15-122 : Principles of Imperative Computation, Spring 2013

Homework 0 Theory

Due: Thursday, January 24, 2013 at the **beginning** of lecture

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.

Question	Points	Score
1	2	
2	10	
3	9	
4	4	
Total:	25	

You *must* use this printout, include this cover sheet,
and staple the whole thing together before turning it in.
Either type up the assignment using 15122-theory0.tex,
or print this PDF and write your answers *neatly* by hand.

1. **Running C0 programs**

(2) (a) The file `foo.c0` contains a function `foo` that takes an integer and returns an integer. From the command line, give two different ways of testing the value of `foo(12)`; one is given.

Solution:

```
% echo "int main() { return foo(12); }" > foo-test.c0
% cc0 -d -x foo.c0 foo-test.c0
33
```

Solution: (Use the `cc0` compiler without `-x` and with the `-o` option.)

Solution: (Use the Coin interpreter.)

You are welcome (on this and other theory homeworks) to actually type out and test your solutions. Keep in mind, though: one reason we ask you to write code by hand in homework is because, on the midterms and final, you will be required to read and write code by hand without the benefit of a computer. One strategy is to try to do homeworks without a compiler at first, but then test your answers just like you'd test your programming assignments. But do what works for you!

Finally, note that the problem above is problem 1(a). The (2) that is in parentheses next to the left of the (a) refers to how many points problem 1(a) is worth.

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, but where the loop invariant will not hold after the loop.

   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:

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

Solution:

(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)`

Case 2: `(a < b)`

Case 3: `(a == b)`

Because we know `a != b` (line 1), this case entails a contradiction, and therefore it can never occur.

(2) (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.

(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:

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` (`i < a`) $\wedge$ (`a < b`) $\Rightarrow$ (`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

Solution: `Assertion 4` is

(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

Solution: `Assertion 6` is

(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

Solution: `Assertion 8` is

4. **Thinking about how proofs can break**

In this problem, we will work with loop invariants for the power function.

```
int pow_impl(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int accum = 1;
  int e = y;
  while (e > 0) {
    accum = accum * x;
    e = e - 1;
  }
  return accum;
}
```

(2) (a) Give a loop invariant that is true initially, is preserved by any iteration of the loop, but that does not entail the postcondition.

Solution:

(b) Give a "loop invariant" that is not true initially, but that is preserved by any iteration of the loop. (It can entail the postcondition or not, either way is fine.)

Solution:

(2) (c) Recall from question 2(e) that the "loop invariant" `e > 0 || accum == POW(x,y)` is not preserved by every iteration of the loop. However, it is true initially and it does imply the postcondition. (You should convince yourself of this.)

Come up with a *different* loop body where `e > 0 || accum == POW(x,y)` *is* a provably valid loop invariant. You're generally not allowed to use specification functions in your code, but in this case you may use the `POW()` function in your revised loop body. Your solution must terminate, otherwise just leaving the loop body empty would be a valid solution! (You should convince yourself of this, too.)

Solution:

```
while (e > 0) {



}
```