. This would make the operation run in constant time and, since the function is heavily used, propagate this speedup to many more functions. Unfortunately, this seemingly simple change would have to be reflected in the node predicate. This change in turn would propagate to every other predicate and from there on to every function.
Nonetheless, there are examples where the interfaces are unaffected. The `list_take_n` function could be replaced with a single pass function without any noticeable changes to the outside.

3.3. **Verifast.**

Although Verifast can be confusing at first, it gets easier and even fun over time. Time, though, is definitely Verifast's primary weakness. The time that has to be spent proving lemmas, that are at times trivial at best[3], can be tremendous. Partial automation would certainly help its cause.
Another weakness, that was a major obstacle especially at the beginning, is the lack of available documentation. A small example is the semi-colon separating the arguments in the node segment and list predicates. I know that it saved me from writing a lot of close statements, but I never found out, what precisely its purpose is. There is a tutorial paper which gives a nice introduction but is not very precise when it comes to explaining the relevant concepts.
Furthermore, the verification process seems to discourage the programmer from writing readable code. Once a function verifies there is hardly an incentive to go back and make improvements. The same holds for the proofs.

[2]The verification runs instantaneously in Verifast IDE 10.4 on a 3 GHz Core 2 Duo Processor.
[3]I had to prove several lemmas where a simple switch statement, without any instructions, was enough to get the proof to verify.

3.4. **Conclusion.** My experiences with Verifast are mixed. On the one hand, I was off to a frustrating start where the lack of documentation forced me to spend a lot of time studying examples instead of doing any actual work. On the other hand, knowing what to prove and how to prove it became easier and even fun after some time. Moreover, the "0 errors found" message the IDE displays upon successful verification is always a nice motivator.
The learning process lead to code that is not very consistent in quality, which is something that still bugs me personally. Due to the previously mentioned limitation of the process, correcting the deficiencies is not something that can be done quickly.

There is however always, and I would like to end on that note, the very comforting matter of fact that the code does **verify**...

References

[1] Verifast examples `http://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/`. [Online; accessed 27-November-2010].

[2] Verifast homepage `http://people.cs.kuleuven.be/~bart.jacobs/verifast/`. [Online; accessed 27-November-2010].

[3] Wikipedia. Longest common subsequence problem `http://en.wikipedia.org/w/index.php?title=Longest_common_subsequence_problem&oldid=389108052`, 2010. [Online; accessed 27-November-2010].

[4] Wikipedia. Longest common substring problem `http://en.wikipedia.org/w/index.php?title=Longest_common_substring_problem&oldid=398399925`, 2010. [Online; accessed 27-November-2010].