CS5310: Symbolic Logic and Applications

Project Report: Week 2

September 7, 2025

1 Week 2 Report

Progress on mentioned deliverables:

1. Provide a summary/explanation of loop invariants provided in class:

A loop invariant is a logical property of a program that remains true before and after each execution of a loop. Loop invariants are useful for reasoning about the correctness of programs because they provide a formal way to connect the initial state of the loop to its final outcome. To prove that a loop satisfies its specification, we typically follow three steps: initialization, maintenance, and termination.

Initialization means showing that the invariant holds before the first iteration of the loop. Maintenance requires showing that if the invariant is true before a given iteration, it remains true after the iteration finishes. Termination then uses the fact that the loop condition is false when the loop exits, together with the invariant, to prove that the desired postcondition holds.

Loop invariant:

```
method loop(n: int) returns (j: int)
{
    var i := 0;
    j := 0;
    while i < n
        invariant 0 <= i <= n
        invariant j == 2 * i
    {
        i := i + 1;
        j := j + 2;
    }
}
```

At the beginning, $i = 0$ and $j = 0$, so the invariants $0 \leq i \leq n$ and $j = 2i$ both hold. During each iteration, i is incremented by one and j is incremented by two. Assuming that $j = 2i$ before an iteration, after the update we obtain

$j' = j+2 = 2i+2 = 2(i+1)$, which shows that the invariant is preserved. When the loop terminates, we have $i = n$, and by the invariant it follows that $j = 2n$. This matches the postcondition of the method and proves its correctness.

2. Write loop invariants by hand for 5 given problems. **COMPLETED. Output is in the github repository.**

Problem 1: The loop invariants are as follows:

i. $0 \leq i \leq n$: This invariant guarantees that the loop variable i stays within valid bounds. It begins at 0 and increases by one in each iteration, never exceeding n because of the loop condition.

ii. $0 \leq j \leq 2n$: Since j starts at 0 and increases by 2 in every iteration, this invariant ensures that j remains within the expected range relative to n.

iii. $j = 2i$: Holds at initialization when i = 0 and j = 0. If $j = 2i$ before an iteration, then after the updates $i' = i + 1$ and $j' = j+2 = 2i+2 = 2(i+1) = 2i'$, so the invariant continues to hold.

The variant $n - i$ decreases with every iteration, ensuring that the loop eventually terminates.

When the loop exits, the condition $i < n$ is false, so $i = n$. At this point, the invariant $j = 2i$ gives $j = 2n$, which satisfies the postcondition of the method. This confirms the correctness of the program.

Problem 2: The loop invariants are as follows:

i. remainder = dividend − divisor × quotient

ii. quotient ≥ 0: Since the quotient starts at zero and only increases by one in each iteration, it is guaranteed to remain nonnegative throughout execution.

iii. dividend $\geq$ remainder ≥ 0: Because the remainder begins equal to the dividend and decreases by the divisor each iteration, it always stays within the range 0 to dividend.

The variant expression remainder decreases with every iteration by at least one, ensuring that the loop eventually terminates.

When the loop exits, we have remainder $<$ divisor by the negation of the loop condition. Combined with the invariant remainder = dividend − divisor × quotient, this gives the required postcondition that dividend = divisor × quotient + remainder with $0 \leq$ remainder $<$ divisor.

Problem 3: The loop invariants are as follows:

i. $x > 0$ ensures that the variable x remains positive throughout the iterations, which is necessary since the GCD is always positive.

ii. $y \geq 0$ guarantees that y never becomes negative, which maintains the correctness of the modulus operation.

iii. $\forall d > 0, (d \mid a \wedge d \mid b) \Rightarrow (d \mid x \wedge d \mid y)$ maintains that any common divisor of the original inputs a and b is also a divisor of the current loop variables x and y. This invariant ensures that the algorithm preserves all common divisors and therefore converges to the greatest one.

Upon termination, when $y = 0$, the loop invariant implies that x is divisible by all common divisors of a and b. Assigning gcd := x satisfies the postconditions, and the code correctly computes the greatest common divisor.

Problem 4: The loop invariants are as follows:

i. n >= 0 ensures that the exponent variable remains non-negative throughout the loop, which is necessary because the Power function is defined for non-negative exponents only.

ii. Power(base, exp) == result * Power(x, n) guarantees that, at any point in the loop, the product of result and x^n equals the original $base^{exp}$.

The loop updates the variables differently depending on whether the exponent n is odd or even. If n is odd, the code multiplies the result by x and decreases n by 1. This keeps the invariant true because $x^n = x \cdot x^{n-1}$. If n is even, it squares x and halves n, which also preserves the invariant since $x^n = (x^2)^{n/2}$.

When the loop ends and n equals 0, the invariant shows that the result equals the original $base^{exp}$.

Problem 5: The loop invariants used in this algorithm are:

i. num ≥ 0 for non-negativity of the tracking variable.

ii. rev * Power(10, NumDigits(num)) + ReverseDigits(num) = ReverseDigits(n)

At each iteration, the last digit of num is appended to rev, and num is reduced by removing its last digit. The invariant guarantees that, no matter how many digits have been processed, the sum of the reversed portion and the remaining portion (adjusted by powers of 10) equals the total reversed number.

When the loop terminates (num = 0), the invariant simplifies to rev = ReverseDigits(n), ensuring that the code correctly computes the reversed number.

3. Start reading CSUR14 survey paper ("Loop invariants: analysis, classification, and examples"). **Started ✓**