Bit manipulation

Let’s look at some examples of masking so you can get a better idea of how it’s used. First, let’s write a function that, given a pixel in the ARGB format, returns the green and blue components of it. Your solution should use only &.

Solution:

1 typedef int pixel;
2 int greenAndBlue(pixel p)
3 // @ensures 0 <= \result && \result <= 0xffff;
4 {
5    // We only want the lower 16 bits of p, so and the others with 0
6    // to get rid of them
7    return p & 0xffff;
8 }

Now, let’s write a function that gets the alpha and red pixels of a pixel in the ARGB format. Your solution can use any of the bitwise operators, but will not need all of them.

Solution:

1 typedef int pixel;
2 int alphaAndRed(pixel p)
3 // @ensures 0 <= \result && \result <= 0xffff;
4 {
5    // First, we want to put the top 16 bits in the bottom of the number.
6    // Then, we want to get rid of any sign extension that the right shift
7    // caused, so we use a mask to get rid of anything above the bottom 16
8    // bits
9    return (x >> 16) & 0xffff;
10 }

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:

\[
\text{int slow_fib(int n)}
\]

that calculates Fibonacci recursively, and obeys all of the mathematical properties of the Fibonacci sequence. We don’t worry about overflow for now – Fibonacci only uses addition, so we can think of it as being defined in terms of modular arithmetic.)
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;
}
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. 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.

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 25, 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 14 and slow_fib(0) == 0 == F[0] by line 8.

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.