CS 2742 (Logic in Computer Science) – Winter 2014
Lecture 25

Antonina Kolokolova

March 26, 2014

8 Correctness of algorithms

Consider the following algorithm for finding an element in the array:

```
Max(A[1], ..., A[n])
    m =A[1]
    for i = 2 to n
        if A[i] > m then
          m = A[i]

    return m
```

What does it mean to prove that this algorithm works correctly? For that, we need to specify what are the starting conditions of the algorithm and what we expect it to output at the end. These conditions are called *preconditions* and *postconditions*. Note that we can talk about preconditions and postconditions for not just the whole algorithm but any piece of code, for example, a precondition and postcondition of the **for** loop.

So which assumptions do we make about the input to the algorithm $Max(A)$? First we are assuming that A is an array of integers, so comparisons such as $A[i] > m$ make sense. Second, since we start with our code with $m = A[1]$, we are using the assumption that the array has at least one element, that is, it is not empty. So the precondition for this algorithm is: "A is a non-empty array of integers". We use the variable n to denote $|A|$, to simplify the presentation.

A postcondition for this program would state that starting with the precondition, this code does the right thing, that is, it returns the maximal element in the array. What do we mean

by a "maximal element of the array", though? Suppose you say that the postcondition is "return m such that $\forall i, 1 \leq i \leq n,\ A[i] \leq m$. Is this enough? Well, somebody can write a piece of code that just sets m to be the largest integer representable on the system, then this postcondition will be satisfy, however, this is clearly not the kind of answer we want to get from a program searching for a maximal element in an array. The additional condition needed here is to ensure that m is indeed an element of the array, in addition to being as large as anything in A. So, to write it formally, we end up with a following postcondition: "return m such that $\exists j, 1 \leq j \leq n,\ A[j] = m$ and $\forall i, 1 \leq i \leq n,\ A[i] \leq m$." Now our postcondition does state that the algorithm returns the maximal element of A.

It is possible to put conditions (pre/postconditons) between any two lines of code, and then prove the correctness of the program by showing that for every line of code, if the preconditions of this line of code are satisfied then so are postcondition. This works well for statements such as "if... then...", but for loops we need a bit more machinery.

8.1 Loop invariants

Intuitively, to prove that the loop executes correctly, we want to show that it is correct on the first iteration, then on the second and so on until the last iteration.. And, moreover, we want to show that the loop terminates. The idea here resembles induction (and in fact it is induction that is used to prove the correctness of loops).

The predicate used to prove correctness of a loop is called a *loop invariant*. For each iteration of the loop, if this predicate $I(k)$ (where k stands for k^{th} iteration) is true before the iteration, the it is true after the iteration (on the changed values of the variables). Furthermore, the predicate $I(0)$ is true before the first iteration and loop terminates in finite number of steps and after the last step the truth of loop invariant ensures the post-condition of the loop.

Let $I(n)$ be a loop invariant, and call the condition which is checked at the start of wach loop (such as $i \leq n$ in **while** $i \leq n$) a *guard condition* G. Then the following theorem (due to Hoare) formalizes the properties of the loop invariant.

Theorem 1 (Loop invariant theorem.)**.** *Consider a while loop with a guard condition G. Let $I(n)$ be the loop invariant predicate. If the following are true, the loop is correct with respect to post- and pre-conditions.*

1) *Basic property: the pre-condition implies $I(0)$*

2) *Induction property: for all integers $k \geq 0$, if guard and $I(k)$ are true before the iteration, then $I(k+1)$ is true after.*

3) *Eventual falsity of the guard: after a finite number of iterations, G becomes false.*

4) Correctness of postconditions: if N is the first place where G is false, and $I(N)$ is true, then post-condition holds.

Example 1. Let us look at the loop invariant for the $Max(A)$ algorithm. To make the guard condition more interesting, consider the following modification of the algorithm:

```
Max(A[1], ..., A[n])
    m =A[1]
    i = 1
    while i ≤ n do
        if A[i] > m then
          m = A[i]
        end if
        i = i + 1
    end while

    return m
```

Let the precondition of the loop be $m = A[1]$ and $i = 1$ where the first element of A is $A[1]$. Let the postcondition of the loop be the same as the postcondition of the whole algorithm: that m is an element of A and it is the largest element. Now, the loop invariant is: $I(i) : m$ is the largest element among the first i elements of A. The guard condition G is $i \leq n$.

Now, we can prove the correctness of this loop by induction on i. First, note that since n is a fixed integer and i starts smaller than n, and it is incremented by i at each step, at some point $i \leq n$ will become false. This will always happen after $n - 1$ iterations.

Now, let us prove the loop invariant. First, $I(1)$ is true because $A[1] = m$, so it is among the first 1 elements of the array, and it is at least as large as $A[1]$ for the same reason. Now, for the induction hypothesis, suppose that m is the maximal element among $A[1] \dots A[i]$; let this be $I(i)$ (note that here we did not start with 0, changing the base case to a large number. But as long as our base case follows from the loop precondition, and does not miss any steps, it is OK). We want to show that after one more iteration $I(i + 1)$ will hold. Consider two cases: 1) $A[i + 1] > m$. In that case, $A[i + 1]$ is greater than all preceding elements of A, by transitivity of $\geq$ relation. Thus, setting m to $A[i + 1]$ satisfies both conditions: it is an element of A (its i+1'st element) and it is at least as large as all elements among $A[1]$ to $A[i + 1]$. Now, for the second case, suppose that $A[i + 1] \leq m$. In this case, m does not change. It is still in A by induction hypothesis: it is $A[j] = m$ for some $j \leq i$. It is greater than or equal to all elements of A from $A[1]$ to $A[i + 1]$ because it is $\forall j < i + 1,\ m \geq A[j]$ by induction hypothesis and $m \geq A[i + 1]$ because this is the case of the "if" that we are considering.

Finally, after the last iteration m is still in the array and it is at least as large as anything

in the whole array. Therefore, it trivially satisfies the postcondition of being the largest element in A.

8.2 Correctness of recursive algorithms

Many of you know the Binary Search algorithm for finding an element in a sorted array. The inputs are an array A of, say, integers and an another integer x, and the output is either the index of the array where that element is located (e.g, if for some i $A[i] = x$, then return i) or an error code (e.g., 0). Because the input array is sorted, it is possible to find x in the array (or check that it is not there) much faster then by checking every element in a way similar to what we did with Max. The BinarySearch algorithm does it as follows: it splits the array into two halves, and then checks if x is greater than the middle element. If so, it is clear that if x is in the array then it must be in the right half; if not, in the left half.

In this section, we will analyze the Recursive Binary search algorithm; the iterative version is left as an exercise. The program consists of two parts: MainBinSearch is a short program making the first recursive call to RecBinSearch, which is the part where all the work is done.

$MainBinSearch(A, x)$
    return $RecBinSearch(A, 1, |A|, x)$

Let's leave the precondition and postcondition for $MainBinSearch$ as an exercise, and look at the recursive part of the program.

The following code does the main part of the work.

$RecBinSearch(A, f, l, x)$
if $f = l$ **then**
  if $A[f] = x$ **then**
    return f
  else
    return 0
else
  $m = (f + l)/2$
  if $A[m] \geq x$ **then**
    return $RecBinSearch(A, f, m, x)$
  else
    return $RecBinSearch(A, m + 1, l, x)$
  end if
end if

RecBinSearch:
Precondition: $1 \leq f \leq l \leq |A|$ and $A[f..l]$ is sorted.
Postcondition: return t, $f \leq t \leq l$, such that $A[t] = x$, or, if x is not in the array between f and l, return $t = 0$.

It is easy to see that the precondition of *RecBinSearch* follows from the precondition for $MainBinSearch$ for $1 = f$ and $l = |A|$. Also, the postcondition of $MainBinSearch$ follows from the postcondition of *RecBinSearch* since the statements that x is somewhere between the first and the last element of the array and that it is somewhere in the array are equivalent.

It remains to prove correctness of *RecBinSearch*. The proof of correctness of *RecBinSearch* will proceed by strong induction on length of the array, $l - f$.

$P(k)$: if $1 \leq f \leq l \leq |A|$ and $|A[f..l] = k$ and $A[f..l]$ is sorted then the call terminates and returns t, $f \leq t \leq l$, such that $A[t] = x$, or, if x is not in the array between f and l, return $t = 0$.

Base Case: $k = 1$. Then $l = f$ so there is just one element in $A[f..l]$. RecBinSearch returns f if this element is x and 0 otherwise, and makes no further recursive calls.

Induction. Step. Strong induction:
Assume for all $1 \leq i < k$ $P(i)$ holds.
Then the algorithm makes one recursive call to *RecBinSearch*, with different parameters depending whether $A[m] \geq x$ and $A[m] < x$. The input to the algorithm is just a part of the array between f and l. As we said before, if $x > A[m]$ then it must be between $A[m+1]$ and $A[l]$; otherwise, it is in the other half. By induction hypothesis, recursive call will return us the correct answer which we just return. Here we are using strong induction because the sizes of the new arrays are half of the size of the old ones, and besides, due to rounding/odd length of the subarray it might be the case that the length of both halves is the same.

The last remaining part to prove is that the algorithm will terminate. Note that the algorithm halves the length of the array at every iteration, so eventually the size of the array will approach 1. It cannot become a 0 because the two halves always differ at most by 1 element, and so if one of them is a 0, then another must be 1, but in this case the array we are splitting would be already of length 1.