CS/Math 240: Introduction to Discrete Mathematics 2/17/2011

Lecture 9 : Invariants

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

DRAFT

Last time we discussed inductive definitions and proving properties of inductively defined concepts using structural induction. Then we saw another application of inductive proofs, namely proving invariants. Today we continue our study of invariants with a discussion of program correctness.

9.1 Program Correctness

Showing that a program is correct means that it does what it is supposed to do. More formally, our goal is to prove that a program satisfies its *specification*, that is, it correctly realizes the prescribed relationship between inputs and outputs. In other words, for each input, the specification tells us what the program should output as a response.

There are two parts to correctness of a program.

1. *Partial correctness*: If the program ever returns a result, it is the correct result.

2. *Termination*: The program returns.

Today we prove the correctness of the grade school multiplication algorithm.

9.2 Grade School Multiplication Algorithm

Let's start start with the specification. As input, the program receives two positive integers, a and b. As output, it should return their product, i.e., $a \cdot b$.

9.2.1 Binary Representation of Integers

To make the analysis easier, we will work with binary representations of numbers instead of decimal representations. As we will see, this makes the grade school multiplication algorithm easier to describe.

In the usual decimal representation of a number, we represent a $(k+1)$-digit integer n as a sequence of digits between 0 and 9 and write it as $d_k d_{k-1} \dots d_1 d_0$ with $d_i \in \{0, 1, \dots, 9\}$ for $i \in \{0, 1, \dots, k\}$. Another way to think about n is as a sum of powers of 10, that is

$$n = \sum_{i=0}^{k} d_i \cdot 10^i, \quad d_i \in \{0, 1, \dots, 9\}. \tag{9.1}$$

For example, the sum of the form (9.1) corresponding to the integer 14376 is

$$1 \cdot 10^4 + 4 \cdot 10^3 + 3 \cdot 10^2 + 7 \cdot 10 + 6 \cdot 10^0$$

(so $d_4 = 1$, $d_3 = 4$, $d_2 = 3$, $d_1 = 7$, and $d_0 = 6$).

To obtain a description of the form (9.1) from some integer n, start with $n_0 = n$. Take the last digit of n_0, that is, take the remainder after dividing n_0 by 10, set d_0 to that remainder, and then subtract d_0 from n_0. Notice that $n_0 - d_0$ is divisible by 10. Dividing $n_0 - d_0$ by 10 gives us an integer n_1. Next, repeat the process with n_1. Take the last digit of n_1 by finding the remainder after dividing n_1 by 10, make the remainder d_1, subtract d_1 from n_1, divide the difference by 10, and get n_2. Keep repeating this process until you end with $n_{k+1} = 0$.

For example, if we do this with $n = 14376$, we get the values in Table 9.1.

i	n_i	d_i	$n_i - d_i$
0	14376	6	14370
1	1437	7	1430
2	143	3	140
3	14	4	10
4	1	1	0
5	0		

Table 9.1: Obtaining the decimal representation of $n = 14376$.

Of course, for decimal numbers, this is a silly procedure since we can just read off the numbers d_0 through d_k from the decimal representation of n right away; however, it gives us a key insight into how to find the binary representation of n. To get the binary representation of n, we cannot just read the bits b_i off of the decimal representation of n, but we can apply the algorithm we described, except with 2 in place of 10, and with the remainders being either 0 or 1 instead of being one of 0 through 9. In the end, we obtain a representation of n of the form

$$\sum_{i=0}^{l} b_i \cdot 2^i, \quad b_i \in \{0, 1\}, \tag{9.2}$$

and can write the binary representation of n as $b_l b_{l-1} \ldots b_1 b_0$.

Table 9.2 shows how we obtain the binary representation of $n = 75$, 1001011.

i	n_i	b_i	$n_i - b_i$
0	75	1	74
1	37	1	36
2	18	0	18
3	9	1	8
4	4	0	2
5	2	0	1
6	1	1	0
7	0		

Table 9.2: Obtaining the binary representation of $n = 75$. Reading the b_i column from bottom to top gives us the binary representation of n, 1001011.

To give some reasoning behind why the algorithm for obtaining a binary representation works, rewrite (9.2) as

$$b_0 + \sum_{i=1}^{l} b_i 2^i = b_0 + 2\left(\sum_{i=1}^{l} b_i 2^{i-1}\right).$$

We see that b_0 is the remainder after dividing n_0 by 2, and we get n_1 by subtracting the remainder and dividing the difference by 2. Then we continue the process with

$$n_1 = \sum_{i=1}^{l} b_i 2^{i-1} = \sum_{i=0}^{l-1} b_{i+1} 2^i.$$

9.2.2 Description of the Algorithm

Now let's review the grade school multiplication algorithm. We write the two numbers we multiply, a and b, above each other. We multiply a by the last digit of b, and write down the result below b. Then we multiply a by the next to last digit of b, and write down the result on the next line, shifted one digit to the left. In general, when we write down the result of multiplying a by some digit of b, we write down the result so that its last digit is in the same column as the digit of b we used to produce the result. If some digit of b is zero, we simply skip it and don't write anything down. Finally, we add up all the intermediate results we wrote down to get the product $a \cdot b$. We show this for $a = 14376$ and $b = 108$ in Figure 9.1a.

```
   14376
·   2108
--------
  115008
 14376
28752
--------
30304608
```

(a) Using decimal representation

```
   10011
·   1101
--------
   10011
  10011
10011
--------
11110111
```

(b) Using binary representation

```
   10011
·   1101
--------
   10011
  10011
--------
  1011111
10011
--------
11110111
```

(c) Adding immediately

Figure 9.1: Grade school multiplication algorithm.

Using binary representations instead of decimal representations greatly simplifies our rules for multiplication. Our multiplication table goes down from being 10 × 10 to being just 2 × 2. This makes the description of the "college version" of the grade school algorithm very easy. In each step, we look at one bit of b. If the bit is 1, we copy down a so that its last bit lines up with the bit of b we're currently considering. Then we perform binary addition of the intermediate results to get the result. In the example in Figure 9.1b, we multiply $19 \cdot 13$ in binary and get 247.

In order to analyze the algorithm, we should define it more formally. In its current version, we need to keep track of possibly many intermediate results, one for each bit of b. That would make the analysis more complicated. Notice that it doesn't matter whether we first perform all the multiplications and then add together all the intermediate results, or whether we perform an addition step after a multiplication step. An example is shown in Figure 9.1c. This version looks like more work because we have to perform multiple additions, but has the advantage that we only need to keep track of one variable that holds the sum of the intermediate results obtained so far.

Finally, we formalize our algorithm as Algorithm 1. The variable y represents the part of the number b we still need to multiply a with. We multiply x by two in each iteration of the loop so as to simulate lining up the last bit of the intermediate result with the bit of b used in the current multiplication step. The variable p holds our running total from Figure 9.1c. Finally, the notation $\lfloor m \rfloor$ on line 6 means that we round m down to the nearest integer, and we read $\lfloor m \rfloor$ as the "floor of m". Thus, $\lfloor y/2 \rfloor$ means we divide y by two and round down, which corresponds to cutting off the last bit from y. We can dispose of that bit because we've already multiplied a with it and won't need it for anything else.

Algorithm 1: Binary Multiplication Algorithm

```
    Input: $a, b$ - positive integers we want to multiply
    Output: $ab$ - product of $a$ and $b$
(1) $x \leftarrow a$
(2) $y \leftarrow b$
(3) $p \leftarrow 0$
(4) while $y > 0$ do
(5)     if $y$ is odd then $p \leftarrow p + x$
(6)     $y \leftarrow \lfloor y/2 \rfloor$
(7)     $x \leftarrow 2x$
(8) end
(9) return $p$
```

9.2.3 Correctness of the Algorithm

Recall that there are two conditions a correct algorithm must satisfy: the partial correctness condition and the termination condition. Let's see what they are in the case of Algorithm 1.

1. *Partial correctness*: When we reach line 9, $p = ab$.
2. *Termination*: We eventually reach line 9. In other words, the loop on line 4 ends after a finite number of iterations.

It is fairly easy to see what happens on the first three lines. We just initialize our variables. Hence, the crux of all our arguments about Algorithm 1's behavior will be in proving facts about the behavior of the loop on line 4.

We can view our algorithm as a system whose state is represented by the values of the variables x, y, and p. We are trying to prove some facts about the state of the algorithm after each repetition of the loop, that is, after each "time step". This sounds like a setting for the use of invariants, and indeed, we will prove certain *loop invariants* on our way towards a proof that Algorithm 1 works correctly. More formally, a loop invariant is a property that holds at the beginning and after any number of iterations of a loop. A loop invariant usually describes relationships among variables.

9.2.3.1 Partial Correctness

To prove partial correctness, we prove the following loop invariant.

Invariant 9.1. *After n iterations of the loop on line 4, $ab = xy + p$.*

Proof. Let x_n, y_n, and p_n be the values of x, y, and p after n iterations of the loop from line 4, respectively. We show that

$$ab = x_n y_n + p_n \tag{9.3}$$

for every natural number n.

We prove by induction that $(\forall n)P(n)$ where $P(n)$ is "equation (9.3) holds".

For the base case, $P(0)$, there haven't been any iterations of the loop yet, so x_0, y_0, and p_0 have the values from the first three lines of Algorithm 1. Thus, $x_0 = a$, $y_0 = b$, and $p_0 = 0$. We see that $x_0 y_0 + p_0 = ab + 0 = ab$, so the base case is proved.

Now we prove the inductive step $(\forall n)P(n) \Rightarrow P(n+1)$. Since there is an if statement in the loop body, we use a proof by cases.

Case 1: y_n is odd. In this case we execute the body of the if statement, and add x_n to p_n. Since nothing else happens to the value of p in the loop body, we get $p_{n+1} = p_n + x_n$. Next, since y_n is odd, we can write $y_n = 2k + 1$ for some integer k, namely $k = (y_n - 1)/2$. After line 6 executes, we have $y_{n+1} = \lfloor y_n/2 \rfloor = \lfloor (2k+1)/2 \rfloor = \lfloor k + \frac{1}{2} \rfloor = k = (y_n - 1)/2$. Finally, the last line of the loop body doubles the value of x, so $x_{n+1} = 2x_n$.

Now we have found the values of x, y and p at the end of the $(n+1)$st iteration of the loop, so we can verify that (9.3) holds after the loop is complete. We have

$$x_{n+1}y_{n+1} + p_{n+1} = 2x_n \frac{y_n - 1}{2} + p_n + x_n = x_n(y_n - 1) + x_n + p_n = x_n y_n + p_n,$$

and the right-hand side is ab by the induction hypothesis. Hence, if y_n is odd, the invariant is maintained.

Case 2: y_n is even. In this case the if statement body is skipped, so the value of p doesn't change, and we get $p_{n+1} = p_n$. Next, since y_n is even, we can write $y_n = 2k$ for some integer k, namely $k = y_n/2$. After line 6 executes, we have $y_{n+1} = \lfloor \frac{y_n}{2} \rfloor = \lfloor \frac{2k}{2} \rfloor = \lfloor k \rfloor = k = y_n/2$. It follows that $y_{n+1} = y_n/2$. Finally, the last line of the loop body doubles the value of x, so $x_{n+1} = 2x_n$.

Now we have found the values of x, y and p at the end of the $(n+1)$st iteration of the loop, so we can verify that (9.3) holds after the loop is complete. We have

$$x_{n+1}y_{n+1} + p_{n+1} = 2x_n \frac{y_n}{2} + p_n = x_n y_n + p_n,$$

and the right-hand side is ab by the induction hypothesis. It follows that if y_n is even, the invariant is maintained.

This completes the proof of the induction step, and also of our proposition. □

We get out of the loop if $y \leq 0$ at the end of some iteration. If we show that $y = 0$ after the last iteration of the loop, Invariant 9.1 implies that $ab = xy + p = 0 + p = p$, which proves that the value of p after the loop terminates is ab. It suffices to show as another loop invariant that y never becomes negative.

Invariant 9.2. *After n iterations of the loop, $y \geq 0$.*

Proof. The proof goes by induction. Like an an earlier proof, let y_n be the value of y after n iterations of the loop.

We see that $y_0 \geq 0$ because $y_0 = b > 0$, which proves the base case.

Now assume $y_n \geq 0$, and consider the $(n+1)$st iteration of the loop. Inside of the loop body, y only changes on line 6. There, we divide y by 2 and round down the result to get y_{n+1}. Since $y_n \geq 0$, $y_n/2 \geq 0$ as well, and rounding down a non-negative number cannot round down below zero, which proves that $y_{n+1} \geq 0$. Hence, the inductive step is proved, and so is the invariant. □

The loop condition implies that when the loop is over, $y \leq 0$. We just showed that $y \geq 0$ throughout the algorithm. It follows that when the loop terminates, $y = 0$, which means that the algorithm returns $p = ab$. Hence, partial correctness of Algorithm 1 is proved.

Let us offer some intuition behind Invariant 9.1. Again, it takes some ingenuity to come up with this invariant. One way of thinking is that in each iteration of the loop, we "shift" x to the left and y to the right. This corresponds to multiplying x by 2 and dividing y by 2. We cut off the last bit of y in the process, and if the last bit of y was 1, we compensate for this loss by adding x to our partial result p.

9.2.3.2 Termination

To complete our correctness proof for Algorithm 1, we show that the loop on line 4 eventually terminates. For this, we show that the value of y decreases in each iteration of the loop by at least one. Intuitively, this is true because y is a positive integer and we divide it by two in every step and round down, which should decrease its value. Now let's argue formally.

Proposition 9.3. *If $y > 0$, $\lfloor y/2 \rfloor \leq y - 1$.*

Proof. Write $y = 2k + r$ where $r \in \{0, 1\}$ and $k \in \mathbb{N}$. Then $\lfloor y/2 \rfloor = \lfloor (2k + r)/2 \rfloor = \lfloor k + r/2 \rfloor = k$.

To show $\lfloor y/2 \rfloor = k \leq y - 1 = 2k + r - 1$, just observe that the only way this would not hold for some $k \in \mathbb{N}$ and $r \in \{0, 1\}$ is if $k = r = 0$, but in that case we would have $y = 0$, which is a case our proposition doesn't consider. □

Since y is initialized to b at the beginning of the algorithm and decreases by at least one in every iteration of the loop by Proposition 9.3, it follows that after at most b iterations of the loop, y becomes zero, at which point the loop terminates and the algorithm returns.

Now the proof of Algorithm 1's correctness is complete.

9.3 Next Time

Next time we will discuss the algorithm for finding the greatest common divisor of two integers.