15-122: Principles of Imperative Computation, Spring 2023

Written Homework 2

Due on Gradescope: Sunday 22nd January, 2023 by 9pm

Name: ____

Andrew ID: ____

Section: ____

This written homework covers more reasoning using loop invariants and assertions, and the C0 types **`int`** and **`bool`**.

Preparing your Submission You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- ***PDFescape*** or ***DocHub***, two web-based PDF editors that work from anywhere.
- ***Acrobat Pro***, installed on all non-CS cluster machines, works on many platforms.
- ***iAnnotate*** works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though*.

Caution Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

Submitting your Work Once you are done, submit this assignment on Gradescope. ***Always check it was correctly uploaded.*** You have unlimited submissions.

Question:	1	2	3	4	Total
Points:	4.5	4	3.5	3	15
Score:					

1. **Point-to Reasoning**

When writing code, we want its correctness to be as "obvious" as possible. In this class, the gold standard for this is whether or not it can be proven with *point-to reasoning* — we can prove it true without tracing its execution over more than one block of code. In particular, since reasoning over multiple iterations of a loop is hard to keep track of, we never want to do that.

In this question, you will judge some sample proofs and determine whether they are "point-to valid". If a proof is not, state which line is invalid and explain why. Refer to the lecture notes for a detailed discussion of point-to reasoning.

Some things to keep in mind:

- When reasoning *inside* a loop, you may not draw conclusions about variable changes over **multiple iterations** of this loop — only over the *current iteration*.
- When reasoning about an *earlier* loop, **you should pretend the body of that loop is unknown**!
- Similarly, **you should pretend the body of a function you are calling is unknown**! (. . . unless it's a specification function.)

Here are some things you **can** use:

- Statements about a variable that hasn't been changed.
- Recent Boolean expressions — in particular conditionals and loop guards.
- Statements based on contracts (such as that the loop invariants held just before the loop guard was checked).
- Assignments within the current block of code.

Here are three examples — study them carefully! Note that the second proof does prove the assertion! However, it uses reasoning that requires taking a "big leap" to the last iteration of the loop. While this assertion is arguably obvious, large code that requires this kind of reasoning frequently will not be as clearly correct.

```
int f(int a, int b)
//@requires 1 <= a && a < b;
{
    int i = 1;
    while (i < a)
    //@loop_invariant i >= 1;
    {
        //@assert i < b;          /*** Assertion 1 ***/
        i += 1;
        //@assert i >= 2;         /*** Assertion 2 ***/
    }
    //@assert i == a;             /*** Assertion 3 ***/
    return i;
}
```

Assertion 1, `//@assert i < b`, is *supposedly* supported by this point-to proof:

A.	`i < a`	by line 5
B.	`a < b`	by line 2 and `a` and `b` unchanged
Therefore we conclude that		
C.	`i < b`	by math on (A) and (B)

This proof <u>is</u> point-to valid (the assertion is inside the loop and the proof only uses the loop guard and a known fact about variables that are not changed in the loop).

Assertion 2, `//@assert i >= 2`, is *supposedly* supported by this point-to proof:

A.	`i == 1` initially	by line 4
B.	`i += 1` at each iteration	by line 9
Therefore we conclude that		
C.	`i >= 2`	by (A) and (B)

This proof <u>is not</u> point-to valid on step (C). This is because point-to reasoning doesn't permit drawing conclusions about variable changes over multiple iterations of the current loop.

(This assertion can be proved by point-to reasoning: `i >= 1` by the loop invariant on line 6, `i+1` can't overflow by line 5, `i = i+1` on line 9, and therefore `i >= 2` by math after line 9.)

Assertion 3, `//@assert i == a`, is *supposedly* supported by this point-to proof:

A.	`i = 1` initially	by line 4
B.	`i += 1`	by line 9
C.	`i >= a`	by the negation of loop guard on line 5
Therefore we conclude that		
D.	`i = a`	by (A–C) since `i` increases by 1 at each iteration, so it become equal to `a` and break the loop guard before it can exceed it

This proof <u>is not</u> point-to valid on step (B). This is because point-to reasoning doesn't permit peeking inside the body of an earlier loop (here `i += 1` on line 9).

(Note that this assertion is *true and this proof provides a convincing argument to support this, but it is not a point-to proof. As written, the above program does not allow any point-to proof of this assertion. However, simple changes in the provided contracts would make writing a valid point-to proof for it easy.)*

1.5pts

1.1

```
 1 int f(int a, int b, int c)
 2 //@requires 0 < a && a < b;
 3 {
 4   while (a < b)  {
 5     if (a <= c) return 42;
 6     a++;
 7     //@assert a != c;      /*** Assertion 1 ***/
 8   }
 9   //@assert a > c;         /*** Assertion 2 ***/
10 }
```

Assertion 1, `//@assert a != c`, is *supposedly* supported by this point-to proof:

A.	`a > c`	by line 5
B.	`a != int_max()`	by line 4
C.	`a+1 > a`	by math on (B)
D.	`a+1 > c`	by math on (A) and (C)
Therefore we conclude that		
E.	`a != c`	by line 6 and math on (D)

Is this proof point-to valid? □ **Yes** □ **No**

If "No", which step is not point-to valid? ________________ Explain why:

____

____

Assertion 2, `//@assert a > c`, is *supposedly* supported by this point-to proof:

A.	**`return`** if `a <= c`	by line 5
Therefore we conclude that		
B.	`a > c`	by math on (A)

Is this proof point-to valid? □ **Yes** □ **No**

If "No", which step is not point-to valid? ________________ Explain why:

____

____

1.5pts

1.2

```
1  int sub_one(int x) {
2    return x - 1;
3  }
4
5  int f(int x)
6  //@requires x > 0;
7  {
8    int a = x - 1;
9    //@assert a >= 0;        /*** Assertion 1 ***/
10   int b = sub_one(x);
11   //@assert a == b;        /*** Assertion 2 ***/
12   return a-b;
13 }
```

Assertion 1, `//@assert a >= 0` is *supposedly* supported by this point-to proof:

A.	`x > 0`	by line 6
B.	`x - 1 >= 0`	by math on (A)
Therefore we conclude that		
C.	`a >= 0`	by line 8 and (B)

Is this proof point-to valid? □ **Yes** □ **No**

If "No", which step is not point-to valid? ________________ Explain why:

____

____

Assertion 2, `//@assert a == b`, is *supposedly* supported by this point-to proof:

A.	`a = x - 1`	by line 8
B.	`b = sub_one(x)`	by line 10
C.	`b = x - 1`	by line 2
Therefore we conclude that		
D.	`a == b`	by math on (A) and (C)

Is this proof point-to valid? □ **Yes** □ **No**

If "No", which step is not point-to valid? ________________ Explain why:

____

____

0.5pts

1.3

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

Assertion 1, `//@assert a > 0`, is *supposedly* supported by this point-to proof:

A. `0 < a` initially	by line 2
B. `a` is always increasing	by line 6
Therefore we conclude that	
C. `a > 0`	by math on (A) and (B)

Is this proof point-to valid? □ **Yes** □ **No**

If "No", which step is not point-to valid? ________________ Explain why:

____

____

Your turn! For each of the assertions below:

- Either circle **SUPPORTED** if it can be proved by means of a valid point-to proof. In this case, provide this proof by filling in the lines with a relevant fact on the left and a justification for it on the right.
- Or circle **UNSUPPORTED** if no such proof exists. In this case, use the lines to write a short explanation of why there is no valid point-to proof for it.

In either case, you may not need all the lines provided.

1pt

1.4

```
int f(int a)
//@requires 0 <= a;
{
    int i = 2*a;
    while (i > a)
    //@loop_invariant i >= a;
    {
        //@assert i > 0;      /*** Assertion A ***/
        a += 2;
        i += 1;
    }
    //@assert i <= a;         /*** Assertion B ***/
    return i;
}
```

Assertion A is: **SUPPORTED/UNSUPPORTED**

a. ____________________

by

b. ____________________

by

c. ____________________

by

d. ____________________

by

e. ____________________

by

Therefore we **can**/**cannot** conclude that

e. ____________________

by

Assertion B is: **SUPPORTED/UNSUPPORTED**

a. ____________________

by

b. ____________________

by

c. ____________________

by

d. ____________________

by

e. ____________________

by

Therefore we **can**/**cannot** conclude that

e. ____________________

by

2. Basics of C0: the int and bool Data Types

1.5pts

2.1 Let *p* be an **int** in the C0 language. Express the following operations in C0 using only constants *in hexadecimal* and *only* the bitwise operators (&, |, ^, ~, <<, >>). Your answers should account for the fact that C0 uses 32-bit integers.

Each answer should consist of ONE line of C0 code. You can use multiple constants and multiple bitwise operations, but no loops and no additional assignment statements.

a. Set x equal to p with its lowest 8 bits cleared to 0 and with its middle 8 bits set to 1 (so that, for example, 0xAB12CD34 becomes 0xAB1FFD00).

```
int x = ________________________________________________;
```

b. Set y equal to p with its highest and lowest 16 bits swapped (so that, for example, 0x1234ABCD becomes 0xABCD1234)

```
int y = ________________________________________________;
```

c. Set z equal to p with its middle 16 bits flipped (0 ⟶ 1 and 1 ⟶ 0) (so that, for example 0xAB0F1812 becomes 0xABF0E712).

```
int z = ________________________________________________;
```

1pt

2.2 The function no_overflow_add is intended to return true if result of adding three non-positive numbers a, b, and c fits in 32 bits (if it does not overflow), and false if it does.

- If the following code satisfies this description, explain why in one sentence.
- If it doesn't satisfy this description, give 32-bit values for a, b, and c that satisfy the preconditions and such that the call no_overflow_add(a,b,c) returns true when it should have returned false, or vice versa. Explain why the result is incorrect in this case.

```
bool no_overflow_add(int a, int b, int c)
//@requires a <= 0 && b <= 0 && c <= 0;
{
  return a + b + c <= 0;
}
```

1.5pts

2.3 Two C0 expressions are equivalent, if identical inputs result in identical outcomes, including errors. For example, $(x * y)/y$ and x are not equivalent, as when y equals 0, the first expression raises an error. For each of the following statements, determine whether the statement is true or false in C0. If it is true, explain why. If it is false, give a counterexample using **valid C0** values to illustrate why.

a. For every **`int`** y: `(y != 0) && (y / y == 1)` is equivalent to `true`.

○ **Equivalent** *because*	○ **Not equivalent** *Counterexample:* $y =$ ____________________

b. For every **`int`** x, $x < 0$ is equivalent to $-x > 0$

○ **Equivalent** *because*	○ **Not equivalent** *Counterexample:* $x =$ ____________________

c. For every **`int`** x, y and z, $x * y < z$ is equivalent to $x < z/y$

○ **Equivalent** *because*	○ **Not equivalent** *Counterexample:* $x =$ ____________________ $y =$ ____________________ $z =$ ____________________

d. For every **`int`** x: $(x$ `<< 1) >> 1` is equivalent to x.

○ **Equivalent** *because*	○ **Not equivalent** *Counterexample:* $x =$ ____________________

e. For every **`int`** x and y:
$(x\%y < 122)\ \texttt{\&\&}\ (y\ \texttt{!=}\ 0)$ is equivalent to $(y\ \texttt{!=}\ 0)\ \texttt{\&\&}\ (x\%y < 122)$

○ **Equivalent** *because*	○ **Not equivalent** *Counterexample:* $x =$ ____________________ $y =$ ____________________

3. **Proving the correctness of functions with one loop**

   The Fibonacci sequence is shown below:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

Each integer i_n, $n \geq 2$, in the sequence is the sum of i_{n-1} and i_{n-2}. By definition, $i_0 = 0$ and $i_1 = 1$.

Consider the following implementation for `fast_fib` that returns the n^{th} Fibonacci number, and the specification function `FIB` for it. The body of the loop is not shown.

```
int FIB(int n)
//@requires n >= 0;
{
    if (n <= 1) return n;
    else return FIB(n-1) + FIB(n-2);
}

int fast_fib(int n)
//@requires n >= 0;
//@ensures \result == FIB(n);
{
    if (n <= 1) return n;
    int a = 0;
    int b = 1;
    int c = 1;
    int x = 2;

    while (x < n)
    //@loop_invariant 2 <= x && x <= n;
    //@loop_invariant a == FIB(x-2);
    //@loop_invariant b == FIB(x-1);
    //@loop_invariant c == a + b;
    {
       // LOOP BODY NOT SHOWN: modifies a, b, c, and x
    }
    return c;
}
```

In this problem, we will reason about the correctness of the `fast_fib` function when the argument `n` is greater than or equal to 2, and we will complete the implementation based on this reasoning.

(NOTE: To completely reason about the correctness of `fast_fib`, we also need to point out that `fast_fib(0) == FIB(0)` and that `fast_fib(1) == FIB(1)`. This is straightforward, because no loops are involved.)

Note: The completed solution below shows you a general format for showing that a postcondition holds given a valid loop invariant. The English explanation is kept to a minimum and point-to reasoning plays a large role. In the future, you may be asked to write an entire solution in a clear, concise manner, and the solution below gives you an example of how you might write such a solution.

1pt

3.1 Loop invariant and negation of the loop guard imply postcondition

Complete the argument that the postcondition is satisfied assuming valid loop invariant(s) by giving appropriate line numbers. Use point-to reasoning.

We know `x <= n` by line ____________________ and

we know `x >= n` by line ____________________, which implies that `x == n` by logic.

The returned value `\result` is the value of `c` after the loop, so to show that the postcondition on line 10 holds when `n >= 2`, it suffices to show `c == FIB(n)` after the loop.

A `a == FIB(x-2)` by ____________________

B `b == FIB(x-1)` by ____________________

C `x >= 2` by ____________________

D `c == a + b` by ____________________

E `== FIB(x-2) + FIB(x-1)` by ____________________

F `== FIB(x)` by C and definition of `FIB`

1pt

3.2 Loop invariant holds initially

Complete the argument for the loop invariants holding initially by giving appropriate line numbers. You do not need to cite the definition or code of `FIB`.

The loop invariant `2 <= x` on line 19 holds initially by line(s) ______________.

The loop invariant `x <= n` on line 19 holds initially by line(s) ______________.

The loop invariant on line 20 holds initially by line(s) ____________________.

The loop invariant on line 21 holds initially by line(s) ____________________.

The loop invariant on line 22 holds initially by lines 13, 14, and 15.

1pt

3.3 The loop invariant is preserved through any single iteration of the loop

Based on the given loop invariants, write the body of the loop. **DO NOT use the specification function `FIB()`. The specification function is meant to be used in contracts only. Also, do not call `fast_fib` recursively, since this isn't fast!**

(NOTE: To check your answer, you would prove that the loop invariants are preserved by an arbitrary iteration of the loop, but you don't have to do that for us here — we'll cover that process in the next question.)

```
while (x < n)
//@loop_invariant 2 <= x && x <= n;
//@loop_invariant a == FIB(x-2);
//@loop_invariant b == FIB(x-1);
//@loop_invariant c == a + b;
{
    x = ____________________________________________________;

    a = ____________________________________________________;

    b = ____________________________________________________;

    c = ____________________________________________________;
}

return c;
```

0.5pts

3.4 The loop terminates

The postcondition is satisfied only if the loop terminates. Explain concisely why the function must terminate with the loop body you gave in the previous task. The expression you write must be decreasing.

The integer expression ____________________ is strictly decreasing because

Since the loop terminates if the value of this expression reaches 0 or less and it is strictly decreasing, the loop must terminate.

4. The Preservation of Loop Invariants

For each of the following loops, state whether the loop invariant is ALWAYS PRESERVED or NOT ALWAYS PRESERVED.

- If you say that the loop invariant is always preserved, prove it using point-to reasoning.
- If you say that the loop invariant is not always preserved, give a *specific counterexample*. To do so, you must provide *specific, concrete* values of all local variables such that
  - the loop guard and loop invariant hold before the loop body executes for some iteration,
  - the loop invariant will not hold after the loop body executes that one iteration,
  - if the code mentions a function you don't know anything about, you may define it as you wish in your counterexample.

Here are two solved examples to give you an idea of how to write your solutions. Integers are defined as C0's 32-bit signed two's-complement numbers; be careful about this when you think about counterexamples!

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

Solution: NOT ALWAYS PRESERVED

Counterexample: x=2 and y=3, satisfies loop invariant and loop guard.

After this iteration, x=3 and y=3, violating loop invariant.

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

Solution: ALWAYS PRESERVED.

Assume: `x < y + 1` (by line 2) before an iteration.

To show: `x' < y + 1` after an iteration.

Since `x' = x + 2` (by line 4), we need to show `x + 2 < y + 1`.

A. `x + 1 < y`	by line 1
B. `x + 2 <= y`	by math (because `x + 1 < y`)
C. `y < y + 1`	by line 2 that lets us know `y != int_max()`
D. `x + 2 < y + 1`	by B and C

1pt

4.1

```
11 while (a != b)
12 //@loop_invariant a > 0 && b > 0;
13 {
14   if (a > b) {
15     a = a - b;
16   } else {
17     b = b - a;
18   }
19 }
```

ALWAYS PRESERVED (*Complete the indicated parts of the proof — you may not need all lines provided)*

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

Assume: `a > 0 && b > 0`

To show: `a‘ > 0 && b‘ > 0`

Case 1, `(a > b)`:

A. ______________________ by ______________________

B. ______________________ by ______________________

C. ______________________ by ______________________

D. ______________________ by ______________________

E. ______________________ by ______________________

F. ______________________ by ______________________

G. ______________________ by ______________________

H. ______________________ by ______________________

I. ______________________ by ______________________

Case 2, `(a < b)`: similar (trust us!)

Case 3, `(a == b)`:

Because we know `a != b` (line `11`), this case is impossible.

0.5pts

4.2

```
while (i < 24)
//@loop_invariant 2*i == j;
{
    i++;
    if (i % 7 != 4) {
        j += 2;
    }
}
```

○ **Always preserved**	○ **Not always preserved**
Assume: ____________________	*Counterexample:*
To show: ____________________	`i =` __________
Proof:	`j =` __________
A. __________________ by ______	
B. __________________ by ______	
C. __________________ by ______	
D. __________________ by ______	
E. __________________ by ______	

0.5pts

4.3

```
while (a != b)
//@loop_invariant a > b || b > a;
{
    if (a > b) {
        a = a - b;
    } else {
        b = b - a;
    }
}
```

○ **Always preserved**	○ **Not always preserved**
Assume: ____________________	*Counterexample:*
To show: ____________________	`a =` __________
Proof:	`b =` __________
A. __________________ by ______	
B. __________________ by ______	
C. __________________ by ______	
D. __________________ by ______	
E. __________________ by ______	

0.5pts

4.4 In this task, you know nothing about what `f` computes, other than that it always returns the same result on the same inputs.

```
 while (i < j)
//@loop_invariant (i + j)/2 == f(lo, hi);
{
    i = i + 2;
    j = j - 2;
    k = (2*i+1)/(2*lo+1) + (2*j-1)/(2*hi-1);
}
```

○ **Always preserved**	○ **Not always preserved**
Assume: ________________	*Counterexample:*
To show: ________________	`i =` ________
Proof:	`j =` ________
A. ________________ by ________	`k =` ________
B. ________________ by ________	`f(lo,hi) =` ________
C. ________________ by ________	
D. ________________ by ________	
E. ________________ by ________	

0.5pts

4.5 In this task, `POW` is the power function as defined in lecture.

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

○ **Always preserved**	○ **Not always preserved**
Assume: ________________	*Counterexample:*
To show: ________________	`e =` ________
Proof:	`accum =` ________
A. ________________ by ________	`x =` ________
B. ________________ by ________	`y =` ________
C. ________________ by ________	
D. ________________ by ________	
E. ________________ by ________	