15-122: Principles of Imperative Computation, Spring 2020

Written Homework 2

Due on Gradescope: Monday 27th January, 2020 by 9pm

Name: ____

Andrew ID: ____

Section: ____

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

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.
- ***Preview***, the Mac's PDF viewer.
- ***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*.

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:	2.5	3	3	3.5	12
Score:					

1. **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 an assertion being *supported* is a subtle but important one.

To support an assertion one may use the following facts:

- When local variables are *untouched* by a loop, statements we know to be true about those variables *before* the loop remain valid *inside* the loop and *after* the loop.
- For local variables 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 invariants held just before the loop guard was checked and that the loop guard returned `true`.
  - After a loop, we know that the loop invariants 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 UNSUPPORTED and 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 preserved by any iteration of the loop. If you claim that the assertion is supported, your answer should be a concise proof; if you claim that the assertion is unsupported, we only expect an informal argument to explain why.

- If the assertion is supported, fill in the lines with a relevant fact on the left and a justification for it on the right.
- If the assertion is *not* supported, use the lines to write a short explanation of why it is not supported.

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

We've given one worked out solution below.

```
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 }
```

`Assertion 1` is: SUPPORTED

– `i < a` (by line 5)
– `a < b` (by line 2) and `a` and `b` not changed by loop

Therefore, we **can** conclude that

– `i < b` since `i < a` and `a < b` implies `i < b`

`Assertion 2` is: UNSUPPORTED

– `!(i < a)` (by line 5)
– `i >= a` (by math)

Therefore, we **cannot** conclude that

– `i == a` (does not follow logically from any fact we know)

`Assertion 3` is: SUPPORTED

– `i >=1` (by line 6)

Therefore, we **can** conclude that

– `i != 0` (by math)

0.5pts

1.1

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

`Assertion A` is: UNSUPPORTED

by

by

by

by

by

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

by

`Assertion B` is: SUPPORTED

by

by

by

by

by

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

by

1pt

1.2

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

`Assertion A` is: ______________________________

______________________________ by

______________________________ by

______________________________ by

______________________________ by

______________________________ by

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

______________________________ by

`Assertion B` is: ______________________________

______________________________ by

______________________________ by

______________________________ by

______________________________ by

______________________________ by

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

______________________________ by

1pt

1.3 If relevant, you may assume the functions `POW` is defined as we did in class.

```
int f(int x, int y)
//@requires 0 <= x;
{
  int i = 0;
  int accum = 1;
  while (i < x)
  //@loop_invariant accum == POW(y, i);
  {
    //@assert i <= x;                    /*** Assertion A ***/
    accum = accum * y;
    i = i + 1;
  }
  //@assert accum == POW(y, x);          /*** Assertion B ***/
  return accum;
}
```

`Assertion A` is: ____________________

____________________ by

____________________ by

____________________ by

____________________ by

____________________ by

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

____________________ by

`Assertion B` is: ____________________

____________________ by

____________________ by

____________________ by

____________________ by

____________________ by

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

____________________ 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. 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 ($0 \Longrightarrow 1$ and $1 \Longrightarrow 0$) (so that, for example `0xAB0F1812` becomes `0xABF0E712`).

```
int z = ______________________________________________;
```

0.5pts

2.2 The function `safe_add` is intended to check that the result of adding three numbers `a`, `b`, and `c` is the same in normal integer arithmetic and in C0's 32-bit two's complement signed modular arithmetic.

Does the following code satisfy this specification? If so, state why in one sentence. If not, give positive 32-bit values for `a`, `b`, and `c` *in hexadecimal* such that the check will return an incorrect result. Explain why the result is incorrect in this case.

```
bool safe_add(int a, int b, int c) {
  if (a > 0 && b > 0 && c > 0 && a + b + c < 0) return false;
  if (a < 0 && b < 0 && c < 0 && a + b + c > 0) return false;
  return true;
}
```

1pt **2.3** For each of the following statements, determine whether the statement is true or false in C0. If it is true, explain why in one sentence. 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`.

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

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

For every **`int`** `x, y`: `x < y` is equivalent to `x - y < 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, $n \geq 1$. 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);
}

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;
  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 ☐ and we know `x >= n` by line ☐ , 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.

`k == i + 2*j`	by line ☐
`== i + 2*PELL(x-1)`	by line ☐
`== PELL(x-2) + 2*PELL(x-1)`	by line ☐
`== PELL(x)`	by `PELL` definition, the commutativity of `+`, and `x >= 1` by line ☐

1pt

3.2 Loop invariant holds initially

Complete the argument for the loop invariants holding initially by giving appropriate line numbers.

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

The loop invariant `x <= n` 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 line(s) ☐.

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

0.5pts

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!**

(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;
//@loop_invariant i == PELL(x-2);
//@loop_invariant j == PELL(x-1);
//@loop_invariant k == i + 2*j;
{
    i = ________________________________________;

    j = ________________________________________;

    k = ________________________________________;

    x = ________________________________________;
}

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 quantity ____________ is strictly decreasing because

Since the loop terminates if this quantity reaches 0 or less and this quantity is strictly decreasing, the loop must terminate.

4. **The Preservation of Loop Invariants**

   The core of proving the correctness of a function with one loop is proving that the loop invariant is *preserved* — that if the loop invariant holds at the beginning of an iteration (just before the loop guard is tested), it still holds at the end of that iteration (just before the loop guard is tested the next time).

   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*. When we ask for a counterexample, what we mean is that we want *specific, concrete* values of the local variables such that the loop guard and loop invariant will hold before the loop body executes for some iteration, but where the loop invariant will not hold after the loop body executes that one iteration.

   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!

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

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. We must 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)

0.5pts

4.1

```
while (x < y && x <= 15122)
//@loop_invariant x <= y;
{
    if (0 <= z && z < 10) {
        x = x + z;
    }
}
```

NOT ALWAYS PRESERVED

Counterexample: $x =$ ______ , $y =$ ______ , $z =$ ______ .
The loop invariant and loop guard are satisfied at the start of the iteration but the loop invariant is not satisfied at the end of that iteration.

0.5pts

4.2

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

ALWAYS PRESERVED

The first loop invariant is always preserved because ______________________

____.

For the second loop invariant, we assume that $i \leq y$ and want to show that $i' \leq y'$ (or equivalently $i' \leq y$ since y does not change in the loop).

Using operational reasoning for one iteration:

By line 5, $i' =$ ______ .

By line 2, $x + 1 \leq$ ______ .

By line 1, ______ $\leq x + 1$.

The previous three statements taken together imply that $i' \leq y$.

0.5pts

4.3 In this example, you are using two functions with the following declarations:

```
bool f(int x);
int mid(int lo, int hi)
  /*@requires 0 <= lo && lo < hi; @*/
  /*@ensures lo <= \result && \result < hi; @*/ ;
```

That is, `mid(lo, hi)` takes two integers and returns an integer in the non-empty range `[lo, hi)`. The function `f(x)` takes an integer and returns a boolean; we don't know anything about its return value, so we reason about both cases.

Now consider the following code that uses functions `f` and `mid`:

```
11 while (lo < hi)
12 //@loop_invariant 0 <= lo && lo <= hi;
13 {
14     m = mid(lo, hi);
15     if (f(m)) {
16         lo = m+1;
17     } else {
18         hi = m;
19     }
20 }
```

ALWAYS PRESERVED (Complete the indicated parts of the proof)

Assume: ______________________________

To show: ______________________________

Case 1: `f(m)` returns `true`

By lines 15 and 16, `lo'` = ______________________________

By line 15, `hi'` = ______________________________

Therefore

Case 2: `f(m)` returns `false`

By line 15, `lo'` = ______________________________

By lines 15 and 18, `hi'` = ______________________________

Therefore. . . *(Skip this, as it looks much like the previous case)*

0.5pts

4.4

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

0.5pts

4.5

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

0.5pts

4.6

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

0.5pts

4.7

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