15-122: Principles of Imperative Computation

Recitation 2 Solutions — Josh Zimmerman

GCD

```
1 int fast_gcd(int x, int y)
2 //@requires x > 0 && y > 0;
3 //@ensures \result == ref_GCD(x,y);
4 {
5   int a = x;
6   int b = y;
7   while (a != b)
8   //@loop_invariant a > 0 && b > 0;
9   //@loop_invariant ref_GCD(a, b) == ref_GCD(x, y);
10  {
11    if (a > b) {
12      a = a - b;
13    }
14    else {
15      b = b - a;
16    }
17  }
18  return a;
19 }
```

This is certainly faster, but does it actually work? Here's the proof:

Solution:

Part 1: Precondition implies loop invariant

First loop invariant: `a > 0 && b > 0`

By precondition, `x > 0 && y > 0`. By line 5, `a == x` and by line 6, `b == y`. Thus, `a > 0 && b > 0`.

Second loop invariant: `ref_GCD(a, b) == ref_GCD(x, y)`

We know that `a == x` and `b == y`, so thus `ref_GCD(a, b) == ref_GCD(x, y)`

Part 2: Preservation of loop invariants

First, assume that both invariants hold at the start of some iteration.

We have three cases: either `a > b`, `b > a`, or `a == b`.

If `a == b`, then we don't enter the loop and so we don't need to consider this case, since there are no more checks of the loop invariants after the last one that succeeded by assumption.

If `a > b`, we enter into the case on lines 11-13.

For the first loop invariant, we know that `b' = b` and that `a' = a - b`. Since `b > 0`, `b' > 0` and since `a > b`, we know that `a' > 0`.

For the second loop invariant, we can apply the mathematical observation made above: If $a > b$, $\gcd(a, b) = \gcd(a - b, b)$.

Since `a > b`, we know that `ref_GCD(a, b) = ref_GCD(a - b, b)` and since `a' == a - b` (and `b' == b`), we know that `ref_GCD(a, b) = ref_GCD(a', b')`. By our assumption, `ref_GCD(a, b) == ref_GCD(x, y)`, so `ref_GCD(a', b') == ref_GCD(x, y)` by the transitivity of equality.

The case when `b > a` is essentially the same, but switching the letters we use.

Thus, the loop invariant holds at the end of the iteration.

Part 3: Loop invariant and negation of the loop exit condition imply postcondition

By the loop invariant, when we exit the loop `ref_GCD(a, b) == ref_GCD(x, y)`. Further, by the negation of the loop exit condition, `a == b`. Since $\gcd(a, a) = a$ for all positive a, we know that `ref_GCD(x, y) == ref_GCD(a, b) == a`. We return `a`, which is `ref_GCD(x, y)` so therefore our postcondition is satisfied if the loop terminates.

Part 4: Termination of the loop

We're almost done now—we just need to argue that the loop does, in fact, exit.

The argument for this is somewhat subtle. The trick is that we look at the quantity $\max(a, b)$. By the first loop invariant, we know that $\max(a, b) > 0$. So, if we can show that after every iteration of the loop, $\max(a', b') < \max(a, b)$, then we know that eventually the loop must exit since $\max(a, b)$ can never go below 1. Now, let's look at that proof.

We again have three cases: either `a > b`, `b > a`, or `a == b`.

If $a > b$, then $\max(a, b) = a$, and $\max(a - b, b) < a$. (Since $b < a$ and $a - b < a$.) Thus, $\max(a', b') < \max(a, b)$ in this case.

If $a == b$, then we exit the loop and terminate.

If $b > a$, then we have an essentially identical case to when $a > b$ and can argue that $\max(a, b)$ decreases simply by changing a few letters.

Therefore, `\result == ref_GCD(x, y)`.

Another buggy program

Solution: Here's the annotated version of the source:

```
1 int add (int x, int y)
2 //@ensures \result == x + y;
3 {
4    int a = x;
5    int b = y;
6    while (b >= 0)
7    //@loop_invariant a + b == x + y;
8    {
9       a++;
10      b--;
11   }
12   // We need b == 0 here because we can't get a strong enough statement from
13   // just the loop invariant. If b < 0 , the loop invariant would be
14   // true, and the result might not be correct from just the loop invariant.
15   //@assert b == 0;
16   return a;
17 }
```

Note that the `//@assert` fails when we call `add(1, 1)`. Adding some print statements to investigate this leads us to the discovery that `b == -1` when we check the `//@assert` . Now, if we look at the loop condition again, we can see that it is incorrect. It should say `while(b > 0)` — otherwise, we're adding one too many times. So, we fix that, and we see that `add(1, 1)` works with no annotation failures and gives 2, as expected.

We're not done, though—we should still try to prove that the function always works, because it's possible that there's another case that doesn't work.

We don't have any `//@requires` statements yet, so let's just try to prove that `add` works for all integers `x` and `y`.

Before the first iteration of the loop, `a == x` and `b == y`, so `a + b == x + y`.

Now, we want to prove that if `a + b == x + y`, then `a' + b' == x' + y'` (we write the values of `a` and `b` after the next iteration as `a'` and `b'`).

After the loop, `a' == a + 1` and `b' == b - 1`. (Note: these calculations may overflow, but that's OK since the overflow would happen if we did `x + y` as well.) This means that `a' + b' == a + 1 + b - 1`, which is equal to `a + b`. By our assumption, that's equal to `x + y`.

Therefore, our loop invariant holds. Now, we need to prove that the `//@assert` holds. We know by the loop exit condition that `!(b > 0)`, so therefore `b <= 0`. However, there's nothing stopping `b` from being negative at this point, which is problematic. We need to require `y >= 0` to ensure a non-negative result at the end of the function. It's also helpful to add a loop invariant that `b` will always be non-negative. At this point, after the corrections we've made, here's our add function:

```
1 int add (int x, int y)
2 //@requires y >= 0;
3 //@ensures \result == x + y;
4 {
5    int a = x;
6    int b = y;
7    while (b > 0)
8    //@loop_invariant a + b == x + y;
9    //@loop_invariant b >= 0;
10   {
11      a++;
12      b--;
13   }
14   // We need b == 0 here because we can't get a strong enough statement from
15   // just the loop invariant. If b < 0 , the loop invariant would be
16   // true, and the result might not be provably correct based on
17   // just the loop invariant.
18   //@assert b == 0;
19   return a;
20 }
```

We must now prove that the second loop invariant holds. It's true initially by the precondition and the fact that `b == y` before we enter the loop. Now, assume that it's true before some iteration of the loop. We also know that, since we're executing another iteration of the loop, `b > 0` by the loop guard. Further, `b' = b - 1`. Since `b > 0`, `b' >= 0` and the loop invariant holds.

Now, let's return to proving that the `//@assert` holds. By the loop invariant, `b >= 0`. Since `!(b > 0)`, or in other words `b <= 0` as well, we know that `b == 0`.

Now, we want to show that that fact combined with the loop invariant implies that the function is correct.

We know that `a + b == x + y` and that `b == 0`. Thus, `a == x + y`.

Finally, we should prove that the function always terminates. `b >= 0` when we start looping, and we decrease `b` each iteration of the loop. Thus, `b` will eventually be 0 and we'll exit the loop.