15-122 : Principles of Imperative Computation, Fall 2014

Written Homework 2

Due in class: Thursday, January 30

Name: ____________________

Andrew ID: ____________________

Recitation: ____________________

The written portion of this week's homework will give you some practice working with the binary representation of integers and reasoning with invariants. You are strongly advised to review the C0 language reference guide (available at http://c0.typesafety.net/) for details on integer manipulation.

Question	Points	Score
1	9	
2	4	
3	7	
Total:	20	

You must do this assignment in one of two ways and bring the stapled printout to the handin box on Thursday:

1) Write your answers *neatly* on this PDF.

2) Use the TeX template at `http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15122-s14/www/theory2.tgz`

1. **Basics of C0**

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

i. Using only shift operators, Set a equal to x, where the blue component has been set to 0, with the alpha, red and green components left unchanged. Assume that x is represented in ARGB (eg `0xAB12CE34` becomes `0xAB12CE00`; see Section 1.1 of the Programming portion for more info)

Solution:

ii. Set b equal to x with the highest 8 bits copied into the lowest 8 bits. (eg `0xAB0F1812` becomes `0xAB0F18AB`)

Solution:

iii. Set c equal to x with every fourth bit starting from the lowest bit flipped. ($0 \Rightarrow 1$ and $1 \Rightarrow 0$) (eg `0xAB12CE34` becomes `0xBA03DF25`)

Solution:

(2) (b) We want to check while executing `int x = a*b;`, multiplying two `int`'s `a` and `b`, if an overflow has occurred. In each of the following examples, state whether the condition correctly evaluates to `true` iff there is overflow. If not, give a counterexample.

i.
```
((a > 0) && (b > 0) && (x < 0) || (a > 0) && (b < 0) && (x > 0) ||
 (a < 0) && (b > 0) && (x > 0) || (a < 0) && (b < 0) && (x < 0))
```

Solution:

ii. `(a != 0) && (x / a ! = b)`

Solution:

(4) (c) Two expressions are equivalent in C0, 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 to illustrate why.

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

Solution:

ii. For every `int` x and y, $x < 0$ is equivalent to $-x > 0$

Solution:

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

Solution:

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

Solution:

2. **Termination**

Recall that the standard way to prove termination of loops is by showing that some bounded quantity moves towards its bound on every iteration of the loop. Prove the termination of the following programs, by providing a quantity with a bound and showing that every iteration moves the value closer to its bound. If the program does not terminate, then provide inputs on which the program does not terminate. You can assume that integers do not overflow for the purposes of this question.

(2) (a)
```
/*  1 */  int move(int x, int y)
/*  2 */  {
/*  3 */    while (x < 50 || y < 50)
/*  4 */    {
/*  5 */      if (y > x)
/*  6 */        x++;
/*  7 */      else
/*  8 */        y++;
/*  9 */    }
/* 10 */  }
```

Solution:

(2) (b)
```
/*  1 */  int move(int x, int y)
/*  2 */  {
/*  3 */    while (y < 50)
/*  4 */    {
/*  5 */      if (y > x)
/*  6 */        x = 2*x;
/*  7 */      else
/*  8 */        y++;
/*  9 */    }
/* 10 */  }
```

Solution:

3. **Reasoning with Invariants**

The Pell sequence is shown below:

```
0, 1, 2, 5, 12, 29, 70, 169, 408, 985, ...
```

Each integer i_n in the sequence is the sum of $2i_{n-1}$ and i_{n-2}. Consider the following implementation for `fastpell` that returns the n^{th} Pell number (the body of the loop is not shown).

```
/*  1 */  int PELL(int n)
/*  2 */  //@requires n >= 1;
/*  3 */  {
/*  4 */    if (n <= 1) return 0;
/*  5 */    else if (n == 2) return 1;
/*  6 */    else return 2 * PELL(n-1) + PELL(n-2);
/*  7 */  }
/*  8 */
/*  9 */  int fastpell(int n)
/* 10 */  //@requires n >= 1;
/* 11 */  //@ensures \result == PELL(n);
/* 12 */  {
/* 13 */    if (n <= 1) return 0;
/* 14 */    if (n == 2) return 1;
/* 15 */    int i = 1;
/* 16 */    int j = 0;
/* 17 */    int k = 2;
/* 18 */    int x = 3;
/* 19 */    while (x < n)
/* 20 */      //@loop_invariant 3 <= x && x <= n;
/* 21 */      //@loop_invariant i == PELL(x-1);
/* 22 */      //@loop_invariant j == PELL(x-2);
/* 23 */      //@loop_invariant k == 2*i+j;
/* 24 */      {
/* 25 */        // LOOP BODY NOT SHOWN
/* 26 */      }
/* 27 */    return k;
/* 28 */  }
```

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.

To completely reason about the correctness of `fastpell`, also need to point out that `fastpell(1) == PELL(1)` and that `fastpell(2) == PELL(2)`. This is straightforward, because no loops are involved.

(2) (a) Show that each loop invariant is true just before the loop condition is tested for the first time, using the precondition and any initialization before the loop condition.

Solution:

- `3 <= x && x <= n` – We know `x` is `3` by line ☐, so `3 <= x` is true because `3 <= 3`.

  Because `x` is `3`, to show `x <= n` we just need to show `3 <= n` before the loop condition is tested for the first time. We know `n` is greater than 1 by line ☐ and we know `n` is not 2 by line ☐, so it follows that `n` is greater than or equal to 3.

- `i == PELL(x-1)` – Because `x` is `3` before the loop condition is tested for the first time, `PELL(x-1)` is ☐ and therefore this loop invariant is initially justified by line ☐.

- `j == PELL(x-2)` – Because `x` is `3` before the loop condition is tested for the first time, `PELL(x-2)` is ☐ and therefore this loop invariant is initially justified by line ☐.

- `k == 2*i+j` – Justified by lines ☐.

(2) (b) Show that the loop invariants and the negated loop guard at termination imply the postcondition.

Solution:

We know `x <= n` by line [ ], and we know `x >= n` by line [ ], so this implies that `x` equals `n`.
The result value is the value of `k` after the loop, so to show that that the postcondition holds when `n >= 3`, it suffices to show that, after the loop, `k` equals `PELL(x)`.

- `k` = `2*i + j` (line [ ])
- `2*i + j` = `2*PELL(x-1) + j` (line [ ])
- `2*PELL(x-1) + j` = `2*PELL(x-1) + PELL(x-2)` (line [ ])
- `2*PELL(x-1) + PELL(x-2)` = `PELL(x)` (`PELL` def., `x >= 3` by line [ ])
- `k` = `PELL(x)` (transitivity, the four preceding facts)

(2) (c) Based on the given loop invariant, write the body of the loop. *DO NOT* use the specification function `PELL()`.

Solution:

```
{
  j =

  i =

  k =

  x =
}
```

(1) (d) Explain why the function must terminate with the loop you gave in 2(c).

Solution: