15-122: Principles of Imperative Computation, Spring 2022

Written Homework 2

Due on Gradescope: Monday 31st January, 2022 by 9pm EST

Name: Greg Budhijanto

Andrew ID: gbudhija

Section: R

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

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

A.	`0 <= a` initially	by line 2
B.	`i = 0` initially	by line 5
C.	`i += 2` on each iteration	by line 8
D.	`a += 1` on each iteration	by line 9
E.	`i <= 2*a`	by (A), (B), (C), and (D)
	Therefore we conclude that	
F.	`i < b`	by (E) and line 2

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

If “No”, the issues with this proof are assumes i==0

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

A.	`!(i < a)`	by line 6
	Therefore we conclude that	
B.	`i >= a`	by math on (A)

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

If “No”, the issues with this proof are

1.5pts

1.2

```
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 1 ***/
9      i++;
10   }
11   //@assert i == a;          /*** Assertion 2 ***/
12   return i;
13 }
```

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

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

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

If "No", the issues with this proof are ____________________

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

A.	`i < a` to enter the loop	by line 5
B.	`i` grows by 1 in the loop	by line 9
	Therefore we conclude that	
C.	`i == a`	by (A) and (B) after the loop

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

If "No", the issues with this proof are Peeking inside loop. Point-to proof not possible with this assertion

0.5pts

1.3

```
1  int absval(int x) {
2    if (x >= 0) return x;
3    return -x;
4  }
5
6  int f(int a, int b, int n)
7  //@requires n >=0;
8  {
9    while (n > 0) {
10     int c = absval(a);
11     //@assert c >= 0;   /*** Assertion 1 ***/
12     a = b - c;
13     b = c;
14     n--;
15   }
16   return a;
17 }
```

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

A. `c = absval(a)`	by line 10
B. `absval` returns non-negative values	by lines 2 and 3
Therefore we conclude that	
C. `c >= 0`	by (A) and (B)

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

If “No”, the issues with this proof are ______________________________

____

____

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

```
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 A ***/
9          a += 2;
10         i += 1;
11     }
12     //@assert i <= a;          /*** Assertion B ***/
13     return i;
14 }
```

Assertion A is: [✓ SUPPORTED]/UNSUPPORTED

A. a>=0 by line 2

B. i>a by line 5

C. a remains unchanged by

by

by

Therefore we [✓ can]/cannot conclude that

D. i>0 by math on A,B,C

Assertion B is: [✓ SUPPORTED]/UNSUPPORTED

A !(i>a) by line 5

by

by

by

by

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

B. i<=a by line 12

& 0 = 0
& 1 = itself
& F

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 u equal to p with its highest 16 bits set to 0 and the remaining bits left unchanged (so that, for example, 0xAB12CE34 becomes 0x0000CE34).

int u = 0x0000FFFF & p ;

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

int v = (0x00FFFF00 & p) | (0x00EEEE00 & p) ;

c. Set w equal to p with its highest 8 bits set to 1 and with its lowest 8 bits set to 0 (so that, for example, 0xAB12CE34 becomes 0xFF12CE00).

int w = 0xFF0000EE & p ;

1pt

2.2 The function no_overflow_add is intended to return true if result of adding three non-negative numbers a, b, and c 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;
}
```

it does not have an upper bound contract
so it will return an error if

1.5pts

2.3 For each of the following statements, determine whether the statement is true or false in C0. If it is true, give a brief but exhaustive explanation of the underlying reason. If it is false, give a counterexample to illustrate why the statement is false.

For every **`int`** `x, y`: if `x < y`, then `x + 1 <= y`.

○ **True** because

[✓ **False**] Counterexample:

For every **`int`** `x`: `x >> 1` is equivalent to `x / 2`.

○ **True** because

[✓ **False**] Counterexample:

For every **`int`** `x, y, z`: `(x + y) * z` is equivalent to `z * y + x * z`.

[✓ **True**] because distributivity and orders of Operations

○ **False** Counterexample:

For every **`int`** `x, y`: `x < y` is equivalent to `x - y < 0`.

○ **True** because

[✓ **False**] Counterexample: $x = -1$, $y = -2$

$(-1) - (-2) == 1 > 0$

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

The Pell sequence is shown below:

$$0, 1, 2, 5, 12, 29, 70, 169, 408, 985, \ldots$$

Each integer i_n in the sequence for $n \geq 3$ is the sum of $2i_{n-1}$ and i_{n-2}. By definition, $i_1 = 0$ and $i_2 = 1$. Consider the following implementation for `fastpell` that returns the n^{th} Pell number, $n \geq 1$, and the specification function `PELL` for it. The body of the loop is not shown.

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

int fastpell(int n)
//@requires n >= 1;
//@ensures \result == PELL(n);
{
  if (n <= 1) return 0;
  if (n == 2) return 1;
  int i = 0;
  int j = 1;                                   k' = 2*j + i
  int k = 2;
  int x = 3;
  while (x < n)
    //@loop_invariant 3 <= x && x <= n;
    //@loop_invariant i == PELL(x-2);
    //@loop_invariant j == PELL(x-1);
    //@loop_invariant k == i + 2*j;
    {
      // LOOP BODY NOT SHOWN: modifies i, j, k, and x
    }
  return k;
}
```

In this problem, we will reason about the correctness of the `fastpell` function when the argument `n` is greater than or equal to 3, and we will complete the implementation based on this reasoning.

(NOTE: To completely reason about the correctness of `fastpell`, we also need to point out that `fastpell(1) == PELL(1)` and that `fastpell(2) == PELL(2)`. 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 20 and

we know `x >= n` by line 19, which implies that `x == n` by logic.

The returned value `\result` is the value of `k` after the loop, so to show that the postcondition on line 11 holds when `n >= 3`, it suffices to show `k == PELL(n)` after the loop.

A `i == PELL(x-2)` by line 21

B `j == PELL(x-1)` by line 22

C `x >= 1` by line 18

D `k == i + 2*j` by line 23

E `== PELL(x-2) + 2*PELL(x-1)` by math on A, B

F `== PELL(x)` by PELL, commutativity of + and C

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 PELL.

The loop invariant `3 <= x` on line 20 holds initially by line(s) 18

The loop invariant `x <= n` on line 20 holds initially by line(s) 10, 14, 1

The loop invariant on line 21 holds initially by line(s) 18, 4 & math

The loop invariant on line 22 holds initially by line(s) 18, 5 & math

The loop invariant on line 23 holds initially by lines 17, 15 and 16.

1 2 3 4 5 6 7 8 9 10

0 1 2 5 12 29 70 169 408 985

i j k

j k

i j k

i k

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 `PELL()`. The specification function is meant to be used in contracts only. Also, do not call `fastpell` recursively, since this isn't fast!**

x n

3<6

4

5

6

(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 3 <= x && x <= n;          3      n
//@loop_invariant i == PELL(x-2);
//@loop_invariant j == PELL(x-1);
//@loop_invariant k == i + 2*j;
{
    i = j                                         ;

    j = k                                         ;

    k = 2*j + i                                   ;

    x = X+1                                       ;
}

return k;
```

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 integer expression (n - x) is strictly decreasing because at the begining of the loop x <= n and each iteration of the loop increments the value of x by 1.

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

a != b

```
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`.

Case 1, `(a > b)`:

A. a != b by line 11
B. a > b by line 14 & case
C. a - b > 0 by math
D. a' = a - b by line 15
E. a' > 0 by A, B, C, D
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

```
1 while (a > 1)
2 //@loop_invariant a >= 1;
3 {
4     if (a % 2 == 0) {   even
5         a = a / 2;
6     } else {        odd
7         a = 3 * a + 1;
8     }
9 }
```

$2/2 = 1$ ✓

[✓ **Always preserved**]

Assume: $a >= 1$ (by line 2) before an iteration

To show: $a' >= 1$ after an iteration

Proof: since $a' = a/2$ or $a' = 3*a+1$, we need to show $a' >= 1$

A. $a > 1$ by line 1

B. if $a\%2 == 0$, $a/2 >= 1$ by line 5, math

C. $a' >= 1$ by A, B, math

D. if $a\%2 \neq 0$, $3*a+1 > 1$ by A

E. $a' > 1$ by A, D, math

○ **Not always preserved**

Counterexample:

0.5pts

4.3

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

$k = 9$ $k' = 4$

$i = -3$ $i' = -2$

[✓ **Always preserved**]

Assume: $i*i == k$ (by line 2) before an iteration

To show: $i'*i' == k'$ after an iteration

Proof:

A. $i' = i+1$, $i = i'-1$ by 5 math

B. $k' = k+2i+1$, $k = k'-2i-1$ by 4 math

C. $k = k' - 2(i'-1) - 1 = k' - 2i' + 1$ by math

D. $i*i = k$, $(i'-1)(i'-1) = k' - 2i' + 1$ by math

E. $i'*i' - 2i' + 1 = k' - 2i' + 1$ by math

F. $i'*i' = k'$ by

○ **Not always preserved**

Counterexample:

$i' = i+1$

$i = i'-1$

$k' = k+2i+1$

$k = k'-2i-1$

0.5pts

4.4 In this task, you know nothing about what $\overset{\text{the}}{\underset{\text{this}}{\mathsf{f}}}$ computes.

```
1 while (a < 700)
2 //@loop_invariant a + b == f(a,b);
3 {
4     int c = f(a,b);
5     a += 1;
6     b = c - a - 1;
7 }
```

c = f(a, b)

a' = a + 1

b' = c + a + 1

○ **Always preserved**

Assume: ______________________

To show: ______________________

Proof:

A. X == 2 * Y, by ______

B. j' = j + y = X/2 by 5, math

C. i' == 4 j' is always true by ______

D. because (i + 2 * X) == 4 * (j + X/2) by ______

E. is the same as i + by ______

☑ **Not always preserved**

Counterexample:
function f cannot be confirmed from the given lines

0.5pts

4.5 In this task, you know nothing about what h computes.

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

i' == 4 * j'

x = 4
y = 2

i' = i + 2 * x
j' = j + y = j + x/2
x' = h(i')

$y = \frac{x}{2}$

$2(j' - j) = x$

$x = \frac{(i' - i)}{2}$

2 (j' - j) = (i' - i)/2

i' - i == 4 x (j' - j)

☑ **Always preserved**

Assume: i == 4 * j by line 2

To show: i' == 4 * j'

Proof:

A. y == X/2 by 1, math

B. i = i' - 2 * x by 4, math

C. j = j' - y = j' - X/2 by 5, math

D. i' - 2 * x = 4 * (j' - x/2) by 2, math

E. i' - 2 * x = 4 j' - 2 * x by math

i' = 4 j' math

○ **Not always preserved**

Counterexample:

x 4
y 2
i 8 i 16
j 2 j 4

i' = i + 2 * X
j' = j + Y

i + 2x = 4j + 2x

$y = \frac{x}{2}$ $j' = j + \frac{x}{2}$