CS292C Homework 1

Deepak Nathani

April 10, 2024

1 Self-Grade

Problem	Self-Grade
1: Dafny Installation	2
2: Minimum of 2D Array	2
3: Binary Search	2
4: Buggy Binary Search	2
5: Bubble Sort	1

2 Problems

Problem 1. Dafny Installation

Install Dafny by following these instructions. We recommend simply installing the Dafny VSCode extension. If you're on macOS or Linux, you will be prompted in VSCode to install .NET 6.0 once you open a .dfy file. Simply follow the link in the prompt to install it.

You should install Dafny 4. If you're prompted by VSCode to upgrade from Dafny 3 to Dafny 4, you should let it automatically upgrade.

The whole process should be fairly quick, taking no more than 5 minutes. Once you're done, run Dafny on the demo file we covered during the tutorial, and attach the output to your submission PDF.

Solution

```
    tutorials/01-dafny/demo.dfy(19,4): Error: a postcondition
    could not be proved on this return path
    |
 19 |     return b; // BUG!
    |     ^^^^^^

    tutorials/01-dafny/demo.dfy(16,10): Related location: this is
    the postcondition that could not be proved
    |
 16 |   ensures c >= a && c >= b && (c == a || c == b)
    |           ^^^^^^

 Dafny program verifier finished with 4 verified, 1 error
```

Problem 2. Minimum of 2D Array

In min.dfy, you will find a Dafny method that finds the minimum of a 2D array. Your tasks are:

(a) State the correctness property of this method in terms of pre-condition(s) (using the requires keyword) and post-condition(s) (using the ensures keyword).

Your pre-condition(s) should allow Dafny to prove that there is no out-of-bound array access.

(b) Annotate each of the two loops with appropriate invariants that allow Dafny to verify the overall correctness of the method.

Hint

> The invariant for the outer should be very similar to your post-condition. For the inner loop, you may need to provide two different invariants: the first invariant talks about all elements from sub-arrays a[0] up to a[i-1] inclusive, and the second invariant specifically looks at the sub-array a[i] and talks about its elements between 0 and j-1 inclusive. It may help draw a dummy 2D array on paper, and mark the elements that have been processed so far when the loop is at (i,j)-th position. Those processed elements will be the ones (and the only ones) that are mentioned in the invariants.

(c) Attach your code with the annotations in your submission. Also, informally explain what your pre- and post-conditions, and loop invariants mean in plain English.

You should not change the implementation code in any way.

Solution

Part (a)

Pre-Condition: Check that the array length is greater than 1 and length of each subarray is greater than 1. This means that we require at least one element in the 2-D array.

$$\text{requires } a.Length \geq 1 \land \forall i : 0 \leq i < a.Length \Rightarrow a[i].Length \geq 1$$

Post-Conditions:

1. Ensure that the returned value is not larger than any other element in the array.

$$\text{ensures } \forall i, j : 0 \leq i < a.Length \land 0 \leq j < a[i].Length \Rightarrow a[i][j] \geq min$$

2. Ensure that the minimum value exists in the array.

$$\text{ensures } \exists i, j : 0 \leq i < a.Length \land 0 \leq j < a[i].Length \land a[i][j] = min$$

Part (b)

Outer Loop:

1. Ensure that the current minimum value is not larger than any other element in the array we have scanned so far.

$$\text{ensures } \forall k, l : 0 \leq k < \text{current index} \land 0 \leq l < a[k].Length \Rightarrow min \leq a[k][l]$$

2. Ensure that the minimum value exists in the array.

$$\text{ensures } \exists k, l : 0 \leq k < a.Length \land 0 \leq l < a[k].Length \land a[k][l] = min$$

Inner Loop:

1. Ensure that the current minimum value is not larger than any other element in the array we have scanned so far. We need two invariants for this. One to examine the subarray from $a[0]$ to $a[i-1]$ and then examine the current subarray $a[i]$ up to $j-1$ index.

$$\text{ensures } \forall k, l : (0 \leq k < i \land 0 \leq l < a[k].Length) \Rightarrow min \leq a[k][l]$$

$$\text{ensures } \forall l : (0 \leq l < j) \rightarrow a[i][l] \geq min$$

2. Ensure that the minimum value exists in the array.

$$\text{ensures } \exists k, l : (0 \leq k < a.Length \land 0 \leq l < a[k].Length) \land (a[k][l] = min)$$

Part (c)

```
method min2D(a: array<array<int>>)
returns (r: int)
// assume this function can only be called with a non-empty array
requires a.Length >= 1 && forall i :: 0 <= i < a.Length ==> a[i].
   ↪ Length >= 1

// ensure r is no larger than any element in the array
ensures forall i, j :: 0 <= i < a.Length && 0 <= j < a[i].Length ==>
   ↪  (r <= a[i][j])

// ensure r exists in the array
ensures exists i, j :: (0 <= i < a.Length && 0 <= j < a[i].Length)
   ↪ && (a[i][j] == r)
{
  var min := a[0][0];
  for i := 0 to a.Length
  // this invariant is for proving the first ensure statement
  invariant forall k, l :: (0 <= k < i && 0 <= l < a[k].Length) ==>
     ↪ min <= a[k][l]

  // this invariant is for proving the second ensure statement
  invariant exists k, l :: (0 <= k < a.Length && 0 <= l < a[k].
     ↪ Length) && (a[k][l] == min)
  {
    for j := 0 to a[i].Length

    // first write an invariants for the subarrays a[0] to a[i-1]
       ↪ inclusive
    invariant forall k, l :: (0 <= k < i && 0 <= l < a[k].Length)
       ↪ ==> (a[k][l] >= min)

    // then write an invariant specifically checking for the
       ↪ subarray a[i] upto j-1 index
    invariant forall l :: (0 <= l < j) ==> (a[i][l] >= min)

    // invariant to prove the second ensure statement
    invariant exists k, l :: (0 <= k < a.Length && 0 <= l < a[k].
       ↪ Length) && (a[k][l] == min)
    {
      if a[i][j] < min {
        min := a[i][j];
      }
    }
  }
  return min;
}
```

Problem 3. Binary Search

The file binary_search.dfy contains a method BinarySearch that performs binary search on a sorted array. The sorted-ness is guaranteed by the $Ordered(a)$ predicate, which is just a formula that asserts $a[i] <= a[j]$ for all $i <= j$. The post-condition is provided, which says that the method returns true if and only if the key x is contained in the array. Also, note that the while-loop is annotated with decreases $hi - lo$, which enables Dafny to prove that the loop terminates because the quantifier $(hi - lo)$ monotonically decreases in each iteration of the loop.

Your task is to replace invariant true in the while-loop of BinarySearch with an appropriate loop invariant that allows Dafny to verify the correctness of the method.

Hint:

> Your invariant needs to talk about where x cannot be found in the array based on the current values of lo and hi.

Attach your code and a brief explanation of your invariant in your submission.

Solution

We need two invariants here:

1. x should be strictly greater than all the elements in the subarray $a[0 : lo]$ and similarly
2. x should be strictly less than all the elements in the subarray $a[hi : len - 1]$

```
method BinarySearch(a: array<int>, x: int)
  returns (found: bool)
  requires Ordered(a)
  ensures
    // <==> means if-and-only if
    // so we're ensuring that
    // - Our search is sound: if we report true, then x must be in a
    // - Our search is complete: if x can be found, then we must
       ↪ report true
    found <==>
    exists i ::
      0 <= i < a.Length &&
      a[i] == x
{
  var len := a.Length;
  var lo := 0;
  var hi := len;
  while lo < hi
    decreases hi - lo
    invariant 0 <= lo <= hi <= len
    // ensure that x is strictly greater than all elements in array
       ↪ a[0:lo]
```

```
    invariant forall i : int :: 0 <= i < lo ==> a[i] < x

    //ensure that x is strictly less than all elements in array a[hi
       ↪ :len-1]
    invariant forall i : int :: hi <= i < len ==> a[i] > x
  {
    var mid := (lo + hi) / 2;
    if x == a[mid] {
      return true;
    } else if x < a[mid] {
      hi := mid;
    } else {
      lo := mid + 1;
    }
  }
  return false;
}
```

Problem 4. Buggy Binary Search

Immediately below BinarySearch, we duplicated its implementation into another method called BuggyBinarySearch. The only change we made is to make the index variables (lo, mid, hi, etc.) into 4-bit integers (unsigned), whereas in Dafny int is the default infinite-precision integer.

Copy and paste the (working) invariant you wrote for BinarySearch into BuggyBinarySearch and see if Dafny can verify the correctness of the method. Note that to make the invariant well-typed, you may need to do some conversion from int to bv4 using the syntax x as $bv4$ which converts a variable of type int to bv4.

After that, you shall see that Dafny cannot verify the correctness of BuggyBV4BinarySearch. Your tasks are to:

(a) locate the exact line where Dafny reports an error,

(b) explain why Dafny cannot verify the correctness of the method

(c) patch only one line of implementation code in the loop body to make verification succeed.

Include your answers to the above questions in your submission. You should also attach the modified code with the fix in your submission.

Hint

> The answers to the above questions can be found in the Wikipedia page for binary search, under the section Implementation Issues

Solution

Part (a)

Reported Error: Dafny cannot verify that the decreasing property is maintained in the line while lo < hi.

Part (b)

Underlying cause of the error is at the line calculating mid: var mid := (lo + hi) / 2;. This is because sum of two bv4 variables can overflow the range of bv4 even though both variables are within the range of bv4.

Part (c)

To circumvent this issue we can use the following formula: var mid := (hi - lo) / 2 + lo.

```
method BuggyBinarySearch(a: array<int>, x: int)
  returns (found: bool)
  requires Ordered(a)
  requires a.Length < 16
  ensures
    found <==>
    exists i : bv4 ::
    0 <= i < a.Length as bv4 &&
    a[i] == x
{
  var len := a.Length as bv4;
  var lo := 0;
  var hi := len;
  while lo < hi
    decreases hi - lo
    invariant 0 <= lo <= hi <= len
    // ensure that x is strictly greater than all elements in array
       ↪ a[0:lo]
    invariant forall i : bv4 :: 0 <= i < lo ==> a[i] < x
    //ensure that x is strictly less than all elements in array a[hi
       ↪ :len-1]
    invariant forall i : bv4 :: hi <= i < len ==> a[i] > x
  {
    var mid := (hi - lo) / 2 + lo;
    if x == a[mid] {
      return true;
    } else if x < a[mid] {
      hi := mid;
    } else {
      lo := mid + 1;
    }
  }
```

```
  return false;
}
```

Problem 5. Bubble Sort

Implement bubble sort in Dafny, and prove its correctness. There are multiple levels you can aim for in terms of the complexity of your implementation and the strength of your correctness proof:

Your bubble sort outputs an ordered list Your bubble sort outputs an ordered list whose elements form the same set as the input list Your bubble sort outputs an ordered list whose elements form the same multiset as the input list You may choose to aim for any of the above levels, but you should clearly state which level you are aiming for in your submission.

You may find this tutorial on verifying selection sort in Dafny helpful.

Attach your implementation, including all necessary annotations, in your submission. You should also informally explain in plain English the meaning of your pre- and post-conditions, and the loop invariants you used.

Solution

Part (a)

Post-Conditions:

1. Ensure that the output is ordered i.e. for every i, j belonging to $[0, a.Length]$, if i ¡= j then $a[i] <= a[j]$.
2. Ensure that the multiset of the output is the same as the input i.e. all elements of the array are preserved.

Part (b)

Outer Loop Invariants

1. Ensure that the current index is within range.
2. Ensure that the subarray from $a[i : a.Length]$ is ordered.
3. Ensure that the right side parition contains all elements greater than the left side. This is to ensure that the larger numbers are infact bubbling towards the end of the array.
4. Ensure that the array elements are preserved.

Part (c)

Inner Loop Invariants

1. Ensure that the subarray from $a[i : a.Length]$ is ordered.

2. Ensure that the right side parition contains all elements greater than the left side. This is to ensure that the larger numbers are infact bubbling towards the end of the array.

3. Ensure that the inner loop is bubbling the maximum element, i.e. the element at current inner loop index should be bigger than all the elements to it's left.

4. Ensure that the array elements are preserved.

```
ghost predicate Ordered(a: array<int>, start: int, end: int)
  reads a
  requires a.Length >= 1
{
  forall i: int, j: int :: 0 <= start <= i <= j <= end < a.Length
    ↪ ==> a[i] <= a[j]
}

ghost predicate bubble(a: array<int>, i: int)
  reads a
  requires a.Length >= 1
{
  forall j, k: int :: 0 <= j <= i < k < a.Length ==> a[j] <= a[k]
}

twostate predicate Preserved(a: array<int>)
  reads a
{
  multiset(a[..]) == multiset(old(a[..]))
}

method bubble_sort(a: array<int>)
  modifies a
  requires a.Length >= 1
  ensures Ordered(a, 0, a.Length-1)
  ensures Preserved(a)
{
  var i : int := a.Length - 1;
  while i > 0
    // write invariants here
    // 1. invariant that i is increasing to prove termination
    invariant -1 <= i < a.Length
    // 2. invariant to prove that the subarray a[i:a.Length-1] is
      ↪ ordered
    invariant Ordered(a, i, a.Length-1)
    // 3. invariant to prove the all numbers in right side partition
      ↪  are bigger than numbers in left side
    invariant bubble(a, i)

    // add invariant that the array is always preserved
```

```
    invariant Preserved(a)

  {

    var j : int := 0;
    while j < i
      // write invariants here
      // 1. invariant that j to prove termination
      invariant 0 < i < a.Length && 0 <= j <= i

      // invariant to make sure that a[j-1] < a[j]
      invariant Ordered(a, i, a.Length-1)
      invariant bubble(a, i)
      invariant forall k :: 0 <= k < j ==> a[k] <= a[j]
      invariant Preserved(a)
    {
      if a[j] > a[j+1] {
        // swap the two numbers
        a[j], a[j+1] := a[j+1], a[j];
      }
      j := j + 1;
    }
    i := i - 1;
  }
}
```