SLAps - Week 2

Vedant Rana

September 2025

Contents

1 Loop Invariants Summary

The term invariant means '*something that does not change*' or '*stays the same*'. Thus, intuitively, a loop invariant is a statement that remains true before *and* after every iteration of a loop.

Formally, a loop invariant is a statement I such that I is true

1. Before the loop has begun (initialization).
2. Before and after every iteration of the loop (preservation).
3. I and the negation of the loop guard imply the post condition (termination).

All three of these properties (initialization, preservation, termination) must be satisfied for a statement I to be a loop invariant.

Correct loop invariants allow us to prove the *partial* correctness of a program. That is,

If the program terminates, then the post-condition holds.

To show total completeness, one would have to prove that the program does indeed terminate. A proof of termination, along with a correct loop invariant, proves total correctness. That is,

The program terminates, and the postcondition holds.

2 Problem 1

Loop invariants for problem 1 is given in lines 12 and 13.

```
1  // Problem 1: Simple Loop with Two Variables
2  // Write loop invariant(s) for this method
3
4  method loop(n: int) returns (j: int)
5      requires n >= 0
6      ensures j == 2 * n
7  {
8      var i := 0;
9      j := 0;
10
11     while i < n
12         invariant j == 2*i
13         invariant 0 <= i <= n
14         decreases n - i
15     {
16         i := i + 1;
17         j := j + 2;
18     }
19 }
```

2.1 Justification

We prove that

$$I : j = 2 * i$$
$$0 \leq i \leq n$$

is the correct loop invariant for problem 1. We do so by proving

$$\begin{aligned} &\text{Initialization} : P \Rightarrow I \\ &\text{Preservation} : \{I \wedge B\} \quad S \quad \{I\} \\ &\text{Termination} : I \wedge \neg B \Rightarrow Q \end{aligned}$$

where

Proof. **Initialization:** Before the loop starts, $i = 0$ and $j = 0$. So $j = 0 = 2i$. By the `requires` clause, $n \geq 0$. So $0 \leq i \leq 0 = n$. Thus

$$P \ \Rightarrow \ I.$$

□

Proof. **Preservation:**
P(k): We show that for every iteration $k \geq 1$ of the loop, I holds before the iteration and after. We proceed via induction.

Base Case: Let $k = 1$. Before the first iteration, $i = 0$ and $j = 0$, so $j = 0 = 2 \cdot i$ and $0 \leq i \leq 0 = n$.

After the first iteration:

$$i = 1, \quad j = 2.$$

So clearly $j = 2 \cdot i$. Since the loop ran for 1 iteration, the guard $i < n$ must have been true, hence

$$0 \leq i \leq n.$$

Thus I is true before and after the first iteration.
Induction Hypothesis (IH): Let $k \geq 1$. Assume $P(k)$ holds. That is,

$$j_B = 2 \cdot i_B \quad \text{and} \quad j_A = 2 \cdot i_A$$

where j_B and i_B represent the values of j and i before the k^{th} iteration respectively (same with j_A and i_A).

Inductive Step: We show that $P(k+1)$ holds. By the induction hypothesis, the loop invariant holds before the $(k+1)$-th iteration. After the $(k+1)$-th iteration,

$$i = i_A + 1, \qquad j = j_A + 2.$$

Now,

$$2 \cdot i = 2 \cdot (i_A + 1) = 2i_A + 2 = j_A + 2 = j.$$

So $P(k+1)$ is true. Hence by induction, $P(k)$ holds for all $k \geq 1$.

□

Proof. **Termination:** Assume $j = 2i$ and $\neg(i < n)$. The loop terminates if $i = n$. So we have

$$j = 2i \quad \text{and} \quad i = n.$$

Therefore,

$$j = 2n.$$

Thus, the post-condition is true. □

By proving Initialization, Preservation, and Termination, we conclude that I is indeed the correct loop invariant for problem 1.

3 Problem 2

Loop invariants for this problem are given in lines 14 and 15.

```
1  // Problem 2: Integer Division (Quotient & Remainder)
2  // Write loop invariant(s) for this method
3
4  method IntegerDivision(dividend: int, divisor: int) returns (
       quotient: int, remainder: int)
5      requires divisor > 0
6      requires dividend > 0
7      ensures dividend == divisor * quotient + remainder
8      ensures 0 <= remainder < divisor
9  {
10     quotient := 0;
11     remainder := dividend;
12
13     while remainder >= divisor
14         invariant dividend == divisor * quotient + remainder
15        invariant remainder>=0
16         decreases remainder
17     {
18         quotient := quotient + 1;
19         remainder := remainder - divisor;
20     }
21 }
```

I modified the code of problem 2 by adding the following requires clause (read line 6) -

```
1      requires dividend > 0
```

Without this requires clause, one may run IntegerDivision(-12,5) (Appendix), and the output would be

$$\text{quotient} = 0$$
$$\text{remainder} = -12$$

which contradicts the

$$0 \leq \text{remainder} < \text{divisor}$$

ensures clause in line 8.

3.1 Justification

We prove that

$$I: \quad \text{dividend} = \text{divisor} \times \text{quotient} + \text{remainder}, \quad \text{remainder} \geq 0$$

is the correct loop invariant for Problem 2.

Claim: Initialization: $P \implies I$.

Proof. **Initialization:** The preconditions are:

$$\text{quotient} = 0 \quad \text{and} \quad \text{remainder} \leq \text{dividend}.$$

Now

$$\begin{aligned} &\text{divisor} \times \text{quotient} + \text{remainder} \\ &= \text{divisor} \times 0 + \text{remainder} \\ &= \text{remainder} \\ &= \text{dividend}. \end{aligned}$$

Thus,

$$\text{dividend} = \text{divisor} \times \text{quotient} + \text{remainder}.$$

By the requires clause

$$\text{remainder} = \text{dividend} > 0.$$

Thus, I holds. □

Claim: Preservation: $\{I \wedge B\} \quad S \quad \{I\}$.

Proof. **Preservation:**

$P(k)$: For iterations $k \geq 1$ of the loop, I is true before and after iteration k.

Base Case: $k = 1$
Before iteration 1, the proof of $P \Rightarrow I$ shows that I holds.
After iteration 1:

$$\begin{aligned} q_{A_1} &= q_{B_1} + 1 \\ r_{A_1} &= r_{B_1} - \text{divisor} \end{aligned}$$

where the subscripts A_i and B_i denote after iteration i and before iteration i, respectively.
Now,

$$\begin{aligned} \text{dividend} &= \text{divisor} \cdot q_{A_1} + r_{A_1} \\ &= \text{divisor} \cdot (q_{B_1} + 1) + r_{B_1}\, \text{divisor} \\ &= \text{divisor} \cdot q_{B_1} + r_{B_1} \\ &= \text{dividend} \end{aligned}$$

We've already shown this is true.
We now need to show that $r_{A_1} \geq 0$. The loop only runs if the guard condition remainder $\geq$ divisor is true. Thus, iteration 1 is executed if $r_{B_1} \geq$ divisor.

Hence, $r_{A_1} = r_{B_1} - \text{divisor} \geq 0$. Thus, I holds after iteration 1 as well.

Inductive Hypothesis (IH): Assume $P(k)$ holds for $k \geq 1$. That is,

$$\text{(Before iteration } k\text{):} \quad \text{dividend} = \text{divisor} \cdot q_{B_k} + r_{B_k}, \quad r_{B_k} \geq 0.$$

$$\text{(After iteration } k\text{):} \quad \text{dividend} = \text{divisor} \cdot q_{A_k} + r_{A_k}, \quad r_{A_k} \geq 0.$$

We show that $P(k+1)$ holds, that is,

$$\text{(Before iteration } k+1\text{):} \quad \text{dividend} = \text{divisor} \cdot q_{B_{k+1}} + r_{B_{k+1}}, \quad r_{B_{k+1}} \geq 0.$$

$$\text{(After iteration } k+1\text{):} \quad \text{dividend} = \text{divisor} \cdot q_{A_{k+1}} + r_{A_{k+1}}, \quad r_{A_{k+1}} \geq 0.$$

Before iteration $k+1$:

$$\begin{aligned} \text{divisor} \cdot q_{B_{k+1}} + r_{B_{k+1}} &= \text{divisor} \cdot q_{A_k} + r_{A_k} \\ &= \text{dividend} \\ r_{B_{k+1}} &\geq 0 \quad \text{.(By the IH)} \end{aligned}$$

After iteration $k+1$:

$$\begin{aligned} \text{divisor} \cdot q_{A_{k+1}} + r_{A_{k+1}} &= \text{divisor} \cdot (q_{B_{k+1}} + 1) + (r_{B_{k+1}} - \text{divisor}) \\ &= \text{divisor} \cdot q_{B_{k+1}} + r_{B_{k+1}} \\ &= \text{divisor} \cdot q_{A_{k+1}} + r_{A_{k+1}} = \text{dividend}. \quad \text{(By the IH)} \end{aligned}$$

By the guard condition of the loop, if the $k+1$ iteration was executed, then it must be because $r_{B_{k+1}} \geq \text{divisor}$.

Thus,

$$\begin{aligned} r_{A_{k+1}} &= r_{B_{k+1}} - \text{divisor} \geq 0 \\ r_{A_{k+1}} &\geq 0 \end{aligned}$$

Hence, by induction, $P(k)$ holds for all $k \geq 1$.

□

Claim: Termination

$$I \wedge \neg B \implies Q.$$

Proof:
Assume I holds and the loop terminates. That is,

$$\begin{aligned} \text{dividend} = \text{divisor} \times q + r \quad &\wedge \\ r \geq 0 \quad \wedge \quad r < \text{divisor}. \end{aligned}$$

Clearly,

$$0 \leq r < \text{divisor},$$

and

$$\text{dividend} = \text{divisor} \times q + r.$$

Thus, Q (the post conditions) holds. □

Since Initialization, Preservation, and Termination are true, I is indeed the correct loop invariant for problem 2.

4 Problem 3

Loop invariants for the GCD program are given in lines 14 and 15 -

```
1     // Problem 3: GCD Calculation
2  // Write loop invariant(s) for this method
3
4  method GCD(a: int, b: int) returns (gcd: int)
5      requires a > 0 && b > 0
6      ensures gcd > 0
7      ensures a % gcd == 0 && b % gcd == 0
8      ensures forall d :: d > 0 && a % d == 0 && b % d == 0 ==> d <=
       gcd
9  {
10     var x := a;
11     var y := b;
12
13     while y != 0
14         invariant x >= 0 && y >= 0
15         invariant forall d :: d>0 && x % d == 0 && y % d == 0 <==>
       a % d == 0 && b % d == 0
16         decreases y
17     {
18         var temp := y;
19         y := x % y;
20         x := temp;
21     }
22
23     gcd := x;
24 }
```

The problem with this loop invariant is that Dafny parses it in such a fashion that I get a division by zero error. The following invariant get rid of this error -

```
    // Problem 3: GCD Calculation
// Write loop invariant(s) for this method

method GCD(a: int, b: int) returns (gcd: int)
    requires a > 0 && b > 0
    ensures gcd > 0
    ensures a % gcd == 0 && b % gcd == 0
    ensures forall d :: d > 0 && a % d == 0 && b % d == 0 ==> d <=
    gcd
{
    var x := a;
    var y := b;

    while y != 0
        invariant x >= 0 && y >= 0
        invariant forall d:: d > 0 ==> (x % d == 0 && y % d == 0
    <==> a % d == 0 && b % d == 0)
        decreases y
    {
        var temp := y;
        y := x % y;
        x := temp;
    }

    gcd := x;
}
```

I wrote the proof for the first invariant, and although I know the invariant was incorrect, here's my proof -

4.1 Justification

I've given a proof in a pdf called Problem3-Proof.pdf.

5 Problem 4

The code is

```
    // Problem 4: Fast Power (Exponentiation by Squaring)
// Write loop invariant(s) for this method

method FastPower(base: int, exp: int) returns (result: int)
    requires exp >= 0
    ensures result == Power(base, exp)
{
    var x := base;
    var n := exp;
    result := 1;

    while n > 0
    invariant n >= 0
    invariant result * Power(x, n) == Power(base, exp)
        // TODO: Write loop invariant(s)
        decreases n
    {
        if n % 2 == 1 {
            result := result * x;
        }
        x := x * x;
        n := n / 2;
    }
}

// Helper function
function Power(base: int, exp: int): int
    requires exp >= 0
{
    if exp == 0 then 1 else base * Power(base, exp - 1)
}
```

5.1 Justification

The invariant is

```
    invariant n >= 0
    invariant result * Power(x, n) == Power(base, exp)
```

The program is trying to compute base raised to some exponent exp. result represents the product already computed (it's like an accumulator), and Power(x,n) represents the remaining computation we have yet to do. This is why, before and after any iteration

```
    result * Power(x, n) == Power(base, exp)
```

is holds. So the initialization and preservation conditions hold.

When the loop terminates (n=0) and the invariant is true, we get -

$$\begin{aligned} \text{result} * \text{Power}(x, n) &== \text{Power(base, exp)} \\ \text{result} * 1 &= \text{Power(base, exp)} \\ \text{result} &= \text{Power(base, exp)}. \end{aligned}$$

Thus termination holds as well. So this is the correct loop invariant.

6 Problem 5

I could not solve this problem, here's all I could figure out -

At any iteration of the algorithm, the ith digit of rev should be the (length of n+1 - i)th digit of n. For example, say n = 3054. After the 2nd iteration of the loop, the 2nd digit of rev should be 5, which is precisely the

length of (3054) + 1 - 2 = 4+1-2 = 3rd digit of n.

So the loop invariant is -
For any iteration i of the loop, the ith digit of rev should be

ith digit of rev = length(n) + 1 - i.

7 Appendix

```
    method IntegerDivision(dividend: int, divisor: int) returns (
    quotient: int, remainder: int)
    requires divisor > 0
    ensures dividend == divisor * quotient + remainder
    ensures 0 <= remainder < divisor
{
    quotient := 0;
    remainder := dividend;

    while remainder >= divisor
        // TODO: Write loop invariant(s)
        decreases remainder
    {
        quotient := quotient + 1;
        remainder := remainder - divisor;
    }
}

method Main() {
    var q, r := IntegerDivision(-12,5);
    print "quotient = ", q, ", remainder = ", r, "\n";
}
```

The output of IntegerDivision(-12,5) is

```
    (base) vedantrana@Richas-MacBook-Air Week_2 % dafny run --no-
    verify problem2.dfy

Dafny program verifier did not attempt verification
quotient = 0, remainder = -12
(base) vedantrana@Richas-MacBook-Air Week_2 %
```

This clearly contradicts the ensures clause that $0 \leq \text{remainder} \leq \text{divisor}$.