Loop Invariants

I, Gries, have 56 years of programming experience.
You have perhaps 1. I don't think your knowledge of programming is enough for you to make a decision like this.

— *David Gries, on allegations of discussing loops in a cumbersome and unintuitive fashion.* · *Feb. 22, 2017*

Purpose

- To ensure your code will always perform as expected
- Help you develop good habits
- Makes it easier to debug code
- Prove your code correct while you write it!

Purpose

Binary Search

```
/**
 * Returns: an index i such that a[i] == k.
 * Requires: k is in a, and a is sorted in ascending order
 */
int search(int[] a, int k) {
    int l = 0, r = a.length-1;
    while (l < r) {
        int m = (l+r)/2;
        if (k <= a[m]) r = m;
        else l = m+1;
    }
    return l;
}
```

- Why is the loop guard `l < r` instead of `l <= r`?
- Why is it `k <= a[m]` and not `k < a[m]`?
- Where do `m` and `m+1` come from?
- If any of these were different, the code would be wrong!

Preliminaries

- **Notation:**
  - For integers i, j and an array b, we write b[i..j] to mean b[i], b[i+1], …, b[j], i.e. all of those elements of b starting at index i up to *and including* index j
  - i = j: b[i..i] = b[i]
  - i > j: b[i..j] is empty
- **Precondition:** Describes the possible states that the program may occupy *before* execution of the loop
- **Postcondition**: Describes the possible states that the program may occupy *after* execution of the loop
- **Loop invariant:** Describes the state of the program *just before* each iteration of the loop (the loop invariant may be broken in the body of the loop, as long as it is restored before the body terminates)

Loop Invariant Steps

For Partial Correctness

- Establishment
- Preservation
- Postcondition

For Total Correctness

- Partial Correctness
- Termination

Establishment (Initialization)

Must show that loop invariant is true right after variables are initialized.

Examples: List is sorted before loop, list.length > 0, etc

Preservation (Maintenance)

```
1
2 // loop invariant P
3 while (b) {
4     // do stuff assuming (P and b)
5     // ensure that P remains true after this iteration is over
6 }
7
8 // now we know (P and not b)
```

- Loop invariant P should be true *whenever* line 3 is executed
- P may be broken in lines 4 - 5 but it must be true upon returning to line 3
- P should still be true when we exit the loop

Postcondition

- Loop invariant true AND guard false should imply the postcondition of the loop

```
int x = 10;
int y = 0;
while (y < x) {
        y++;
}
```

Loop invariant? y <= x

Postcondition? y == x

Termination (Progress)

- Identify a decrementing function that strictly decreases
- Decrementing function cannot decrease indefinitely
- Therefore, the loop must terminate in a finite number of steps

Introductory Example: Maximum of an Array

Given:
Int[] a = <non-empty array of positive #s>

Prove:
The following loop returns the maximum of a

Maximum of Array: Code

```
int max = 0;
int m = 0;
for (int i = 0; i < a.length; i++) {
    if (a[i] > m) {
        m = a[i];
    }
}
max = m;
```

Maximum of Array: Invariant

Coming up with invariant:

- **Precondition (exists?)**
- **Postcondition?**

```
int max = 0;
int m = 0;
for (int i = 0; i < a.length; i++) {
    if (a[i] > m) {
        m = a[i];
    }
}
max = m;
```

Maximum of Array: Invariant

Coming up with invariant:

m @ i-1 <= m @ i

m of a[0..i-1] <= m of a[0..i]

m is the max. element of a[0..i-1]

0 <= i <= a.length

```
int max = 0;
int m = 0;
for (int i = 0; i < a.length; i++) {
    if (a[i] > m) {
        m = a[i];
    }
}
max = m;
```

Max of Array: Establishment (Initialization)

True at init:

- i-1 is -1, as per our notation rules, a[0..-1] is [].
- Is m the maximum of []? Yes!

```
int max = 0;
int m = 0;
for (int i = 0; i < a.length; i++) {
    if (a[i] > m) {
        m = a[i];
    }
}
max = m;
```

Max of Array: Preservation (Maintenance)

True at top of loop:

- Assume inv. m is max of a[0..i-1]

We know the max of a[0..i] is just the max of max of a[0..i-1] and a[i]. Do we see this in the code?

```
int max = 0;
int m = 0;
for (int i = 0; i < a.length; i++) {
    if (a[i] > m) {
        m = a[i];
    }
}
max = m;
```

Max of Array: Preservation (Maintenance)

Yes!

```
int max = 0;
int m = 0;
for (int i = 0; i < a.length; i++) {
    if (a[i] > m) {
        m = a[i];
    }
}
max = m;
```

Max of Array: Preservation (Maintenance)

If m is max of a[0..i-1], and i<a.length, then by the highlighted code, m is max of a[0..i] by end of iteration. By next iter, m is again max of a[0..i-1] (because i has increased)

Preservation: proved B)

```
int max = 0;
int m = 0;
for (int i = 0; i < a.length; i++) {
    if (a[i] > m) {
        m = a[i];
    }
}
max = m;
```

Max of Array: Postcondition

Once guard is false: i == a.length
Loop invariant:
m is max of a[0..i-1]. So,
m is max of a[0..a.length-1],
which is the whole array.

So, guard false and loop invariant true implies the postcondition!

```
int max = 0;
int m = 0;
for (int i = 0; i < a.length; i++) {
    if (a[i] > m) {
        m = a[i];
    }
}
max = m;
```

Max of Array: Termination

How do we know loop terminates?

By the for loop guard. Consider our DF to be (a.length-i). This decreases each iteration, as i increases and a.length is unchanged.

```
int max = 0;
int m = 0;
for (int i = 0; i < a.length; i++) {
    if (a[i] > m) {
        m = a[i];
    }
}
max = m;
```

Another example: fast exponentiation

```
/**
 *  Returns: x^e
 *  Requires: e ≥ 0
 *  Performance: O(Log e)
 */
static int pow(int x, int e) {
    int r = 1, b = x, y = e;
    // Loop invariant: r·b^y = x^e  and  y ≥ 0
    while (y > 0) {
        if (y % 2 == 1) r = r*b;
        y = y/2;
        b = b*b;
    }
    return r;
}
```

- Common algorithm for computing powers efficiently
- Wish to prove correctness using loop invariants
- What should…
  - the precondition be?
  - the loop invariant be? (given)
  - the postcondition be?

Fast exponentiation: Establishment

- Does the invariant hold just before we begin execution of the loop?
- Yes! Why?
  - When r = 1, b = x, and y = e, it follows that $r*b^y = 1*x^e = x^e$ and y >= 0 since e >= 0 by assumption
  - So the loop invariant holds!
  - The precondition that e >= 0 is crucial

```
/**
 *  Returns: x^e
 *  Requires: e ≥ 0
 *  Performance: O(log e)
 */
static int pow(int x, int e) {
    int r = 1, b = x, y = e;
    // loop invariant: r·b^y = x^e  and  y ≥ 0
    while (y > 0) {
        if (y % 2 == 1) r = r*b;
        y = y/2;
        b = b*b;
    }
    return r;
}
```

Fast exponentiation: Preservation

```
/**
 *  Returns: x^e
 *  Requires: e ≥ 0
 *  Performance: O(Log e)
 */
static int pow(int x, int e) {
    int r = 1, b = x, y = e;
    // Loop invariant: r·b^y = x^e  and  y ≥ 0
    while (y > 0) {
        if (y % 2 == 1) r = r*b;
        y = y/2;
        b = b*b;
    }
    return r;
}
```

- Let y, r, and b be such that $y > 0$ and $r*b^y = x^e$
  - So both the loop invariant and the guard hold and we enter the loop body
- Let y', r', and b' be the values stored in y, r, and b after *one* iteration of body
- Does the loop invariant still hold with y', r', and b'?

Fast exponentiation: Preservation

```
/**
 *  Returns: x^e
 *  Requires: e ≥ 0
 *  Performance: O(log e)
 */
static int pow(int x, int e) {
    int r = 1, b = x, y = e;
    // loop invariant: r·b^y = x^e  and  y ≥ 0
    while (y > 0) {
        if (y % 2 == 1) r = r*b;
        y = y/2;
        b = b*b;
    }
    return r;
}
```

YES! Why?

- Suppose y was odd so (y % 2 == 1). Then r’ = r*b, y’ = (y-1)/2, b’ = b^2
  - We verify $r'*(b')^{y'} = (r*b)*(b^2)^{(y-1)/2} = (r*b)b^{y-1} = r*b^y = x^e$ and y’ >= 0 since y > 0
  - Thus invariant is preserved in this case
- Suppose y was even so (y % 2 == 0). Then r’ = r, y’ = y/2, b’ = b^2
  - We verify $r'*(b')^{y'} = r*(b^2)^{y/2} = r*b^y = x^e$ and y’ >= 0 since y > 0
  - Thus invariant is preserved in this case
- Invariant is preserved in all cases!

Fast exponentiation: Postcondition

- The desired result is $r = x^e$
- Suppose that the loop terminates, so (y > 0) eventually becomes false
- Since y >= 0 by the invariant, and the *invariant still holds*, we can confidently conclude y == 0!
- Also by the invariant:
  - $r*b^0 = x^e$
  - $r*1 = x^e$
  - $r = x^e$
- Therefore, if we terminate, we terminate with the correct result

```
/**
 *  Returns: x^e
 *  Requires: e ≥ 0
 *  Performance: O(log e)
 */
static int pow(int x, int e) {
    int r = 1, b = x, y = e;
    // Loop invariant: r·b^y = x^e  and  y ≥ 0
    while (y > 0) {
        if (y % 2 == 1) r = r*b;
        y = y/2;
        b = b*b;
    }
    return r;
}
```

Fast exponentiation: Termination

```
/**
 *  Returns: x^e
 *  Requires: e ≥ 0
 *  Performance: O(log e)
 */
static int pow(int x, int e) {
    int r = 1, b = x, y = e;
    // loop invariant: r·b^y = x^e  and  y ≥ 0
    while (y > 0) {
        if (y % 2 == 1) r = r*b;
        y = y/2;
        b = b*b;
    }
    return r;
}
```

- We have proved that, *if* the loop terminates, the correct result is returned
- How do we know we always terminate?
- In general, can be complicated, but easy here:
    - Since we only (integer) divide y by 2 in each iteration, y is strictly decreasing
    - This cannot continue indefinitely since we constrain y >= 0 in the loop invariant
    - Therefore eventually y == 0 and we terminate

Hoare Logic

Formally prove partial correctness statements!

Take CS 4110! Or CS 4160! Or most classes at Cornell covering programming languages!