15-122 : Principles of Imperative Computation, Summer 1 2014

Written Homework 2

Due: Thursday, May 29 before recitation

Name: _____

Andrew ID: _____

Recitation: _____

The written portion of this week's homework has two parts. The first part will give you some practice working with the binary representation of integers (you may wish to review details of integer manipulation at http://c0.typesafety.net/) and reasoning with invariants. In the second part, you will work on specifying and implementing search in an array.

Question	Points	Score
1	7	
2	6	
3	7	
4	5	
5	4	
6	4	
7	2	
Total:	35	

You *must* print this PDF and write your answers *neatly* by hand. You can hand in the assignment in class before the lecture starts.

Part I: Invari-ints

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 flipped (0 becomes 1 and 1 becomes 0), starting from the lowest bit. (eg `0xAB12CE34` becomes `0xBA03DF25`)

Solution:

(4) (b) 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: $x < 0$ is equivalent to $-x > 0$

Solution:

iii. For every `int` x, y and z where $y \neq 0$: $x * y < z$ 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 (specific values for `x` and `y`) on which the program does not terminate. You should assume that **integers do not overflow**. An example is given below.

```
/* 1 */  int move(int x, int y)
/* 2 */  {
/* 3 */    while (x > 10)
/* 4 */    {
/* 5 */      x--;
/* 6 */      y++;
/* 7 */    }
/* 8 */    return x;
/* 9 */  }
```

Example Solution: The quantity `x` is lower bounded by 10 and decreases by 1 on each iteration.

(3) (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 */    return x;
/* 11 */  }
```

Solution:

(3) (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 */    }
/* 11 */    return x;
/* 12 */  }
```

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$^{\text{th}}$ 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 */    if (n == 2) return 1;
/*  6 */    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:

Part II: Searching for Specifications

For the remainder of the homework, we will be using and referring to functions from `arrayutil.c0`. This file is available on Piazza.

4. **Preconditions and Postconditions**

Here is our initial, buggy specification of search for the first occurrence of `x` in an array.

```
/*  1 */   int search(int x, int[] A, int n)
/*  2 */   //@requires 0 <= n && n <= \length(A);
/*  3 */   /*@ensures (\result == -1 && !is_in(x, A, 0, n))
/*  4 */            || (0 <= \result && \result < n
/*  5 */                && A[\result] == x
/*  6 */                && A[\result-1] < x); @*/
```

(1) (a) Give values of `A` and `\result` such that the precondition evaluates to true and checking the postcondition will cause an array-out-of-bounds exception.

Solution:

- `x = 42`
- `A =` 
- `n = 5`
- `\result =`

(1) (b) Notice that line `6` seems to be relying on `A` being sorted. It might be possible, then, that unsorted input will reveal additional bugs in our initial specification.

Give values for `A` and `\result` such that the precondition and the postcondition both evaluate to true, but `\result` is *not* the index of the first occurrence of `x` in the array. You are not allowed to choose `\result == -1`.

Solution:

- `x = 42`
- `A =` 
- `n = 5`
- `\result =`

(1) (c) Give values for `A` and `\result` such that the precondition evaluates to true, the postcondition evaluates to *false*, and `\result` *is* the index of the first occurrence of `x` in the array.

Solution:

- `x = 42`
- `A =` 
- `n = 5`
- `\result =`

(1) (d) Edit line `6` slightly so that, if we added an additional precondition

```
//@requires is_sorted(A, 0, n);
```

the postcondition for `search` would be safe and would correctly enforce that `A[\result]` is the first occurrence of `x` in `A`. Do *not* use any of the `arrayutil.c0` specification functions.

Solution:

```
/* 6 */
```

(1) (e) Edit line `6` so that *whether or not* we require that the array is sorted, the postcondition for `search` is safe and correct. You'll need to use one of the `arrayutil.c0` specification functions.

Solution:

```
/* 6 */
```

5. **The Loop Invariant**

Now we will consider a buggy implementation with a correct specification.

```
/*   1 */   int search(int x, int[] A, int n)
/*   2 */   //@requires 0 <= n && n <= \length(A);
/*   3 */   //@requires is_sorted(A, 0, n);
/*   4 */   /*@ensures (\result == -1 && !is_in(x, A, 0, n))
/*   5 */            || (0 <= \result && \result < n
/*   6 */                && A[\result] == x
/*   7 */                /* YOUR ANSWER FROM 4(d) */); @*/
/*   8 */   {
/*   9 */     int lower = 0;
/*  10 */     int upper = n;
/*  11 */     while (lower < upper)
/*  12 */     //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
/*  13 */     //@loop_invariant lower == 0 || x  > A[lower - 1];
/*  14 */     //@loop_invariant upper == n || x <= A[upper];
/*  15 */     {

...

/* m-3 */     }
/* m-2 */     //@assert lower == upper;
/* m-1 */     return -1;
/*   m */   }
```

You should assume that the missing loop body does not write to the array `A` or modify the assignable variables `x`, `A`, or `n`.

(1) (a) Prove that the loop invariants (lines 12-14) hold initially:

Solution:

(1) (b) These loop invariants do not imply the postcondition when the function exits on line `m-1`! Give specific values for `A`, `lower`, and `upper` such the precondition evaluates to true, the loop guard evaluates to false, the loop invariants evaluate to true, and the postcondition evaluates to false, given that `\result == -1`.

Solution:

- `x = 42`
- `A =` 
- `n = 5`
- `\result = -1`
- `lower =`
- `upper =`

(2) (c) Modify the code *after* the loop so that, if the loop terminates, the postcondition will always be true.

Take care to ensure that any array access you make is safe! You know that the loop invariants on lines `12-14` are true, and you know that the loop guard is false (which, together with the first loop invariant on line `12`, justifies the assertion `lower == upper`).

Solution:

```
   /* Loop ends here... */
   //@assert lower == upper;



   if (_____________________________________________) {



       return _____________________________________;
   }
   return -1;
}
```

6. **Binary Search**

Now we'll fill in the loop body with code that does binary search (on a sorted array, of course!).

```
/*   1 */   int search(int x, int[] A, int n)
/*   2 */   //@requires 0 <= n && n <= \length(A);
/*   3 */   //@requires is_sorted(A, 0, n);
/*   4 */   /*@ensures (\result == -1 && !is_in(x, A, 0, n))
/*   5 */           || (0 <= \result && \result < n
/*   6 */               && A[\result] == x
/*   7 */               /* YOUR ANSWER FROM 4(d) */); @*/
/*   8 */   {
/*   9 */     int lower = 0;
/*  10 */     int upper = n;
/*  11 */     while (lower < upper)
/*  12 */     //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
/*  13 */     //@loop_invariant lower == 0 || x  > A[lower - 1];
/*  14 */     //@loop_invariant upper == n || x <= A[upper];
/*  15 */     {
/*  16 */       int mid = lower + (upper-lower)/2;
/*  17 */       if (A[lower] == x) return lower;
/*  18 */       if (A[mid] < x) lower = mid+1;
/*  19 */       else { /*@assert(A[mid] >= x); @*/
/*  20 */         upper = mid;
/*  21 */       }
/*  22 */     }
/*  23 */     //@assert lower == upper;
/*  24 */     return -1;
/*  25 */   }
```

(2) (a) Prove that this loop has to terminate. (What quantity gets smaller every time the loop body runs and is guaranteed to be positive?)

Solution:

(2) (b) Prove that, in the case that the code returns on line `17` , the postcondition on lines `4-7` – with your modification from 4(d) – always evaluates to true.

Solution: When we start the loop, we know the following:

- `0 <= n && n <= \length(A)` by the function's precondition (line `2`, `A` and `n` are never modified by the function)
- `A[0,n)` SORTED by the function's precondition (line `3`, `A` and `n` are never modified by the function and `A` is not written to anywhere)
- `lower < upper` by the loop guard (line `11`)
- `0 <= lower && lower <= upper && upper <= n` by the first loop invariant (line `12`)
- `lower == 0 || x > A[lower-1]` by the second loop invariant (line `13`)
- `upper == n || x <= A[upper]` by the third loop invariant (line `14`)

7. **Code Revisions**

Here's an alternate loop body that performs linear search. You can use it as a replacement for lines 15-22 on page 6:

```
/*  15 */     {
/*  16 */       if (A[lower] == x)
/*  17 */         return lower;
/*  18 */       if (A[lower] > x)
/*  19 */         return -1;
/*  20 */       //@assert A[lower] < x;
/*  21 */       lower = lower + 1;
/*  22 */     }
```

(1) (a) The loop invariants in line 12-14 are still preserved by the new loop body. Prove that the invariant in line 14 is still preserved by the new loop body.

Solution:

(1) (b) You might have noticed in the previous part that `upper` does not change during the loop. So now, complete this simpler loop invariant for the modified code by writing a line that tells you something about `upper`. The resulting loop invariant should be true initially, should be preserved by any iteration of the loop, and should allow you to prove the postcondition *without* the modifications you made in 5(c). (You don't have to write the proof.)

Solution:

```
/*  12 */     //@loop_invariant 0 <= lower && lower <= upper;

/*  13 */     //@loop_invariant lower == 0 || x  > A[lower - 1];

/*  14 */     //@loop_invariant
```