15-122: Principles of Imperative Computation

Recitation 4 Solutions — Josh Zimmerman

Overflow

Overflow can cause potentially nasty bugs, so it's often advisable to check that your operations will not cause overflow before performing them.

For example, let's add 7 + 1 using two's complement arithmetic:

```
  0111
+ 0001
------
  1000
```

But $1000 = -1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0 = -8$. So 1 + 7 = -8 in four-bit two's complement arithmetic, which is not what we wanted to happen. (Similarly, -8 - 1 does strange things—you should play with that and see what happens.)

There's some more weird behavior when we negate the minimum int (In a 4-bit system, that's 1000.) Recall that we define the negation of an integer `x` as `-x =  x+1`. We then have the following:

```
-x = (˜x+1) = (˜1000 + 1) = (0111 + 1) = (1000) = x
```

This is an important oddity to be aware of: **if you negate the minimum integer, you get the minimum integer back.**

Overflow can occur with unsigned integers as well. Let's add 15+1 assuming that both are unsigned integers:

```
  1111
+ 0001
------
  0000
```

Overflow is not a theoretical problem. Here are some real-world examples where integer overflow has led to bugs:

(a) In the original Pac-Man game, the level counter was an unsigned 8-bit integer and could only store values from 0-255 (in this case, the integers were unsigned types, not signed types that we used in the examples above). Reaching level 256 would cause the counter to overflow, leading to a 'kill screen'

(b) In the original Donkey Kong game, the amount of time to complete the level was directly dependent on the level value and was calculated via a formula 100 x (10 x (level + 4)). Reaching level 22 would yield the calculation (100 x 260); because of integer overflow, this would be evaluated as (100 x 4), leading to an impossibly short amount of time to complete the level.

So, clearly, overflow is bad. However, it is also a non-trivial problem to catch. Let's discuss the process of implementing a function `int safe_add(int a, int b)` that will use a precondition to ensure that the calculation $a + b$ does not overflow. Let's discuss the possible cases:

(a) `a > 0 && b > 0`: While it tempting to calculate the sum `a+b` and check for overflow via the expression `a+b > 0`, this solution does not generalize well (i.e. actually perform the computation that we want to check for overflow) although it will work in c0. A better method is to check if `a <= int_max() - b`, as `int_max() - b` will not overflow (as `b > 0`).

(b) `a < 0 && b < 0`: Likewise, the expression `a + b < 0` is not a good approach. We instead check that `a >= int_min() - b`, as `int_min() - b` will not overflow (as `b < 0`)

(c) `a <= 0 && b >= 0`: In this case, `a+b` will not overflow (it will be between `int_min()` and `int_max()`)

(d) `a >= 0 && b <= 0`: In this case, `a+b` will not overflow (it will be between `int_min()` and `int_max()`)

Checkpoint 0

Based on the discussion above, what is the pre-condition for the function `safe_add`?

Solution:

```
int safe_add(int a, int b)
/*@requires (a > 0 && b > 0 && a <= int_max() - b)
  @ || (a < 0 && b < 0 && a >= int_min() - b)
  @ || (a <= 0 && b >= 0) || (a >= 0 && b <= 0)
  @*/
{
  return a + b;
}
```

Checkpoint 1

Try to develop a precondition for the function `safe_mult`. For the sake of simplicity, assume that `a > 0` and `b > 0` (you can try the other cases as an exercise later).
Here's a hint: if `a > 0 && b > 0`, is it true that if `a * b > 0` that overflow did not happen? This should also make it apparent why we adopted the strategy used in `safe_add`.

Solution:

```
int safe_mult(int a, int b)
/*@requires a > 0 && b > 0 && a <= int_max()/b
 @*/
{
  return a * b;
}
```

Arrays

Arrays are stored as *addresses* in C0. We'll talk more about pointers later in the course, but for now it's sufficient to know that a variable representing an array stores the address of the array (i.e. where it is located in memory) and that indexing into the array allows us to access the value of the element at that index.

Like other variables, array variables can be assigned to other array variables. However, assigning array variables just changes the *reference* that one array variable stores.

```
void addOne(int[] arr, int size)
//@requires size == \length(arr);
{
   // It's important to make sure that all array accesses are in bounds.
   // Ideally, we should be able to prove that they are.
   for (int i = 0; i < size; i++)
    //@loop_invariant 0 <= i && i <= size;
    {
      //arr[i] gets the value at index i in array arr; we can assign
      //arr[i] to another value or we can assign a variable to store
      //the value at arr[i]
      arr[i] += 1;
    }
   // When we take (loop invariant) and not(loop exit condition),
   // we can easily conclude that i == size, which is often useful
}

int main()
{
   int array_size = 2;

    //allocate an int array of array_size elements, with all
    //elements initialized to 0. Only way to create array in c0
   int[] a = alloc_array(int, array_size);

   //create an alias to a; any changes made to elements in a
   // will be reflected in b
   int[] b = a;

   //@assert a[0] == 0;
   //@assert b[0] == 0;

   // We must pass the size of the array to the function because otherwise
   // the function has no way of verifying that all array accesses are
   // in bounds.
   addOne(a, array_size);

   // The function call changes the area in memory that a and b
   // refer to, so it changes both a and b when we access them.
   assert (a[0] == 1 && a[1] == 1);
   assert (b[0] == 1 && b[1] == 1);

   return 0;
}
```

Let's try a variation of `addOne`, but with a twist. We will use the same main function as given above.

```
void addOne(int[] arr, int size)
//@requires size == \length(arr);
{
  arr = alloc_array(int, size);
  for (int i = 0; i < size; i++)
   //@loop_invariant 0 <= i && i <= size;
   {
    arr[i] += 1;
   }
}
```

Although we assigned `arr` to a new array and added one to each element in the new array, the changes are not reflected back in the calling function! The reason for this is that C0 is a *call by value* language. This means that when we pass arguments to a function, *copies* of the arguments are passed. Changes made to function's arguments will *not* be reflected back in the calling function.

Checkpoint 2

Fill in the blanks in the code to show that there are no out of bounds array accesses. You are given the function `slow_fib` that calculates Fibonacci recursively (so it can be used as a reference function)

```
int slow_fib(int n)
//@requires n >= 0;
;
```

:

```
int fib(int n)
//@requires n >= 0;
//@ensures \result == slow_fib(n);
{
   int[] F = alloc_array(int, n);
   if (n > 0) {
      F[0] = 0;
   }
   else {
      return 0;
   }
   if (n > 1) {
      F[1] = 1;
   }
   else {
     return 1;
   }
   for (int i = 2; i < n; i++)
     //@loop_invariant 2 <= i && i <= n;
     /*@loop_invariant F[i – 1] == slow_fib(i – 1)
                   && F[i – 2] == slow_fib(i – 2); @*/
     {
       F[i] = F[i – 1] + F[i – 2];
     }
   return F[n – 1] + F[n – 2];
}
```

Checkpoint 3

Are the invariants strong enough to prove the postcondition? If so, then use the loop invariant to prove the postcondition.

Solution:

Array access

The conditions above are necessary and sufficient to show that there are no out of bounds array accesses. Before we reference `F[0]` or `F[1]`, we check with conditional statements (lines 6 and 12) to make sure the accesses are in bounds.

Then, in the loop, our loop invariant guarantees that `2 <= i`. Thus, when we access `F[i - 2]`, we can be sure that `i - 2 >= 0`, so we won't be attempting to access a negative array element. Further, we know that `i < n` by the loop exit condition and `n == \length (F)`, so accessing `F[i]` can't cause any problems. (Neither can accessing `F[i - 1]`—`i - 1` is between `i - 2` and `i`.)

Then, when we access `F[n - 1]` and `F[n - 2]` on line 24, we know that `n == \length (F)`, and that `n > 1`. Since `n > 1`, `n > n - 2 >= 0`, so accessing `n - 2` is fine. Accessing `F[n - 1]` is okay since `n == \length (F)` and `n - 1` must also be positive.

Showing that the postcondition holds.

For the first loop invariant: We know `i >= 2` initially since it was initialized to 2. We know `i <= n` since if `n` were less than 2, we would already have returned.

`i' == i + 1`. Since `2 <= i`, `2 <= i'` as well (assuming no overflow). Further, by the loop guard, `i < n`. Thus, `i' <= n`.

For the second loop invariant:

If we assume that `slow_fib` follows the mathematical definition of Fibonacci correctly, we can show that the loop invariant holds at the start of the loop: `slow_fib(1) == 1 == F[1]` by line 13 and `slow_fib(0) == 0 == F[0]` by line 7.

Then, we can show that it is preserved. `F[i] == F[i - 1] + F[i - 2]`. By the loop invariant, `F[i - 1] == slow_fib(i - 1)` and `F[i - 2] == slow_fib(i - 2)`, so `F[i] == slow_fib(i)`. Also, `i' = i + 1` (so `i' - 1 == i`). Thus, `slow_fib(i' - 1) == F[i' - 1]`

Further, by the loop invariant, `slow_fib(i - 1) == F[i - 1]`, so `slow_fib(i' - 2) == F[i' - 2]`.

Finally, at the end of the loop, we know `i == n` by the loop invariant and the negated loop exit condition. So, we know that `F[n - 1] + F[n - 2] == slow_fib(n)`. Then we return that quantity, so we know that our postcondition is correct.

Termination

The loop terminates since `i` starts out as a number less than `n` and is incremented by 1 each iteration until it reaches `n`, which must happen since `n` and `i` are finite.