15-122: Principles of Imperative Computation

QuickCheck 3 Solutions

Today, we'll do the QuickCheck at the beginning of recitation. You will have ten minutes to do this. Your TA will go over answers at the end. Then follow your TA's instructions on whether or not to hand it in

Name:

Andrew ID:

Section (circle one): A B C D E F G H

Simplicio and *Sagredo* (you) are a couple of 122 students. They are trying to complete an exercise, but are a bit stuck. Help them out!

Simplicio is trying to recollect the binary search function taught in lecture yesterday, so that he can understand the details that *Salviati* is going to go over in recitation today. He's got most of the outline, but is missing out on some critical points.

Fill in the blanks and show him that you (*Sagredo*) are awesome at algorithms like this one.

```
int binsearch(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (-1 == \result && !is_in(x, A, 0, n))
      || ((0 <= \result && \result < n) && A[\result ] == x);
 @*/
{
   int lower = 0;
   int upper = n;
   while (lower < upper)
   //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
   //@loop_invariant lower == 0 || A[lower-1] < x;
   //@loop_invariant upper == n || A[upper] > x;
   {
      int mid = lower + (upper-lower)/2;
      if (A[mid] < x) {
         // We can ignore the bottom half of the array now, since we
         // know that every thing in that half must be less than x
         lower = mid+1;
      } else if (A[mid] > x) {
         // We can ignore the upper half of the array, since we know
         // that everything in that half must be greater than x
         upper = mid;
```

```
    } else {
      //@assert A[mid] == x;
       return mid;
    }
  }
  //@assert lower == upper;
  return −1;
}
```

Simplicio wrote two functions, `mult` and `main`, and included contracts to test their correctness, as part of his homework:

```
int mult(int x, int y)
//@requires x >= 0 && y >= 0;
//@ensures \result == x*y;
{
 int k = x;
 int n = y;
 int res = 0;
 while (n != 0)
 //@loop_invariant x * y == k * n + res;
 {
   if ((k & 1) == 1) res = res + n;
   k = k >> 1;
   n = n << 1;
 }
 return res;
}

int main()
{
 int a;
 a = mult(3,4);
 return a;
}
```

Unfortunately, *Simplicio* lost his laptop with his only implementation of C0 contract checking. Homework is due soon and he desperately needs your help to ensure that his contracts are dynamically checked. He already wrote a function to test boolean expressions, `void CHECK(bool b)`, and he marked the places in his code where he thinks it might be useful to call `CHECK` (see next page).

Help *Simplicio* out by adding explicit contract checking to his program. In the solution box, write a call to the `CHECK` function next to the number corresponding to a designated point in *Simplicio's* program. For instance, writing `CHECK(B)` next to `/* 1 */` denotes that the boolean expression `B` should be checked at position `1` in the program. Note that since contract checking is expensive, you should only call `CHECK` when needed. Some positions may be left blank.

```
1  int mult(int x, int y) {
2
3   /* 1 */
4   int k = x; int n = y;
5   int res = 0;
6
7   /* 2 */
8   while (n != 0) {
9      /* 3 */
10     if ((k & 1) == 1) res = res + n;
11     k = k >> 1;
12     n = n << 1;
13     /* 4 */
14   }
15  /* 5 */
16  /* 6 */
17  return res;
18  /* 7 */
19 }
20
21 int main() {
22  int a;
23
24  /* 8 */
25  a = mult(3,4);
26
27  /* 9 */
28  return a;
29 }
```

```
/* 1 */ CHECK((x >= 0) && (y >= 0));
/* 2 */ CHECK(x * y == (k * n + res));
/* 3 */  // do nothing here!
/* 4 */ CHECK(x * y == (k * n + res));
/* 5 */  // do nothing here!
/* 6 */ CHECK(res == x * y);
/* 7 */  // do nothing here!
/* 8 */  // do nothing here
/* 9 */  // do nothing here
```

Note: it is not wrong, in terms of the relative order of operations, to have the precondition check on line 7 and the postcondition check on line 8, but it makes more sense to leave these checks associated with the function `mult`.

Obviously lines 5 and 6 are in the same place in the code, so writing the postcondition in either of these places is fine. But we don't actually check the loop invariant after the loop, we check it *before* the loop guard succeeds or fails (which we can achieve by repeating the loop invariant check on lines 2 and 4).