Using Dijkstra-Gries formal method to refine an informal program

Srinivas Nayak

Abstract

In this note, a detailed step-by-step refinement of an informally developed bidirectional bubble sort algorithm is presented. A less rigorous formal approach has been followed to refine an already developed algorithm to a provable one. Proofs of correctness that have guided the refinement steps have been provided along with the intermediate versions of the algorithm. It is concluded that, Dijkstra-Gries formal method can be easily used to refine an already developed program to a provable one.

Introduction

If someone meets a software developer, in introduction, he will obviously ask two obvious questions, "What do you do?" and "How do you do it?" I think, any software developer, with a little hesitation, will keep his answers simple, "modify existing code" and "in ad-hoc manner". If readers do not agree to both of my answers, it is a good sign! But I believe, for most of the developers, though they do not say it loud, in reality, they do the same. To my notice, scope for developing a new code from scratch is very less in software industries. Daily job of a software developer includes (a) refining the existing code when a bug is found, (b) copying necessary lines of source code from an already developed system (c) modifying existing source code to suit a new requirement. Popular terms for these activities are 'bug fixing', 'porting' and 'enhancement' respectively. No doubt, much time is spent playing around with existing code.

Saying this, it remains to see, how this job is accomplished. Large parts of the software developer community do their job in an ad-hoc manner. Since last forty years we have witnessed methodologies that enhance our ability to develop a program correctly and prove its correctness. These methodologies are called formal methodologies as opposed to ad-hoc method of developing program. Needless to say, ad-hoc method of developing program is still dominant in industries despite we have other better methods. In essence, in a typical software industry, everyday work practice includes modification or refinement of an existing code developed in ad-hoc manner.

It is been around a year I came across the formal method of developing programs in the works of Dijkstra and Gries (hereafter referred simply as formal method). One question always comes to my mind, can I apply this in my daily work? We have many examples in books, which show how to develop a program formally from scratch with the goal in mind and a proof of correctness guiding the development. But what about refining an existing code? When I deal with a piece of code that is already developed in ad-hoc manner, can I use this methodology to refine them? Searching an answer to this, I thought of applying the formal method to refine an already developed program, which I have developed in ad-hoc way (as I usually do!). My experience through this little experiment has lead to this note which describes in detail the steps carried out in refining a sorting algorithm. I have devoted separate sections for each step I have followed in refining my algorithm. Each section starts with finding a construct in the program to refine, then describes application of formal method to find a correct construct, discusses verification of

new construct against the correctness criteria and finally presents a proof of correctness for the algorithm developed till then.

A sorting algorithm

While I was going through the quick sort algorithm, I planned to design an algorithm that is similar to quick sort but with a little different idea. In quick sort algorithm, we pick an arbitrary element (pivot), proceed to place the pivot in its actual position in the list (position where the element would be, had the given list been sorted) and then, we apply quick sort recursively to both the sub-lists on two sides of the pivot. Essentially, here we choose only one element as pivot and we continue till we place it in its actual position and then, since the pivot is in proper place for the sorted list, we worry about all other elements rather than the pivot.

I wanted to make two changes to the above idea. Firstly, I will not try to place my chosen element in its actual position in one iteration. I am satisfied, if in a single iteration, my chosen element makes some progress in moving towards its actual position. Secondly, I decided not to choose any single element as pivot, so that, in a single iteration, I may try to move any number of elements towards their actual positions.

With this idea, I am sure that, in every iteration I am making some progress towards the completion of sorting and for the list to be totally sorted, I may go on many such iterations.

My algorithm

I quickly wrote down the first version of my algorithm which maps to my idea outlined above.

Version 0:

```
//sorts an array of n+1 elements in ascending order.
my_sorting_algorithm()
{
      u := 0;
      d := n;
      while(1)
      {
            if( sorted ) exit;

            // Added after I debug the program with computer
            if( u=n /\ d=0 )
            {
                  u := 0;
                  d := n;
            }

            // L-R Loop
            p := u;
            u := u+1;
            while( u<n /\ a[p]>a[u] )
            {
                  swap(a[p], a[u]);
                  p := p+1;
                  u := u+1;

            }
```

```
            // R-L Loop
            q := d;
            d := d-1;
            while( d>0 /\ a[q]<a[d] )
            {
                  swap(a[q], a[d]);
                  q := q-1;
                  d := d-1;

            }
      }
}
```

I hope a little explanation to this algorithm here will be in order. This algorithm sorts an array a of n+1 elements in ascending order. I have chosen n+1 elements in my array, only because I can refer to the last element of my array simply as a[n] rather than a[n-1].

In L-R Loop, p points to my chosen element and u points to its next element. Variable u will hold values from range 0 to n. Whenever I found my chosen element (pointed by p) greater than its right neighbor (pointed by u), I swap them. Now, by increasing p and u, I again pick my earlier chosen element. By repeating the same compare and swap strategy, I keep moving my chosen element towards its actual position. If I can't carry my chosen element further, I choose its next element and proceed. The same plan has been devised for the R-L Loop. Once I complete my L-R Loop and R-L Loop, I am sure that some progress has been made towards sorting, that is, the array is now somewhat sorted. I need to repeat the same until my array is fully sorted. For checking whether the array is sorted or not, I simply employed a small subroutine which compares each pair of elements in the array to decide, if the array is sorted or not. When the subroutine returns true, the array is sorted and I exit the loop. In my algorithm, I have used 'sorted' to denote such a device that lets me know when the array is sorted.

[Note] At a first look, use of a subroutine to check the sortedness seems awkward. When I explained this algorithm to my friend, immediately he came up with an idea to use a flag instead of subroutine. The flag, which is initially true, shall be changed to false, in case we enter any of the inner loops, denoting the array is still not sorted. We will check the flag after exiting from the second inner loop. If it is still true, we will exit from the outer loop, otherwise we will continue executing.

Initially, this scheme looked promising. Little after, when I analyzed our new scheme thoroughly, I found that it will not work. Why? Readers can use the array {1, 2, 5, 3, 4} to see why this scheme doesn't work. [End of the Note]

So far so good. I am happy that my algorithm maps exactly to my idea, but how do I check, if my algorithm is correct? I immediately wrote a program for it and sat in front of my computer. As usual, it failed. It was an infinite loop! I put debug statements all over the body of my program and finally I found that I am not re-initializing the variables u and d after they reached their maximum and minimum values respectively. I should continue picking elements and pushing them towards their actual positions and for that reason I have to re-initialize u to 0 and d to n. The same needs to be done as long as the array is not sorted.

I was satisfied that my algorithm was able to sort all the lists which I have given it as input. However, the correctness of the algorithm can not be judged by a couple of test runs. Now I was looking for a proof which can convince myself --and also my friends-- that my algorithm is correct. I was excited that I have discovered a variant of quick sort (because, the idea was born while I was reading quick sort!) and its proof was crucial for me to convince my friends that I have discovered an algorithm that really sorts! I started refining my algorithm for its provability. My sincere thanks to Dr. David Gries for pointing out that my algorithm is a variant of bubble sort and not of quick sort.

Formalizing inner loops

Lets clearly visualize the problem at hand. Sorting in ascending order can be seen as a process of finding a configuration (of the array elements) which is lexicographically smallest of all the other possible configurations. We keep values of all the elements unchanged while sorting. This definition is more formal than the common idea about sorting.

In refining my algorithm, I wanted to focus on the inner loops first. To formalize and reason about the loops I need to find Hoare triples for them. Keeping other parts of the algorithm unchanged, I started refining the first inner loop (L-R Loop). When we look a bit deep into the sorting process, we find that, just before the inner loops, the arrangement of elements inside the array is changed where as the initial values of array elements remains unchanged. Hence the precondition for the L-R Loop can be:

Q1: A1=perm(a,A).

This states that, considering array A[0:n] as the initial configuration of values of the given array a[0:n], A1 is a permutation of A.

Now I have to choose the postcondition for the L-R loop. Since I expect my loop to make some progress towards sorting, the postcondition is immediate:

R1: A1=perm(a,A) /\ A1<=A.

The second clause states that the new configuration A1 is lexicographically smaller than or equal to the initial configuration A.

The next task was to find the invariant for the L-R Loop; a assertion that is true all throughout the loop execution as well as before the loop starts and after the loop execution ends. I had chosen my loop condition to be

B1: (u<n /\ a[p]>a[u]).

With a hope that this condition may give me some insight on finding the invariant, I wanted to have a close look at it. I found two things:

1. First clause u<n remains true whenever the loop is entered. At the end of the loop, u becomes same as n. So, the predicate u<=n remains true throughout the loop execution.

2. When the second clause a[p]>a[u] is true, we swap a[p] and a[u]. This only changes the configuration of the array a and the new configuration A1 is just a permutation of the original configuration A. Hence

A1=perm(a,A) is true when the loop is executed. When the loop ends or the clause a[p]>a[u] is false, A1=perm(a,A) still holds true.

Another beautiful observation surfaces itself immediately from the second one.

3. We can see that after the swapping is done, the bigger element shifts to right and smaller element shifts to left. This causes the new configuration (a[u],a[p]) to be lexicographically smaller than the old configuration (a[p],a[u]). This makes A1<A. In case a[p] and a[u] are same, A1 equals A. So A1<=A is true as long as the loop is executing. A1<=A is true also when the loop starts and ends.

And finally the assignment statements that increments the values of the variable p and u gives us:

4. p is always less than u whenever the loop starts, gets executed and ends.

Now combining the observations from 1 to 4 I can frame my invariant as:

P1: u<=n /\ p<u /\ A1=perm(a,A) /\ A1<=A.

The last thing I have to find is the bound function. In L-R Loop, we operate on the array going from start to end of it till u becomes n. So I could choose my bound function to be:

t1: n-u.

When the loop ends, t1 is 0 which is as desired. Following exactly the similar steps, I find Q2, R2, P2 and t2 for the second inner loop (R-L Loop). With all the above findings, my next version of algorithm looks as follows.

Version 1:

```
//sorts an array of n+1 elements in ascending order.
bidirectional_bubble_sort()
{
      // a[0:n] = A[0:n]

      u := 0;
      d := n;

      while(1)
      {
            if ( sorted ) exit;

            if( u=n /\ d=0 )
            {
                  u := 0;
                  d := n;
            }

            // Bubble-Up Loop
            p := u;
            u := u+1;

            // precondition   Q1 : A1=perm(a,A)
            // invariant      P1 : u<=n /\ p<u /\ A1=perm(a,A) /\ A1<=A
            // bound function t1 : n-u
```

```
            while( u<n /\ a[p]>a[u] )
            {
                  swap(a[p], a[u]);
                  p := p+1;
                  u := u+1;
            }
            // postcondition  R1 : A1=perm(a,A) /\ A1<=A

            // Bubble-Down Loop
            q := d;
            d := d-1;

            // precondition   Q2 : A2=perm(a,A)
            // invariant      P2 : d>=0 /\ q>d /\ A2=perm(a,A) /\ A2<=A1
            // bound function t2 : d
            while( d>0 /\ a[q]<a[d] )
            {
                  swap(a[q], a[d]);
                  q := q-1;
                  d := d-1;
            }
            // postcondition  R2 : A2=perm(a,A) /\ A2<=A1
      }
}
```

To see the correctness of loops, we shall verify whether our loops satisfy the following criteria.

A loop with precondition Q, postcondition R, invariant P, loop condition B and bound function t is correct, when
1. Invariant P is true before execution of the loop begins.
2. Invariant P remains valid after execution of the loop body.
3. P /\ ~B => R, i.e. desired result is obtained after the loop execution terminated.
4. P /\ B => t>0, i.e. number of loop iteration is bounded
5. Each loop iteration decreases bound function t.

By initializing u to 0, initializing p to value of u and incrementing u, we reach a state just before the Bubble-Up loop that satisfies the first criterion. Verification of second criterion is also easy. Inside the Bubble-Up loop, we only swap two elements of the array, that will leave the array as a permutation of the earlier configuration and incrementing p and u will satisfy the clause p<u. This shows that the second criterion is met.

According to the third criterion, loop invariant together with loop termination condition shall provide us the desired result. Fortunately, here in this case, our loop invariant is stronger than the postcondition, and since the loop invariant remains valid after the loop exits, we are sure that postcondition remains true at any cost.

When loop condition is true, we can see that, u is less than n. This implies that bound function n-u is a finite number and greater than 0. This shows that the Bubble up loop satisfies the forth criterion of correctness. Finally, we could see that at each iteration, we decrement the value of u inside the loop body. This satisfies the last criterion. I hope, the convincing proof laid out above shall now allow us to say that the above algorithm is correct.

Even though the current version of algorithm has no logical refinement over the earlier version, it is more formal which allows us to prove its

correctness and there by can serve as a stronger base for our further refinements.

Refinement of formal version

Let's now look at the formal version of the algorithm for some refinements. We will again pick up the inner loops and try to find if we could refine them further. We see two immediate improvements.

1. Our bound function in case of the Bubble-up loop is only a promising one. In the sense, it just assures the loop to iterate only a finite number of times. But truly, the loop may terminate much before the bound function reaches its lower boundary. In case we have a non-sorted array given (this is true often!), as soon as we have two elements that satisfy a[p]>a[u], the loop terminates. This shows that our loop really does not make a single pass of element-checking from begriming to end of the array, rather it terminates much before. This can be rectified if we deploy the first clause of B1 as our loop condition and the second clause as the condition for swapping inside the loop.

As a further improvement, following the best practice, we could make our loop condition a weakest one, u#n. Our new loop will look like as follows.

```
while( u # n )
{
      if ( a[p]>a[u] ) -> swap(a[p], a[u]) fi
      p := p+1;
      u := u+1;
}
```

We could weaken our loop invariant to remove the clause u<=n from it, which has already taken the role of loop condition. Still, our loop condition together with the invariant implies the post condition.

The similar we can do for the Bubble-down loop.

2. As another improvement, we could avoid the unnecessary assignment statements inside and outside of the outer loop. Since our loop condition has been changed, we could see that, when the loop terminates, u and d has value n and 0 respectively. This helps us removing the reinitialization of the variables and as a replacement; we could always initialize the variables u and d inside the outer loop. This again helps us to remove the initialization outside the outer loop. Now our new algorithm will look like as follows.

```
bidirectional_bubble_sort()
{
      // a[0:n] = A[0:n]
      while(1)
      {
            if ( sorted ) exit;

            u := 0;
            d := n;

            // Bubble-Up Loop
            p := u;
            u := u+1;
            ...
```

```
            // Bubble-Down Loop
            q := d;
            d := d-1;
            ...
      }
}
```

As a further enhancement, if we like, we can club the assignments u := 0; p := u; u := u+1; together to make a single multiple assignment statement p,u := 0,1; and club the assignments d := n; q := d; d := d-1; together to make another single multiple assignment statement q,d := n,n-1;.

Please note that, we could do so, because we never change the value of d till we begin the Bubble-down loop.

With these improvements, our next version of the algorithm will be as given below.

Version 2:

```
//sorts an array of n+1 elements in ascending order.
bidirectional_bubble_sort()
{
      // a[0:n] = A[0:n]
      while(1)
      {
            if ( sorted ) exit;

            // Bubble-Up Loop
            p,u := 0,1;

            // precondition   Q1 : A1=perm(a,A)
            // invariant      P1 : p<u /\ A1=perm(a,A) /\ A1<=A
            // bound function t1 : n-u
            while( u # n )
            {
                  if ( a[p]>a[u] ) -> swap(a[p], a[u]) fi
                  p := p+1;
                  u := u+1;
            }
            // postcondition  R1 : A1=perm(a,A) /\ A1<=A


            // Bubble-Down Loop
            q,d := n,n-1;

            // precondition   Q2 : A2=perm(a,A)
            // invariant      P2 : q>d /\ A2=perm(a,A) /\ A2<=A1
            // bound function t2 : d
            while( d # 0 )
            {
                  if ( a[q]<a[d] ) -> swap(a[q], a[d]) fi
                  q := q-1;
                  d := d-1;
            }
            // postcondition  R2 : A2=perm(a,A) /\ A2<=A1
      }
}
```

Removal of unnecessary variables

Although our current version of algorithm is simple, still we have further scope to make it simpler. A close observation reveals that, we use four variable in our algorithm out of which two are used only to keep track of the neighboring element. For instance, in Bubble-Up loop p and u keeps track of two neighboring elements from which we could retire variable p, since we can represent the next element of an element say a[u], just by specifying a[u+1]. This allows us to remove both the variables p and q from our algorithm. We initialize u and d to 0 and n respectively. We specify the next element of a[u] as a[u+1] and previous element of a[d] as a[d-1].

This improvement gives us another opportunity to simplify our loop invariants to become
P1: A1=perm(a,A) /\ A1<=A
P2: A2=perm(a,A) /\ A2<=A1.

With this improvement our algorithm takes the form as below.

Version 3:

```
//sorts an array a[0:n] in ascending order.
bidirectional_bubble_sort()
{
      // a[0:n] = A[0:n]
      while(1)
      {
            if ( sorted ) exit;

            //Bubble-Up Loop
            u := 0;

            // precondition   Q1 : A1=perm(a,A)
            // invariant      P1 : A1=perm(a,A) /\ A1<=A
            // bound function t1 : n-u
            while( u # n )
            {
               if ( a[u]>a[u+1] ) -> swap(a[u], a[u+1]) fi
               u := u+1;
            }
            // postcondition  R1 : A1=perm(a,A) /\ A1<=A


            //Bubble-Down Loop
            d := n;

            // precondition   Q2 : A2=perm(a,A)
            // invariant      P2 : A2=perm(a,A) /\ A2<=A1
            // bound function t2 : d
            while( d # 0 )
            {
               if ( a[d-1]<a[d] ) -> swap(a[d-1], a[d]) fi
               d := d-1;
            }
            // postcondition  R2 : A2=perm(a,A) /\ A2<=A1
      }
}
```

Formalizing outer loop

Let us now turn our attention to the outer loop. In the current version of our algorithm, our outer loop iterates forever. By saying this, I am emphasizing the nature of the loop condition ('true') which is never false to stop the loop. But we want the loop to stop when our list is sorted. To this end, we have provided an if-statement at beginning of the loop which stops the loop whenever 'sorted' (implemented currently as a subroutine) is true.

To start formalizing our outer loop and to keep our formalization simple, let us ask ourselves two simple questions, do we really need a subroutine to check the sortedness? Can't a simple variable do the job? In our current version of algorithm, we have still kept 'sorted' as an abstract condition. To have a complete implementation, I had used a subroutine to satisfy the need. At any rate, we are free to choose a different implementation if we like to do so. Since a subroutine will be heavier, let's look for some other options. Keeping the same meaning of our abstract notion, it will not be wrong if we consider 'sorted' as a boolean variable whose job is to inform us whether the given list is sorted or not. For the sake of completeness, let's make it explicit that when the list is sorted, 'sorted' shall have the value 'true' and when the list is not sorted, 'false'. Initially 'sorted' can have a value 'false' to denote that we assume the given list is not sorted. We will change its value to 'true' when we find the given list sorted. It is easy to see that, we can now employ a suitable form of 'sorted' as the condition for our outer while loop. This immediately removes foreverness from our loop structure, makes program simple by removing the extra if condition inside while loop and still satisfies all our needs.

With this idea in mind, let's specify precondition and postcondition for the outer loop formally. Our precondition is nothing but,

Q0: a[0:n] = A[0:n]

which means, the initial configuration of elements in the given array is denoted by the array A whose elements are the initial values of a's elements.

As we know, we have to sort the given list without changing its content and as we said earlier, our helping variable 'sorted' shall inform us when we have completed sorting. So our post condition may be

R0: perm(a, A) /\ (sorted => a[0..n] is in ascending order).

We need not have to search for the invariant our loop. We know, our array content shall not be changed when the loop executes. So we can have the following invariant.

P0: perm(a, A).

And what could be the loop condition? As we know, our loop to be correct, P /\ ~B => R should hold. From this we get the loop condition,

B0: ~sorted

which says, the loop shall run, until the list is unsorted and shall stop as soon as the list is sorted.

Now we have to find our bound function. We can see that, whenever we have our list unsorted, we need to swap elements to bring it back in

order. How many such inversions (swapping of elements) will finally bring our list sorted? Let us hope that it will be a finite number, since our list is finite. But just saying this is not enough. If we examine carefully, we find that we treat the process of ascending order sorting as a process of finding a configuration (of elements) which is lexicographically smallest among all the configurations that might exist. With each inversion we bring up a configuration that is lexicographically smaller than the previous configuration. Since the number of configuration for a finite number of elements is finite, the number of steps to find the smallest configuration is also finite. It seems that 'number of inversions' has a upper bound and can be used as bound function of our outer loop if the loop body doesn't bring up a new configuration that is greater than the previous one.

The bound function must satisfy the last two criteria of loop correctness which we have followed in the section "Formalizing inner loops". If we carefully check the validity of proposed bound function -- number of inversions-- against the forth criterion, we can see that, P /\ B => t>0 may not hold for some case. Which case it fails?

If the input array is already sorted, number of inversions needed to sort the array is zero and so P /\ B => t>0 is false. We can observe, initialization of 'sorted' to a 'false' value makes the antecedent true in this case. One may complain that this initialization is not correct in case the given array is already sorted. By initializing 'sorted' to 'false' before the loop, we do not declare that the given array is unsorted. As we said earlier, this only denotes our assumption about the sortedness of array which we confirm after processing the whole array at least once. This shows, our proposed bound function shall be of no help, because it will not allow us an extra iteration of the loop which we use to check and confirm sortedness of the given array irrespective of whether it is initially sorted or not.

We may think of a different scheme for our bound function where it is expressed as an ordered pair. A promising bound function for our need could be an ordered pair (~sorted, number of inversions). As with the case ordered pairs, the decreasing order of values of our bound function will be (true,n), (true, n-1), (true, n-2),..., (true, 0), (false, 0). It is easy to find that this allows us an extra iteration for confirming the sortedness after number of inversions becomes zero, i.e. all the elements are sorted. With the bound function

t0: (~sorted, number of inversions in a[0..n])

we can see that P /\ B => t>0 always holds because when B is true, value of t is (true, 0) and still has not reached its lowest boundary (false, 0). Moreover, each iteration of our outer loop will decrease t because, in an iteration we either find a new configuration that is smaller than the previous, there by decreasing the number of inversions needed further or we change the value of 'sorted' from false to true, there by effectively decreasing the value of ~sorted from true to false. This satisfies the last two criteria for loop correctness.

With this, the structure of our outer loop looks as follows.

```
boolean sorted := false;
// precondition   Q0 : a[0:n] = A[0:n]
// invariant      P0 : perm(a, A)
// bound function t0 : (~sorted, number of inversions in a[0..n])
```

```
while( ~sorted )
{
      //Bubble-Up Loop
      ...

      //Bubble-Down Loop
      ...
}
// postcondition  R0 : perm(a, A) /\ (sorted => a[0..n] is in ascending
order.
```

Determining the sortedness

The remaining task is to assign 'sorted' an appropriate value according to the sortedness of our array. To know if our list is sorted, we need to (at least once) compare each neighboring elements occurring in the list to check the sortedness. If we find, at any time, two consecutive elements are not in order, we may declare that the array is not sorted. If we see our algorithm, both the inner loops check the whole array with an additional swapping in case they find any two elements unsorted. So, hopefully we can use them to determining whether our loop is sorted or not. For this, we may assign 'sorted' a value 'true' just before the second loop and if in case the inside if condition executes, we may assign a value 'false' to 'sorted' denoting the unsortedness.

We need to change the postcondition of Bubble-Down loop to show the fact that this loop determines the sortedness of array assigning proper values to 'sorted'. The new post condition may look like

R2: A2=perm(a,A) /\ A2<=A1 /\ (sorted => a[0..n] is sorted).

And accordingly our new invariant shall be

P2: A2=perm(a,A) /\ A2<=A1 /\ (sorted => a[d..n] is sorted).

[Note that the consequence of last clause in the invariant is 'a[d..n] is sorted' as opposed to the consequence of last clause in the postcondition which is 'a[0..n] is sorted']

We shall be able to show the correctness of the Bubble-Down loop which we just modified for a new postcondition. We can verify the correctness criteria against this loop. We can easily show that criterion 1 is met, since d has a value n before the loop starts. Criterion 2 is met because, in case the if-statement is executed, we have a 'false' value for 'sorted' and since 'false' implies any consequence, P2 remains valid. We can see that when d becomes 0, R2 is implied from P2, because we just need to place 0 for d in P2. Here we satisfy the criterion 3. Since our bound function of the Bubble-Down loop d is not equal to zero at the beginning of the loop, number of loop iterations is bounded; this satisfies criterion 4. Finally, Bubble-Down loop decrease the bound function d by one on each iteration. This satisfies our criterion 5. Keeping the Bubble-Up loop untouched and making changes to Bubble-Down Loop, our final version of sorting algorithm looks as follows.

Version 4:

```
//sorts an array of n+1 elements in ascending order.
bidirectional_bubble_sort()
{
      boolean sorted := false;
      // precondition   Q0 : a[0:n]=A[0:n]
```

```
      // invariant      P0 : perm(a, A)
      // bound function t0 : (~sorted, number of inversions in a[0..n])
      while( ~sorted )
      {
            //Bubble-Up Loop
            u := 0;

            // precondition   Q1 : A1=perm(a,A)
            // invariant      P1 : A1=perm(a,A) /\ A1<=A
            // bound function t1 : n-u
            while( u # n )
            {
               if ( a[u]>a[u+1] )
               {
               swap(a[u], a[u+1]);
               }
               u := u+1;
            }
            // postcondition  R1 : A1=perm(a,A) /\ A1<=A

            //Bubble-Down Loop
            d := n;
            sorted := true;

            // precondition   Q2 : A2=perm(a,A)
            // invariant      P2 : A2=perm(a,A) /\ A2<=A1 /\ (sorted =>
a[d..n] is sorted)
            // bound function t2 : d
            while( d # 0 )
            {
               if ( a[d-1]<a[d] )
               {
               swap(a[d-1], a[d]);
               sorted := false;
               }

               d := d-1;
            }
            // postcondition  R2 : A2=perm(a,A) /\ A2<=A1 /\ (sorted =>
a[0..n] is sorted)
      }
      // postcondition  R0 : perm(a, A) /\ (sorted => a[0..n] is in
ascending order
}
```

Here we shall show the correctness of outer loop and hence of the whole program. We can say, our program correct if our outer loop satisfies the above five criteria against which earlier we have checked the other loops. We can see that our loop invariant P0 is valid before the outer loop starts. This is in accordance with our first criterion for correctness. Second criterion demands us to show that invariant P0 remains valid after execution of loop body. This criterion is met, because we only perform swap operations in the loop which always leaves array a to be a permutation of its initial configuration A. According to third criterion, at the exit of loop, we must have obtained desired result i.e. the post condition must be valid. When ~B0 is true, i.e. 'sorted' has a value 'true', the loop must have iterated at least once, since 'sorted' was initially 'false'. This implies, postcondition of Bubble-Down loop is valid, which in turn implies the validity of R0. Earlier in this section we have already shown that our bound function

meets the forth and fifth criteria there by guarantying the termination of our outer loop. This proves the correctness of our algorithm.

Discussions

Successful application of formal method in refinement of a program can be judged from the easiness of its application and the benefit it results. Formal method is generally applied top-down in constructing a new program. When we have a program already developed, refinement is better to start bottom-up. This seems intuitive because we need to preserve the logic of implementation as well as the design. Following bottom-up approach we started with refinement of the inner loop followed by the outer loop. Refinement process that we have followed here repeats two fundamental steps, (1) correcting the lowest level constructs such as inner loops or conditional statements and (2) optimizing the code around the correct construct developed in step 1. For the first step, we use formal correctness criteria to modify components of a program construct, for example, loop condition, invariant, condition of the conditional statement etc. Once that is done, second step only looks for opportunity to optimize the program keeping the correctness of program intact, for example, removing unnecessary/unused variables, removing/modifying unnecessary assignments etc.

Benefits of refinement using formal method are easily seen comparing initial version of the algorithm with the final version. Among the visible benefits, we were able to remove unnecessary variables (p and q), unnecessary assignment statements and two conditional statements from our initial version of the program. Many of the other benefits are hidden. Although it is not visible from the text program, it is not very hard to spot them if we follow every refinement steps that we have carried out.

1. If we observe carefully, we will notice that our initial version of algorithm will not be able to use a simple boolean variable like 'sorted', to notify whether the array is sorted or not, at the end of an outer loop iteration. That forced me to use a subroutine for the checking of sortedness and quit. A big performance benefit is gained by using a boolean variable in place of a subroutine.

2. The idea of reinitializing the variables u and d which we have inserted in our algorithm after a small debug session is no more needed in the final version. Inner loops are now guaranteed to iterate for n times and terminate with u and d with values n and 0 respectively. This allows us not to differentiate between initialization and reinitialization.

3. Our inner loops and their termination conditions are now simpler to logically reason about them. Unlike their counter parts in our earlier ad-hoc treatment, these inner loops now have specific purposes with respect to the guaranteed termination of outer loop. Where the second inner loop helps us proving the guaranteed termination of outer loop, first inner loop helps in speeding up the sorting process.

4. After all, the most important benefit with the final version of the program is that it can be easily proved to be correct where as its ad-hoc counter part can not. While refining the algorithm, we developed the invariants and bound function as a by product which will help us to prove its correctness immediately. In comparison to this, our initial version with more program constructs and undefined bound functions will never be easier to prove.

5. Important but the least visible benefits being a clearer understanding of the sorting process and a compact algorithm with program execution steps that exactly match with the mental process of sorting with the conceived idea.

Conclusion

We started refining an already developed sorting algorithm to see if the formal method can be used to assist us while modifying an existing code for its correctness. In all steps of refinement process, keeping the idea behind algorithm intact, parts of algorithm were refined for their correctness. We used formal correctness criteria to develop and judge the intermediate versions of algorithm in each step of refinement. The result was a correct, provable program that is much simpler and compact than its informal version. Besides the correctness of algorithm, better execution performance of the program is also resulted. We believe that formal method can be successfully used to refine existing programs. Using correctness criteria as check list, a developed algorithm can be modified to suit the correctness need, keeping the logic of implementation unchanged.

Acknowledgements

I am indebted to Dr. Gries for his help in formalizing the outer loop presented here. Apart from some cosmetic changes, his idea of formalizing this loop is presented here as is. My failure in attempting to formalize the same with a wrong bound function has been discussed here for the purpose of "learning from the mistakes". Thanks are due to him for his constant help in my exploration of the science of programming.

References

[1]Gries, D. The Science of Programming, Springer-Verlag, New York, 1981.
[2]Dijkstra, E.W. A Discipline of Programming, Prentice Hall, Englewood Cliffs, 1976.

18 January 2009
Motorola India Pvt. Ltd.
Hyderabad, India