15-122 : Principles of Imperative Computation, Fall 2013

Written Homework 1 Solutions

Name: ______________________________

Andrew ID: ______________________________

Recitation: ______________________________

The theory portion of this week's homework will introduce you to the way we reason about C0 code in 15-122.

2. **The preservation of loop invariants**

The core of proving the correctness of a function with a loop is proving that the loop invariant is *preserved* – that if the loop invariant holds at the beginning of a loop, it still holds at the end.

For each of the following loops, state whether the loop invariant is always preserved or not. If you say that the loop invariant is always preserved, prove this. If you say that the loop invariant is not always preserved, give initial values of the assignable variables such that the loop guard and loop invariant will hold before the loop body executes, but where the loop invariant will not hold after the loop body executes.

In this problem and in the next one, we will give solutions to some of the questions to give you some idea of what we are looking for. We will give two answers: a "long form" answer where we write out all the reasoning, and a "short form" answer where we just give the facts that we know to be true and the line or lines that justify those facts. You can answer questions either way; we prefer the shorter version.

(0) (a)
```
/* 1 */  while(j < 10000)
/* 2 */    //@loop_invariant 2 * i == j;
/* 3 */    {
/* 4 */      i = i+2;
/* 5 */      j = j+4;
/* 6 */    }
```

Solution: The loop invariant is always preserved.

Long version: From line `1`, we know that `j < 10000` at the beginning of the loop and from line `2` we know `2 * i` = `j` at the beginning of the loop.

We use primed variables to refer to the values stored in `i` and `j` at the end of the loop. Therefore, to show that the loop invariant is preserved, we need to show that `2 * i'` = `j'` where `i'` = `i+2` (line `4`) and `j'` = `j+4` (line `5`).

Therefore we have that `2 * i'` = `2 * (i+2)` = `2 * (i+2)` = `2*i + 4` = `2*i + 4` = `j + 4` = `j'`, which by transitivity is what we needed to show.

Short version:

– `2 * i'` = `2 * (i+2)`	(line 4)
– `2 * (i+2)` = `2*i + 4`	(distributivity)
– `2*i + 4` = `j + 4`	(line 2)
– `j + 4` = `j'`	(line 5)
– `2 * i'` = `j'`	(transitivity, the four preceding facts)

(2) (b)
```
/* 1 */  while (k <= n)
/* 2 */    //@loop_invariant i*i == k;
/* 3 */    {
/* 4 */      k = k + 2*i + 1;
/* 5 */      i = i + 1;
/* 6 */    }
```

Solution: The loop invariant is always preserved.

– `k’` = `k + 2*i + 1`	(line 4)
– `k + 2*i + 1` = `i*i + 2*i + 1`	(line 2)
– `i*i + 2*i + 1` = `(i + 1)*(i + 1)`	(math)
– `(i + 1)*(i + 1)` = `i’*i’`	(line 5)
– `i’*i’` = `k’`	(transitivity, the last 4 facts)

(2) (c)
```
/* 1 */  while(i < x)
/* 2 */    //@loop_invariant x <= y;
/* 3 */    //@loop_invariant i < y;
/* 4 */    {
/* 5 */      i++;
/* 6 */    }
```

Solution: The loop invariant is not always preserved.

Let `x` = `y` = 4, and let `i` = 3. After the loop runs, `i` will be 4 so `i < y` will evaluate to `false`.

(2) (d)

```
/* 1 */  while (a != b)
/* 2 */    //@loop_invariant a > 0 && b > 0;
/* 3 */    {
/* 4 */      if (a > b) {
/* 5 */        a = a - b;
/* 6 */      } else {
/* 7 */        b = b - a;
/* 8 */      }
/* 9 */    }
```

Solution: The loop invariant is always preserved.

We reason by case analysis on the relationship between the integers `a` and `b`.

Case 1: `(a > b)`

– `a'` = `a - b`	(line 4, line 5)
– `b'` = `b`	(line 4)
– `b > 0`	(line 2)
– `a' > 0`	$(a > b) \land (a' = a - b) \Rightarrow a' > 0$
– `b' > 0`	$(b > 0) \land (b' = b) \Rightarrow b' > 0$
– `a' > 0 && b' > 0`	(previous two facts)

Case 2: `(a < b)`

– `a'` = `a`	(line 4)
– `b'` = `b - a`	(line 4, line 7)
– `a > 0`	(line 2)
– `a' > 0`	$(a > 0) \land (a' = a) \Rightarrow a' > 0$
– `b' > 0`	$(a < b) \land (b' = b - a) \Rightarrow b' > 0$
– `a' > 0 && b' > 0`	(previous two facts)

Case 3: `(a == b)`

Because we know `a != b` (line 1), this case is impossible.

(1) (e)
```
/* 1 */  while (e > 0)
/* 2 */    //@loop_invariant e > 0 || accum == POW(x,y);
/* 3 */    {
/* 4 */      accum = accum * x;
/* 5 */      e = e - 1;
/* 6 */    }
```

Solution: The loop invariant is not always preserved.

Lots of counterexamples here where `e = 1` before the loop body runs.

Here's one example: if `x` = `y` = `accum` = `3`, and `e` = 1, then after the loop `accum` = `9` and `e` = `0`. `POW(3,3) = 27` which is not `9`, and `0` is not greater than `0`, so after the loop body the loop invariant does not hold.

(2) (f)
```
/* 1 */  while(x == 2*y)
/* 2 */    //@loop_invariant i == 4*j;
/* 3 */    {
/* 4 */      i = i+2*x;
/* 5 */      j = j+y;
/* 6 */      x = f(i);
/* 7 */    }
```

Solution:

– `i’` = `i+2*x`	(line 4)
– `i+2*x` = `4*j + 2*x`	(line 2)
– `4*j + 2*x` = `4*j + 4*y`	(line 1)
– `4*j + 4*y` = `4*(j+y)`	(distributivity)
– `4*(j+y)` = `4*j’`	(line 5)
– `i’` = `4*j’`	(transitivity, the last 5 facts)

3. **Assertions in loops**

This question involves a series of functions `f` with one loop; each contains additional `//@assert` statements. None of the assertions will ever fail – they will never evaluate to `false` when the function `f` is called with arguments that satisfy the precondition. However, if our loop invariants aren't up to the task, we may not be able to *prove* these assertions hold.

When assignable variables are *untouched* by a loop, statements we know to be true about those untouched assignables *before* the loop remain valid *inside* the loop and *after* the loop. For assignables that are modified by the loop, the loop guard and the loop invariants are the only statements we can use. Inside of a loop, we know that the loop invariant held just before the loop guard was checked and that the loop guard returned `true`. After a loop, we know that the loop invariant held just before the loop guard was checked for the last time and that the loop guard returned `false`.

For each of the problems below, you can assume that the loop invariant is true initially (before the loop guard is checked the first time) and that it is always preserved.

(0) (a)

```
/*  1 */  int f(int a, int b)
/*  2 */  //@requires 0 <= a && a < b;
/*  3 */  {
/*  4 */    int i = 0;
/*  5 */    while (i < a) {
/*  6 */      //@assert i < b; /*** Assertion 1 ***/
/*  7 */      i += 1;
/*  8 */    }
/*  9 */    //@assert i == a; /*** Assertion 2 ***/
/* 10 */    return i;
/* 11 */  }
```

Solution: `Assertion 1` is supported.

Long version: Because the assignables `a` and `b` are not modified by the loop, the assertion `a < b` from line `2` can be used at line `6`. Because we are inside the loop, we know the loop guard held at the beginning of the loop, so line `5` gives us that `i < a`. The facts `i < a` and `a < b` together imply `i < b`.

Short version:

- `i < a` (line 5)
- `a < b` (line 2)
- `i < b` $(\texttt{i < a}) \land (\texttt{a < b}) \Rightarrow (\texttt{i < b})$

Solution: `Assertion 2` is unsupported.

At line `9`, we know that the loop guard `i < a` is false – that is, we know that `!(i < a)`, which is the same thing as saying `i >= a`. We can't conclude, from this, that `i` is equal to `a`.

(3) (b)

```
/*  1 */  int f(int a, int b)
/*  2 */  //@requires 0 <= a && a <= b;
/*  3 */  {
/*  4 */    int i = 0;
/*  5 */    while (i < a)
/*  6 */      //@loop_invariant 0 <= i;
/*  7 */      {
/*  8 */        //@assert 0 <= i && i < b; /*** Assertion 3 ***/
/*  9 */        i += 1;
/* 10 */      }
/* 11 */    //@assert i <= b; /*** Assertion 4 ***/
/* 12 */    return i;
/* 13 */  }
```

Solution: `Assertion 3` is supported.

We have that `0 <= i` from line `6`.

We have `i < b` from lines `5` and `2` – (`0 <= i`) $\wedge$ (`a <= b`) $\Rightarrow$ `i < b`.

Solution: `Assertion 4` is unsupported.

This just mimics `Assertion 2`. We know `i >= a` from line `5`, so `i` is greater than or equal to `a` but that doesn't mean it's less than `b`.

(3) (c)

```
/*  1 */  int f(int a, int b)
/*  2 */  //@requires 0 <= a && a <= b;
/*  3 */  {
/*  4 */    int i = 0;
/*  5 */    while (i < a)
/*  6 */      //@loop_invariant i <= a;
/*  7 */      {
/*  8 */        //@assert i < b; /*** Assertion 5 ***/
/*  9 */        i += 1;
/* 10 */      }
/* 11 */    //@assert i == a; /*** Assertion 6 ***/
/* 12 */    return i;
/* 13 */  }
```

Solution: `Assertion 5` is supported.
We have `i < b` from lines `5` and `2` – (`i < a`) $\wedge$ (`a <= b`) $\Rightarrow$ `i < b`.

Solution: `Assertion 6` is supported.
We have `i <= a` from line `6`, and `i >= a` from line `5`; together these imply that `i == a`.

(3) (d)

```
/*  1 */  int f(int a, int b)
/*  2 */  //@requires 0 <= a && 2*a < b;
/*  3 */  {
/*  4 */    int i = 0;
/*  5 */    while (i < a) {
/*  6 */      //@assert i < b; /*** Assertion 7 ***/
/*  7 */      i += 2;
/*  8 */      a += 1;
/*  9 */    }
/* 10 */    //@assert a <= i; /*** Assertion 8 ***/
/* 11 */    return i;
/* 12 */  }
```

Solution: `Assertion 7` is unsupported. Because `a` is modified by the loop, we cannot use line `2` to reason about the relationship between `i` and `a` inside of the loop, so we have no way of knowing the relationship between `i` and `b`.

For your perusal, a version of this loop where the loop invariants *do* entail `Assertion 7` follows; we have to create a new temporary variable `j` and use that instead of `a` to preserve the old value of `a` while we're inside the loop.

```
int f(int a, int b)
//@requires 0 <= a && 2*a < b;
{
  int i = 0;
  int j = a;
  while (i < j)
    //@loop_invariant 0 == i%2;
    //@loop_invariant a == j - i/2;
    {
      //@assert i < b; /*** Assertion 7 ***/
      i += 2;
      j += 1;
    }
  //@assert j <= i; /*** Assertion 8 ***/
  return i;
}
```

By the last two loop invariants, we have that `2*a == 2*j - i`, and so this, together with the precondition `2*a <= b`, gives us that `2*j - i < b`. Because of the loop guard, we know that `2*j - i` is greater than `j`, which is in turn greater than `i`. So we have `i < j < 2*j - i < b`, which is what we needed.

Solution: `Assertion 8` is supported.

The statement `a <= i` is just the negation of the loop guard, line `5`.