Overflow

Now, as an exercise, try to develop a precondition for the function safe_mult. For the sake of simplicity, let’s just try to develop pre-conditions, assuming that \(a > 0\) and \(b > 0\) (you can try the other cases as an exercise):

Before we begin, let’s try to answer the following question: if \(a > 0 \land b > 0\), is it true that if \(a \times b > 0\) that overflow did not happen? This should make it apparent why we adopted the strategy used in safe_add.

Solution: \(a > 0 \land b > 0\): This time, attempting to compute \(a \times b\) and check for overflow via the expression \(a \times b > 0\) is actually wrong. If \(a\) and \(b\) are sufficiently large, \(a \times b\) may actually be positive (and therefore, \(a \times b > 0\) will not catch the overflow).

The solution is to observe that \(a \leq \text{int\_max}() / b\), as \(\text{int\_max}() / b\) will not overflow as \(b > 0\) and \(0 < \text{int\_max}() / b < \text{int\_max}()\).

This leads to the following pre-condition for the function safe_mult:

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

Fibonacci and Arrays

Here’s a slightly more complicated loop: it’s a function that calculates the \(n\)th Fibonacci number more efficiently than the naive recursive implementation. Assume that we have a function:

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

that calculates Fibonacci recursively (so it can be used as a reference function):

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

Fill in the blanks in the code to show that there are no out of bounds array accesses.

Are the invariants strong enough 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. We have the following:

(a) Before we reference F[0] or F[1], we check with conditional statements (lines 7 and 13) to make sure the accesses are in bounds.

(b) 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.

(c) Further, we know that i < n by the loop exit condition. As length (F) == n (as we allocate F with length n), accessing F[i] won’t lead to an error.

(d) Moreover, as F[i-1] is between F[i-2] and F[i], which are both valid accesses, accessing F[i-1] won’t lead to an error.

(e) Finally, when we access F[n-2] and F[n-1], we won’t have a problem as n >= 2 (if we entered the loop), so n − 2 >= 0, so F[n-2] is a safe access. The same can be said of accessing F[n-1]

Correctness

We will first show that the loop invariants are initially true and that they are preserved for each iteration of the loop:

(1) 2 <= i & i <= n:

(a) Initialization:
   i. i >= 2, since i is initialized to 2
   ii. i <= n, since n >= 2 (by pre-condition and conditional checks before loop body). So, as i = 2 initially, i <= n

(b) Preservation:
i. Assume that at the beginning of an iteration, 2 \leq i \& \& i \leq n. As we enter the loop body, i < n by the loop guard.

ii. The new value of i is i' = i + 1

iii. Clearly, if i \geq 2, then i' = i + 1 \geq 2 Also, if i < n, then i' = i+1 \leq n

(2) F[i-1] = slow_fib(i-1) \& \& F[i-2] = slow_fib(i-2):

(a) **Initialization:**


ii. These values match slow_fib(0) and slow_fib(1) respectively, so the invariant is initially true.

(b) **Preservation:**

i. Assume that at the beginning of an iteration, F[i-1] = slow_fib(i-1) and F[i-2] = slow_fib(i-2). As we enter the loop body, i < n.

ii. We set F[i] = F[i-1] + F[i-2] in the loop body and we increment i to i' = i+1

iii. We have F[i'-1] = F[i+1-1] = F[i] = F[i-1] + F[i-2] = slow_fib(i) = slow_fib(i'-1),

   by definition of the Fibonacci numbers.

iv. Also, F[i'-2] = F[i+1-2] = F[i-1] = slow_fib(i-1) = slow_fib(i'-2), as before.

As both loop invariants are true initially and are preserved, we can use them to show that the post-condition is implied. Before we do that, though, we shall show that loop terminates.

**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. We know that i = n at termination by the negation of the loop guard (i \geq n) and the loop invariant (i \leq n).

**Implication of Post-condition:**

(a) We know by the negated loop guard and the loop invariant 2 \leq i \& \& i \leq n that i = n at termination.

(b) Thus, we can substitute i = n in the second loop invariant, yielding F[n-1] = slow_fib(n-1) \& \& F[n-2] = slow_fib(n-2).

(c) As we return F[n-1] + F[n-2] and by the definition of the Fibonacci numbers, F[n] = F[n-1] + F[n-2] = slow_fib(n). Hence, the post-condition is proven true.