CSCI 2600 — Principles of Software

Homework 2 - Jimmy Chen

Directions:

This is a template for students' answers to Homework 2. Feel free to modify as much as you want!

Problems

Problem 1 (8 pts.): Sum of natural numbers

Below we give, in Java syntax, the multiplication method which should be computing the product of two numbers.

```
// Precondition: n >= 0
int sumn(int n) {
    int i = 0, t = 0;
    while (i < n) {
        i = i + 1;
        t = t + i;
    }
    return t;
}
// Postcondition: t = n * (n + 1) / 2
```

a) Find a suitable loop invariant. (1 pt.)
Answer:
Loop Invariant: t = i * (i + 1) / 2

b) Show that the invariant holds before the loop (base case). (1 pt.)
Answer:
Base Case: i == 0 and t == 0
(t == i * (i + 1) / 2) = (t == 0 * (0 + 1) / 2) = (0 == 0 * (0 + 1) / 2)
(i $\leq$ n) = (0 $\leq$ n) // precondition

c) Show by induction that if the invariant holds after the k-th iteration, and execution takes a k+1st iteration, the invariant still holds (inductive step). (2 pts.) Induction: Assume the invariant holds for iteration k. We will show that it holds for iteration $k + 1$.
Answer:
assume t_old = i_old * (i_old + 1) / 2
i_new = i_old + 1

// assume t_old = i_old * (i_old + 1) / 2
t_new = t_old + i_new
t_new = (i_new-1 * (i_new-1 + 1) / 2) + i_new
t_new = i_new * (i_new + 1) / 2

i_old $\leq$ n; if i_old == n, we would have exited
i_old < n
i_new = i_old + 1 $\leq$ n

d) Show that the loop exit condition and the loop invariant imply the postcondition $t = n{*}(n{+}1)/2$.

(1 pt.)
Answer:
Postcondition Implication:
At Exit: !(i<n) && (i $\leq$ n && t == (i * (i + 1) / 2))
=> i == n && t == (i * (i + 1) / 2))
=> t == n * (n + 1) / 2

e) Find a suitable decrementing function. Show that the function is non-negative before loop starts, that it decreases at each iteration and that when it reaches 0 the loop is exited. (1 pts.)
Answer:
Decrementing Function:
D = n - i // initially it's i == 0 and from precondition n $\leq$ 0 so D $\leq$ 0.

Decreases at each iteration:
D = n - i
D_old = n_old -i_old
i_new = i_old + 1
D_new = n - i_new
D_new = n - (i_old + 1)
D_new = (n - i_old) - 1
D_new = D_old - 1
This shows that it is decrementing at each iteration

D = 0:
D == 0 => i == n which is the loop exit condition
f) Implement the sum of natural numbers in Dafny. (2 pts., autograded)

Problem 2 (15 pts.): Loopy square root

Below we give, in Dafny syntax, the square root method which should be computing the square root of a number.

```
method loopysqrt(n:int) returns (root:int)
    requires n >= 0
    ensures root * root == n
{
    root := 0;
    var a := n;
    while (a > 0)
        //decreases //FILL IN DECREMENTING FUNCTION HERE
        //invariant //FILL IN INVARIANT HERE
        {
            root := root + 1;
            a := a - (2 * root - 1);
        }
}
```

a) Test this code by creating the `Main()` method and calling `loopysqrt()` with arguments like 4, 25, 49, etc. to convince yourself that this algorithm appears to be working correctly. In your answer, describe your tests and the corresponding output. (1 pt.)
Tests and Output:
Answer:

```
method Main() {
    var n := loopysqrt(4);
    print n, "\n"; // 2
    var m := loopysqrt(25);
    print m, "\n"; // 5
    var o := loopysqrt(49);
    print o, "\n"; // 7
}
```

Above is the Main I created to test the output. I used 4, 25, and 49, and it printed 2, 5, and 7 respectively.

b) Yet, the code given above fails to verify with Dafny. One of the reasons for this is that it actually does have a bug. More specifically, this code may produce the result which does not comply with the specification. Write a test (or tests) that reveals the bug. In your answer, describe your test(s), the corresponding outputs, and the bug that you found. Also, indicate which part of the specification is violated. (1 pt.)
Tests and Outputs:
Answer:

```
    method Main() {
    var p := loopysqrt(10);
    print p, "\n"; // 4
    var q := loopysqrt(15);
    print q, "\n"; // 4
    var r := loopysqrt(17);
    print r, "\n"; // 5
}
```

Bug Found:
For non-perfect squares, the function would always return the next square root, i.e., if 10 was passed in, it would return 4, and same for 15.

Violation of Specification:
The code didn't pass the postcondition as root * root was greater than n when it was not a perfect square.

c) Now find and fix this bug. Note that there might be several different ways of fixing the bug. Use the method that you think would be the best. You are not allowed to change the header of loopysqrt() or add, remove, or change any specifications or annotations. Do not worry about Dafny verifying the code for now, just fix the bug and convince yourself that loopysqrt() is now correct. You are also required to keep the overall algorithm the same as in the original version of the code. In your answer, describe how you fixed the bug and show the output of the same tests you ran before after fixing the bug. (2 pts.)
Tests and Outputs:

```
method Main() {
    var p := loopysqrt(10);
    print p, "\n"; // 3
    var q := loopysqrt(15);
    print q, "\n"; // 3
    var r := loopysqrt(17);
    print r, "\n"; // 4
}
```

Bug Fix:
I fixed the bug by adding a conditional statement inside the loop, to check if a becomes negative. If it is negative, it means that we encountered a non-perfect square, so we should break out of the loop so that it doesn't return a root larger than the sqrt of n. I.e. 15 would return 3 instead of 4, as 4*4 is 16, which is larger.

Code Additions:

```
1. (inside loop)
    if (0 > a - (2 * (root + 1) - 1)) {
        break;
    }
```

d) Update the specification of loopysqrt() to match the way you fixed the bug. If you changed the postcondition, make sure that it is the strongest possible postcondition. In your answer, describe your changes and explain why they were necessary. (1 pt.)
Changes:
I changed the post-condition to root*root <= n. It was necessary because this allows a valid answer when n is not a perfect square. When it was root*root == n, it was too strong, as it only allowed for perfect squares to complete.

e) Does your Dafny code verify now? Why or why not? If it doesn't verify, does it mean that your code still has bugs in it? (1 pt.)
Answer:
My dafny code still doesn't verify because you need to explicitly describe the loop invariant to ensure that the loop terminates. This doesn't mean that my code still has bugs.

f) f your Dafny code doesn't verify, uncomment `invariant` and/or `decreases` annotations and supply the actual invariant and/or decrementing function. Make sure your code now verifies. In your answer, describe how you guessed the invariant and/or decrementing function. Explain why your code was failing Dafny verification earlier but does verify now, despite the fact that you have not made any changes to your actual code (annotations are not part of the code). (1 pt.)
Answer:

```
invariant n == root * root + a && a >= 0
decreases a
```

I guessed my decrementing function, as a, because a is always the one being subtracted from at the end of each iteration. And for the invariant, I made sure that it fell within the pre and post-condition, and that it stayed true for each iteration. The code now passes the verification because adding the loop invariant allows the program to verify that the postcondition is met at each iteration.

g) Finally, remove the precondition from your Dafny code and make necessary changes to the remaining annotations and/or code ensure that code still verifies. You are not allowed to change the header of loopysqrt(). You are also required to keep the overall algorithm the same as in the previous versions of the code. As before, make sure that your postcondition is the strongest. Did removing the precondition make your code more difficult? What effect did removing the precondition have on the client of loopysqrt()? (2 pts.)
Answer:
Removing the precondition made it harder to figure out how to fix the code. The precondition being removed cause the post condition to not work anymore, as it doesn't account for n having to be greater than 0, as you can't have a negative root. To fix the verification, i added n ¡ 0 to the post-condition, and a if statement that catches negatives, before the loop starts.

Code Addition:

```
method loopysqrt(n:int) returns (root:int)
    ensures root * root <= n || n < 0
{
    if (n < 0) {
        return 0;
    }
...
...
```

h) Use computational induction to prove by hand the total correctness of the final version of your Dafny code. (6 pts.)
Base case:
n == 0 && a == n
n = 0 * 0 + a, so n == a.
a >= 0, proving the base case

Induction: Assume the invariant holds for iteration k. We will show that it holds for iteration $k+1$.
Assume: n == root * root + a
root_new = root_old + 1

Assume: n == root - 1 * root - 1 + a_old
a_new = a_old - ((root + 1) * (root + 1) - 1)
a_new = a_old - (2 * root + 1)
n == root * root + a_new // showing that iteration k+1 holds

Postcondition Implication:
At Exit: !(a>0) && n == root * root + a && a >= 0
Since a is ≤ 0, we can say that a == 0 at the end. n == root * root + 0, which is n == root * root, which falls within the post condition.

Decreases at each iteration:
D = a // decreasing at each iteration, from precondition a >= 0, so D >= 0
D = 0:
D == 0 ==> a == 0 (loop exit condition)

Problem 3 (12 pts.): Array of differences

Below is the pseudocode for creating an array containing elements each of which is the difference between two adjacent elements of the input array:

```
Precondition: arr != null ∧ arr.Length > 0
int[] difference(int[] arr) {
   diffs ← new int[arr.Length - 1]
   a ← 0
   while (a < diffs.Length)
   {
      diffs[a] ← arr[a + 1] - arr[a]
      a ← a + 1
   }
   return diffs
}
```

`Postcondition: diffs.Length = arr.Length - 1` ∧ forall k :: 0 <= k < diffs.Length ==> diffs[k] == arr[k + 1] - arr[k]

a) Find a suitable loop invariant. (2 pts.)
Invariant:
Answer: `Postcondition: diffs.Length = arr.Length - 1` $\wedge \forall$ `, s.t.` $0 \leq$ `a < diffs.Length, diffs[a] = arr[a + 1] - arr[a]`

b) Show that the invariant holds before the loop (base case). (1 pt.)
Base Case: a == 0
Since a is set to be 0, it falls in 0 <= a <= diffs.Length. When a == 0, theres no valid number between 0 & 0 so the invariant holds.

c) Show by induction that if the invariant holds after k-th iteration, and execution takes a k+1-st iteration, the invariant still holds (inductive step). (4 pts.)
Induction: Assume the invariant holds for iteration k. We will show that it holds for iteration $k+1$.
Assume: 0 <= a <= diffs.Length
0 <= a + 1 <= diffs.Length // this is true because the exit condition will compile before a is greater than diffs.Length

Assume: forall k :: 0 <= k < a ==> diffs[k] == arr[k + 1] - arr[k]
forall k :: 0 <= k < a + 1 ==> diffs[k] == arr[k + 1] - arr[k] // showing that the property holds for a+1
When a is incremented by 1, it becomes a + 1. Therefore, for the case of k = a, we have: 0 <= a < a + 1

This inequality is true as long as a is a non-negative integer. So, the condition 0 <= k < a is satisfied for k = a.

Substituting k for a: diffs[a] == arr[a + 1] - arr[a]
This is the same as our assignment statement within the loop (diffs[a] := arr[a + 1] - arr[a];), so the property holds for k = a. And so if the invariant holds for the k-th iteration, it implies that the property holds for the k+1th iteration.

d) Show that the loop exit condition and the loop invariant imply the postcondition diffs.Length $=$ arr.Length $- 1 \wedge \forall k, s.t. 0 \leq k <$ diffs.Length, diffs[k] $=$ arr[k + 1] $-$ arr[k].$(1pt.)$
Postcondition Implication:
The loop exits when a is no longer less than diffs.Length, which means the loop iterates until a reaches the last index of diffs. This implies that a will iterate over all valid indices of diffs, which corresponds to 0 to diffs.Length - 1.

diffs.Length == arr.Length - 1: This is satisfied because the loop iterates diffs.

Length times (a ranges from 0 to diffs.Length - 1), resulting in diffs.Length elements being computed.

forall k :: 0 <= k < diffs.Length ==> diffs[k] == arr[k + 1] - arr[k]: This is satisfied because of the loop invariant, which ensures that each element of diffs is the difference between consecutive elements of arr.

e) Find a suitable decrementing function. Show that the function is not negative before loop starts, that it decreases at each iteration and that when it reaches 0 the loop is exited. (2 pts.)
Decrementing Function:
D = diffs.Length - a Not negative before the loop starts: Initially, a is set to 0, and diffs.Length is at least 1 (since arr.Length is required to be greater than 0), so D = diffs.Length - a is non-negative.

Decreases at each iteration:

D_old = diffs.Length - a_old
After one iteration, a_new = a_old + 1
D_new = diffs.Length - a_new
= diffs.Length - (a_old + 1)
= (diffs.Length - a_old) - 1
= D_old - 1
Thus, D_new = D_old - 1, showing that it decreases by 1 at each iteration.

D = 0:

When the loop exits, D = 0. This implies that diffs.Length - a = 0, which happens when a equals diffs.Length. f) Implement the array of differences in Dafny. (2 pts., autograded)

g) Extra credit (3 pts.) Implement difference in Dafny using sequences instead of arrays.

Problem 4 (15 pts.) The Simplified Dutch National Flag Problem

a) Given an array arr[0..N-1] where each of the elements can be classified as red or blue, write pseudocode to rearrange the elements of arr so that all occurrences of blue come after all occurrences of red and the variable k indicates the boundary between the regions. That is, all arr[0..k-1] elements will be red and elements arr[k..N-1] will be blue. You might need to define method swap(arr, i, j) which swaps the ith and jth elements of arr. (7 pts.)

The following picture illustrates the condition of the array at exit.

```
         0                    k                 N
        +--------------------+-----------------+
    arr |   all red          | all blue        |
        +--------------------+-----------------+
```

Psuedocode:

```
// Precondition: arr != null && arr.Length > 0
    && 0 <= i < arr.Length && 0 <= j < arr.Length
function swap (arr char, int i, int j)
    temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp

// Precondition: arr != null
int dutch(arr char) return (k int)
    i = 0
    j = length(arr)
    while i < j:
        if arr[i] == 'r'
            i += 1
        else:
            arr[i] = arr[j-1]
            arr[j-1] = 'b'
            j -= 1
    k = i;
// Postcondition: 0 <= k <= arr.Length
    && forall i :: 0 <= i < k ==> arr[i] == 'r'
    && forall i :: k <= i < arr.Length ==> arr[i] == 'b'
```

b) Write an expression for the postcondition. (2 pts.)
Postcondition:

```
ensures 0 <= k <= arr.Length
ensures forall i :: 0 <= i < k ==> arr[i] == 'r'
ensures forall i :: k <= i < arr.Length ==> arr[i] == 'b'
```

c) Write a suitable loop invariant for all loops in your pseudocode. (4 pts.)
Loop Invariant:

```
invariant 0 <= i <= arr.Length
invariant 0 <= j <= arr.Length
invariant forall m :: 0 <= m < i ==> arr[m] == 'r'
invariant forall m :: j <= m < arr.Length ==> arr[m] == 'b'
```

d) Implement your pseudocode in Dafny. (2 pts., autograded)