15-122 : Principles of Imperative Computation, Fall 2015

Written Homework 2

Due: **Monday**, September 14, 2015 **by 6PM**

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.

Print out this PDF double-sided, **staple** pages in order, and write your answers on these pages *neatly*.

The assignment is due on **Monday**, September 14, 2015 **by 6PM**.

You can hand in the assignment to your TA during lab or in the box outside of GHC 4117 (in the CS Undergraduate Program suite). WARNING: The box is removed promptly at 6PM.

You must hand in your homework yourself; do not give it to someone else to hand in.

(This page intentionally left blank. Remember to print double-sided!)

Question	Points	Score
1	8	
2	4	
3	6	
4	7	
Total:	25	

1. **Safety in C0**

   We'll talk a lot in this class about proving that contracts (preconditions, postconditions, assertions, and loop invariants) will always evaluate to `true`. This is important, because it's how we prove a function correct.

   Before we can even talk about correctness, though, we want to use our contracts to think about *safety*. There are five kinds of safety violations that we've talked about so far in class:

   - Giving a function call arguments that violate its preconditions;
   - Division or modulo by zero;
   - Bitshifting an integer left or right by less than zero or more than 31;
   - Allocating an array with negative length; and
   - Accessing an array out of bounds.

   Whenever we have an operation that's potentially unsafe, we must be able to point to contracts that ensure its safety *without reasoning about multiple iterations of any loop at once*. That means we may use the following facts:

   - When locals are *untouched* by a loop, statements we know to be true about those locals *before* the loop remain valid *inside* the loop and *after* the loop.
   - For locals 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 the safety of each potentially unsafe operation is SUPPORTED or UNSUPPORTED given the existing contracts, and briefly explain your reasoning**. You can assume that any loop invariants are true initially (before the loop guard is checked the first time) and that they are 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.

   We've given two examples below.

0pts

(a)
```
/*  1 */  void fill(int x, int[] A)
/*  2 */    /*@requires x == \length(A); @*/;
/*  3 */
/*  4 */  int[] f(int x, int[] B)
/*  5 */  //@requires 1 <= x && x < \length(B);
/*  6 */  {
/*  7 */      int i = 1;
/*  8 */      int[] A = alloc_array(int, x);
/*  9 */
/* 10 */      while (i < x)
/* 11 */      //@loop_invariant i >= 1;
/* 12 */      {
/* 13 */          B[i] = B[i] - 3;
/* 14 */          i += 1;
/* 15 */      }
/* 16 */
/* 17 */      fill(i, A);
/* 18 */      return A;
/* 19 */  }
```

Safety of the array access on line 13 is: SUPPORTED.

To show: `0 <= i && i < \length(B)`

- `i >= 1` (line 11)
- `0 <= i` (because `i >= 1`)
- `i < x` (line 10)
- `x < \length(B)` (line 5)
- `i < \length(B)` (because `i < x` and `x < \length(B)`)

Safety of the function call on line 17 is: UNSUPPORTED.

We know by line 8 that `\length(A) == x`, so for safety of the function call we need to know `i == x`.

By line 10, we know that the loop guard `i < x` is false – that is, we know that `!(i < x)`, which is the same thing as saying `i >= x`. We can't conclude, from this, that `i` is equal to `x`.

With a different loop invariant on line 11, safety of the function call on line 17 **would have been supported**. You'll demonstrate this in the next question.

That means that, even though we have a function call whose precondition will never fail, our loop invariants aren't good enough for us conclude (using only logical reasoning based on existing contracts) that that function call is safe!

2pts

(b) This is the **exact same code** from the previous example, but because there is a different loop invariant, the answers, which are given, have changed. You just have to provide explanations.

```
/*  1 */  void fill(int x, int[] A)
/*  2 */    /*@requires x == \length(A); @*/;
/*  3 */
/*  4 */  int[] f(int x, int[] B)
/*  5 */  //@requires 1 <= x && x < \length(B);
/*  6 */  {
/*  7 */      int i = 1;
/*  8 */      int[] A = alloc_array(int, x);
/*  9 */
/* 10 */      while (i < x)
/* 11 */      //@loop_invariant i <= \length(A);
/* 12 */      {
/* 13 */          B[i] = B[i] - 3;
/* 14 */          i += 1;
/* 15 */      }
/* 16 */
/* 17 */      fill(i, A);
/* 18 */      return A;
/* 19 */  }
```

Safety of the array access on line 13 is: UNSUPPORTED.

Safety of the function call on line 17 is: SUPPORTED.

3pts

(c)
```
/*  1 */  int gap(int x, int y)
/*  2 */    /*@requires x <= y; @*/ ;
/*  3 */
/*  4 */  int g(int a)
/*  5 */  //@requires 0 <= a;
/*  6 */  {
/*  7 */      int i = 2*a;
/*  8 */
/*  9 */      while (i > a)
/* 10 */      //@loop_invariant i >= a;
/* 11 */      {
/* 12 */          int[] A = alloc_array(int, i-1);
/* 13 */          a += 2;
/* 14 */          i += 1;
/* 15 */      }
/* 16 */
/* 17 */      return gap(i, a);
/* 18 */  }
```

Safety of the array allocation on line 12 is: _____

Safety of the function call on line 17 is: _____

3pts

(d)
```
/*  1 */  int h(int n) {
/*  2 */      int x = 1;
/*  3 */      while (n >= 25) {
/*  4 */          x += 1;
/*  5 */          if (n % 2 == 1) {
/*  6 */              n = 3*n + 1;
/*  7 */          } else {
/*  8 */              n = n/2;
/*  9 */          }
/* 10 */      }
/* 11 */
/* 12 */      if (n >= 0) {
/* 13 */          return x << n;
/* 14 */      }
/* 15 */
/* 16 */      return 1000000 / x;
/* 17 */  }
```

Safety of the left-shift on line 13 is: _____

Safety of the division on line 16 is: _____

2. **Basics of C0: the `int` and `bool` data types**

2pts (a) 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.

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

```
int x =
```

ii. Set `y` equal to `p` with its highest and lowest 16 bits swapped (so that `0x1234ABCD` becomes `0xABCD1234`)

```
int y =
```

iii. Set `z` equal to `p` with its middle 16 bits flipped ($0 \Longrightarrow 1$ and $1 \Longrightarrow 0$) (so that `0xAB0F1812` becomes `0xABF0E712`).

```
int z =
```

The function `safe_add` is intended to check that the result of adding the three numbers `a`, `b`, and `c` has the same result in normal integer arithmetic and in 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 32-bit values for a, b and c *in hexadecimal* such that the check will return an incorrect result:

```
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;
}
```

2pts

(b) 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 functions with one loop correct**

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. The body of the loop is not shown.

```
/*   1 */  int FIB(int n)
/*   2 */  //@requires n >= 0;
/*   3 */  {
/*   4 */      if (n <= 1) return n;
/*   5 */      else return FIB(n-1) + FIB(n-2);
/*   6 */  }
/*   7 */
/*   8 */  int fast_fib(int n)
/*   9 */  //@requires n >= 0;
/*  10 */  //@ensures \result == FIB(n);
/*  11 */  {
/*  12 */      if (n <= 1) return n;
/*  13 */      int a = 0;
/*  14 */      int b = 1;
/*  15 */      int c = 1;
/*  16 */      int x = 2;
/*  17 */
/*  18 */      while (x < n)
/*  19 */      //@loop_invariant 2 <= x && x <= n;
/*  20 */      //@loop_invariant a == FIB(x-2);
/*  21 */      //@loop_invariant b == FIB(x-1);
/*  22 */      //@loop_invariant c == a + b;
/*  23 */      {
                  // LOOP BODY NOT SHOWN: modifies a, b, c, and x
/* y-3 */      }
/* y-2 */
/* y-1 */      return c;
/*   y */  }
```

In this problem, we will reason about the correctness of the `fast_fib` 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 `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 logical/mathematical reasoning plays a large role. In the future, you may be asked to write an entire solution in a clear, concise manner, and this solution gives you an example of how you might write such a solution.

2pts (a) **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 logical reasoning.

We know `x <= n` by line [ ] and we know `x >= n` by line [ ], which implies that `x == n`.

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.

```
c == a + b                  by line [ ]
  == a + FIB(x-1)           by line [ ]
  == FIB(x-2) + FIB(x-1)    by line [ ]
  == FIB(x)                 by FIB definition and x >= 0 by line [ ]
```

2pts (b) **Loop invariant holds initially**

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

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 (c) **Loop invariant 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 Question 4.)

Solution:

```
/*  18 */      while (x < n)
/*  19 */      //@loop_invariant 2 <= x && x <= n;
/*  20 */      //@loop_invariant a == FIB(x-2);
/*  21 */      //@loop_invariant b == FIB(x-1);
/*  22 */      //@loop_invariant c == a + b;
/*  23 */      {
/*  24 */          x = ________________________________________;
/*  25 */
/*  26 */          a = ________________________________________;
/*  27 */
/*  28 */          b = ________________________________________;
/*  29 */
/*  30 */          c = ________________________________________;
/*  31 */      }
/*  32 */
/*  33 */      return c;
/*  34 */     }
```

1pt (d) **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 part c.

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 a 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 this. 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.

We give 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!

0pts (a)

```
/*  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.

0pts (b)

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

Solution: ALWAYS PRESERVED.

Assume `x < y + 1` (line 2) before an iteration. We must show `x' < y + 1` after an iteration.

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

- `x + 1 < y` (line 1)
- `x + 2 <= y` (because `x + 1 < y`)
- `y < y + 1` (line 2 lets us know `y != int_max()`)
- `x + 2 < y + 1` (because `x + 2 <= y` and `y < y + 1`)

1pt (c)

```
/*  1 */  while (x < y && x <= 15122)
/*  2 */  //@loop_invariant x <= y;
/*  3 */  {
/*  4 */      if (0 <= z && z < 10) {
/*  5 */          x = x + z;
/*  6 */      }
/*  7 */  }
```

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.

1pt (d)

```
/*  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 true 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 1, [ ] $\leq x + 1$.

By line 2, $x + 1 \leq$ [ ] .

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

2pts (e) In this example, you are using two functions with the following declarations:

```
/*  1 */  bool f(int x);
/*  2 */  int mid(int lower, int upper)
/*  3 */    /*@requires 0 <= lower && lower < upper; @*/
/*  4 */    /*@ensures lower <= \result && \result < upper; @*/ ;
```

That is, `mid(lower, upper)` takes two integers and returns an integer in the non-empty range `[lower, upper)`. The function `f(x)` takes an integer and returns a bool; 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 (lower < upper)
/* 12 */  //@loop_invariant 0 <= lower && lower <= upper;
/* 13 */  {
/* 14 */      m = mid(lower, upper);
/* 15 */      if (f(m)) {
/* 16 */          lower = m+1;
/* 17 */      } else {
/* 18 */          upper = 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, $lower' =$ _____

By line 15, $upper' =$ _____

Therefore...

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

By line 15, $lower' =$ _____

By lines 15 and 18, $upper' =$ _____

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

1pt (f)

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

1pt (g)

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

1pt

(h)
```
/*  1 */  while (0 <= b && b < a)
/*  2 */  //@loop_invariant a % 17 == 0 && b % 17 == 0;
/*  3 */  {
/*  4 */      a = a - b;
/*  5 */  }
```