15-122 : Principles of Imperative Computation, Spring 2014

Written Homework 1

Due: Thursday, January 23, 2014

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	9	
3	9	
Total:	20	

You *must* print this PDF and write your answers *neatly* by hand. You can hand in the assignment in class before the lecture starts.

1. **Running C0 programs**

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
```

(1) (a) Use the `cc0` compiler without `-x` and with the `-o` option.

Solution:

(1) (b) Use the Coin interpreter.

Solution:

You are encouraged (on this and other "written" 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 problems above are problem 1(a) and 1(b). The (1) that is in parentheses next to the left of the (a) and the (b) refers to how many points the sub-problems are 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 a specific counterexample, that is 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.

   Don't forget to consult the posted solved examples to get a good idea of what is expected for such questions.

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

Solution:

(2) (b)

```
/* 1 */  while (i < 24)
/* 2 */  //@loop_invariant i == 2*j;
/* 3 */  {
/* 4 */    if (i % 5 == 0){
/* 5 */      break;
/* 6 */    }
/* 7 */    i++;
/* 8 */    if(i % 7 == 0){
/* 9 */      continue;
/* 10 */   }
/* 11 */   j += 2;
/* 12 */ }
```

Solution:

(2) (c)

```
/* 1 */  while(0 < j && j <= 1000)
/* 2 */  //@loop_invariant j % 2 == (1 - i);
/* 3 */  {
/* 4 */    j = j + 3;
/* 5 */    i = (i + 1) % 2;
/* 6 */  }
```

Solution:

(2) (d)

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

Solution:

(2) (e)

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

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. The distinction between an assertion being *true* and *supported* is a subtle but important one.

   To support an assertion one may use the following facts:

   - 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, state whether each assertion is supported or not explain your reasoning. 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 1 <= a && a < b;
/*  3 */  {
/*  4 */    int i = 1;
/*  5 */    while (i < a)
/*  6 */    //@loop_invariant i >= 1
/*  7 */    {
/*  8 */      //@assert i < b; /*** Assertion 1 ***/
/*  9 */      i += 1;
/* 10 */    }
/* 11 */    //@assert i == a; /*** Assertion 2 ***/
/* 12 */    //@assert i != 0; /*** Assertion 3 ***/
/* 13 */    return i;
/* 14 */  }
```

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`.

Solution: `Assertion 3` is supported. After the loop, we know that the loop invariant `i >= 1` still holds. We can conclude from this, that `i` is not equal to `0`.

(3) (b)

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

Solution: `Assertion 4` is

Solution: `Assertion 5` is

(3) (c)

```
/*  1 */  int f(int a)
/*  2 */  //@requires 0 <= a;
/*  3 */  {
/*  4 */    int i = 2*a;
/*  5 */    while (i > a)
/*  6 */      //@loop_invariant i >= a;
/*  7 */      {
/*  8 */        //@assert i > 0; /*** Assertion 6 ***/
/*  9 */        a += 2;
/* 10 *          i += 1;
/* 11 */      }
/* 12 */    //@assert i <= a; /*** Assertion 7 ***/
/* 13 */    return i;
/* 14 */  }
```

Solution: `Assertion 6` is

Solution: `Assertion 7` is

(3) (d)

```
/*  1 */  int f(int a, int b)
/*  2 */  //@requires 0 <= a;
/*  3 */  {
/*  4 */    int i = 0;
/*  5 */    int accum = 1;
/*  6 */    while (i < a)
/*  7 */    //@loop_invariant accum == POW(b, i)
/*  8 */    {
/*  9 */      //@assert i <= a; /*** Assertion 8 ***/
/* 10 */      accum = accum * b;
/* 11 */      i = i + 1;
/* 12 */    }
/* 13 */    //@assert accum == POW(b, a); /*** Assertion 9 ***/
/* 14 */    return accum;
/* 15 */  }
```

Solution: `Assertion 8` is

Solution: `Assertion 9` is