CS/Math 240: Introduction to Discrete Mathematics 2/22/2011

Lecture 10 : Program Correctness

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

DRAFT

Last time we started discussing correctness of programs as one application of proofs by induction. Today we give one more example of a program correctness proof, and then start discussing recursion. We can view recursion as a generalization of inductive definitions.

10.1 One More Program Correctness Proof

Recall that in order to prove a program correct, we need to show that the program satisfies two conditions:

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

In order to prove partial correctness, we must know what the output is supposed to be on a given input. This information is provided by the program's *specification*, which is a formal description of the relationship between the program's inputs and outputs.

Last time we proved correctness of the grade school multiplication algorithm. Today we prove correctness of an algorithm for finding the greatest common divisor of two integers.

10.1.1 Greatest Common Divisor

We start by defining what we mean by greatest common divisor.

Definition 10.1. *The* greatest common divisor *of integers a and b, denoted* $\gcd(a, b)$*, is the largest integer d such that d divides both a and b. Furthermore, if c divides both a and b, then c also divides* $\gcd(a, b)$.

Because zero is divisible by every integer, $\gcd(0, 0)$ is undefined. In more generality, $\gcd(a, 0)$ is undefined if $a = 0$, and is a when $a > 0$. To see the latter, note that the largest divisor of a is a itself, and a also divides zero (because every integer does). Similarly, $\gcd(0, b)$ is undefined if $b = 0$, and is b otherwise.

We state and prove three properties of the greatest common divisor. We will use these properties to construct an algorithm for finding greatest common divisors.

Lemma 10.2. *Let $a, b \in \mathbb{N}$ with at least one of a, b nonzero. Then the following three properties hold.*

(i) If $a = b$, $\gcd(a, b) = a = b$.

(ii) If $a < b$, $\gcd(a, b) = \gcd(a, b - a)$.

(iii) If $a > b$, $\gcd(a, b) = \gcd(a - b, a)$.

Proof. When $a = b$, a divides both a and b. Furthermore, no integer greater than a divides a, so $\gcd(a, b) = a = b$. This proves (i).

Now let's argue (ii). We show that d is a divisor of both a and b if and only if d is a divisor of both a and $b - a$. We do so by proving two implications.

First assume that d is a divisor of both a and b. Then there exist integers c_1 and c_2 such that $a = c_1d$ and $b = c_2d$. Therefore, we can write $b - a = (c_2 - c_1)d$, and we see that $b - a$ is a multiple of d, which implies that d divides $b - a$. Since d also divides a by assumption, we have shown that d divides both a and $b - a$.

For the other direction, assume d is a divisor of both a and $b - a$. Then there exist integers c_1 and c_2 such that $a = c_1d$ and $b - a = c_2d$. Therefore, we can write $b = (b - a) + a = (c_2 + c_1)d$, and we see that b is a multiple of d, which implies that d divides b. Since d also divides a by assumption, we have shown that d divides both a and b.

Therefore, the sets $\{d \mid d \text{ divides } a \text{ and } d \text{ divides } b\}$ and $\{d \mid d \text{ divides } a \text{ and } d \text{ divides } b - a\}$ are the same, so they both have the same largest element. This largest element is the greatest common divisor of both a and b, and of a and $b - a$. This completes the proof of (ii).

We would argue (iii) the same way as (ii), except with the roles of a and b switched. This completes the proof of the lemma. □

10.1.2 The GCD Algorithm

The observations proved as Lemma 10.2 are a good starting point for an algorithm that computes the greatest common divisor of two numbers. The basic idea is to use (ii) and (iii) of Lemma 10.2 to decrease the values of x and y used for the computation of the greatest common divisor with the hope that they eventually achieve values for which the greatest common divisor is easy to compute (such as case (i) of Lemma 10.2).

We describe the algorithm formally now, and show the specification together with the algorithm.

Algorithm 1: GCD Algorithm

```
Input: $a, b$ positive integers
Output: $\gcd(a, b)$
(1) $(x, y) \leftarrow (a, b)$
(2) while $x \neq y$ do
(3)     if $x < y$ then $y \leftarrow y - x$
(4)                else $x \leftarrow x - y$
(5) end
(6) return $x$
```

To argue correctness of Algorithm 1, we must show that the algorithm returns the greatest common divisor of its two inputs, a and b, and that it terminates. To that end, we define and prove some loop invariants.

Observe that the loop on line 2 terminates when $x = y$, and in that case $\gcd(x, y) = x$. At the beginning, we have $x = a$ and $y = b$, so $\gcd(x, y) = \gcd(a, b)$. If we show that we maintain $\gcd(a, b) = \gcd(x, y)$ throughout the algorithm, that will give us partial correctness. In fact, it is possible to show this, and we do so now.

Invariant 10.3. *After n iterations of the loop on line 2,* $\gcd(a, b) = \gcd(x, y)$.

Proof. We prove this invariant by induction on the number of iterations of the loop.

Let x_n and y_n be the values of x and y after n iterations of the loop, respectively.

For the base case, we have $x_0 = a$ and $y_0 = b$ from line 1, so $\gcd(x_0, y_0) = \gcd(a, b)$.

To prove the inductive step, assume that $\gcd(x_n, y_n) = \gcd(a, b)$.

If $x_n = y_n$, there isn't going to be another iteration of the loop. In this case we don't need to argue about the values of x and y after the $(n + 1)$st iteration of the loop at all.

Now suppose that $x_n \neq y_n$. Then there will be another iteration of the loop, and we have two cases to consider.

Case 1: $x_n < y_n$. Then $x_{n+1} = x_n$ and $y_{n+1} = y_n - x_n$. We see that

$$\gcd(x_{n+1}, y_{n+1}) = \gcd(x_n, y_n - x_n) = \gcd(x_n, y_n) = \gcd(a, b). \tag{10.1}$$

The second equality in (10.1) follows by part (ii) of Lemma 10.2 and the last equality follows by the induction hypothesis. Therefore, $\gcd(x_{n+1}, y_{n+1}) = \gcd(a, b)$ in this case.

Case 2: $x_n > y_n$. Then $x_{n+1} = x_n - y_n$ and $y_{n+1} = y_n$. We see that

$$\gcd(x_{n+1}, y_{n+1}) = \gcd(x_n - y_n, y_n) = \gcd(x_n, y_n) = \gcd(a, b). \tag{10.2}$$

The second equality in (10.2) follows by part (iii) of Lemma 10.2 and the last equality follows by the induction hypothesis. Therefore, $\gcd(x_{n+1}, y_{n+1}) = \gcd(a, b)$ in this case as well.

This completes the proof of the induction step, and of the invariant. □

To argue partial correctness, we observe that when the loop terminates, $\gcd(x, y) = \gcd(a, b)$ by Invariant 10.3. Next, since the loop terminated, $x = y$, so $\gcd(x, y) = x$. Therefore, $x = \gcd(a, b)$, and we return x, so we return $\gcd(a, b)$ as desired. This completes the proof of partial correctness.

Here we remark that in practice, we would not spell out the argument for Case 2 in the proof of Invariant 10.3 because it is almost the same as the argument for Case 1. In mathematical writing, we would just say that if $x_n > y_n$, an argument similar to the one for Case 1 shows that the invariant is maintained after $n + 1$ iterations of the loop in Case 2 as well.

We actually took this shortcut in the proof of Lemma 10.2 where we omitted the proof of part (iii). Make sure that when you take such a shortcut in mathematical writing, you at least check that the omitted proof goes through, for example by writing it down somewhere else.

We now argue that Algorithm 1 terminates by showing that the loop on line 2 terminates after some number of iterations. We prove an additional loop invariant in order to do so.

Invariant 10.4. *After n iterations of the loop, $x > 0$ and $y > 0$.*

Proof. We prove the invariant by induction. As in the proof of Invariant 10.3, let x_n and y_n be the values of x and y after n iterations of the loop, respectively.

Before the loop starts, $x_0 = a$, $y_0 = b$, and the specification tells us that $a, b > 0$. This proves the base case.

Now assume that $x_n > 0$ and $y_n > 0$ for some n. If $x_n = y_n$, there is not going to be an $(n+1)$st iteration of the loop. If $x_n \neq y_n$, there are two cases. If $x_n < y_n$, $x_{n+1} = x_n > 0$, and we subtract x from y to get $y_{n+1} = y_n - x_n$, which is greater than zero because $y_n > x_n$. Thus, $x_{n+1}, y_{n+1} > 0$ in this case. We can argue similarly in the case $x_n > y_n$, and we see that the invariant is maintained after the $(n + 1)$st iteration of the loop. □

To prove that an algorithm terminates, we often find a quantity, sometimes called a *potential*, that decreases in discrete steps over time, and show that it cannot go below a certain threshold. In our case, the potential is $x + y$, and decreases by either x or y (that's the "discrete step") after each iteration of the loop (i.e., "over time"). We use this quantity to prove termination.

When $x = y$, the algorithm terminates right away, so there is nothing to prove in this case. Now suppose $x \neq y$ after n iterations of the loop. That is, in our earlier notation, $x_n \neq y_n$. Then, depending on which of x_n, y_n is greater, either $x_{n+1} = x_n - y_n$ and $y_{n+1} = y_n$, or or $y_{n+1} = y_n - x_n$ and $x_{n+1} = x_n$. Invariant 10.4 tells us that $x_n, y_n > 0$, so either $x_{n+1} \leq x_n - 1$ or $y_{n+1} \leq y_n - 1$. In either case, $x_{n+1} + y_{n+1} \leq x_n + y_n - 1$, so $x + y$ decreases by at least 1 in each iteration of the loop.

Since $x_0 = a$ and $y_0 = b$, it follows that if the algorithm has gone through $m = a + b - 2$ iterations of the loop without terminating, $x_m + y_m \leq x_0 + y_0 - m = 2$. But Invariant 10.4 tells us that $x_n + y_n \geq 2$ after every iteration of the algorithm, so $x_m + y_m \geq 2$ (this states that our potential $x + y$ never goes below a certain threshold). Therefore, $x_m = y_m = 1$ since both x_m and y_m are positive integers, and the algorithm terminates after the m-th iteration.

This completes the proof of correctness of Algorithm 1.

10.1.3 A Remark about Writing Proofs

Observe that we cannot prove $x_n+y_n \geq 2$ only with Invariant 10.3. If we did not prove Invariant 10.4 beforehand, we would not be able to show that our potential $x+y$ is bounded below by the threshold value of 2.

So, what would have happened if we had never stated and proved Invariant 10.4 and if we had never stated that lemma? We would have still known that $x + y$ was decreasing. That would have led us, after some thinking, to the following thought process: The quantity $x + y$ decreases after every iteration, so it will become negative at some point. But if it gets negative, one of x or y would be negative. This can't happen, though, because we only subtract smaller values from larger ones in the loop. Actually, we can't even get x or y to be zero because that would mean x and y were equal before the iteration that supposedly sets one of them to zero. So we see that $x, y > 0$ at all times, and the smallest $x + y$ can be is 2. At this point, we would have written down the statement of Invariant 10.4 and continued with our termination proof.

The lesson to take away from this is that you should not give up if you get stuck on a proof. When you get stuck, think about why you are stuck. Maybe other facts you know could help you continue with the proof. There may be some facts you have not used in your argument yet. Also try to make more observations about the problem. Many proofs consist of multiple ingredients that need to be mixed together in the right way, and chances are you are not going to find all the ingredients and techniques right away.

As we said earlier in the course, this takes some ingenuity, so make sure you give yourself enough time, too. Also, if you are stuck on a proof too long, open your mind to the possibility that the fact you are trying to prove is wrong (maybe not if we tell you to prove a fact on a homework assignment, but it's a good thing to keep in mind).

10.1.4 On the Importance of a Precise Specification

Before we move on to recursion, we point out the importance of a precise specification. Suppose we allowed a or b to be any natural numbers. Consider the input $a = 0$, $b = 1$. Then $x = 0$ and $y = 1$ when we enter the loop for the first time. The condition of the if statement is satisfied, so we subtract x from y on line 3, which doesn't do anything since $x = 0$. Thus, x and y have not changed at all in this iteration of the loop, and the same thing will happen in subsequent iterations. Thus, our algorithm gets stuck in an infinite loop on this input.

Note that in the situation above, Invariant 10.4 doesn't hold when the program starts, so we cannot use it to prove termination on the bad input. If we allowed inputs to be zero, we would not be able to prove the base case in the inductive proof of Invariant 10.4, and would fail to prove correctness of Algorithm 1. But we did require $a, b > 0$, so we don't need to reason about inputs that don't satisfy that requirement, and our proof of Invariant 10.4 goes through.

The lesson to take away from this is that, in addition to specifying the input-output relationship, the specification also indicates which inputs the program was designed to work with. We only need to prove correctness on those inputs. The program may work on other inputs too, but isn't guaranteed to work—it could also wipe your hard drive on bad input. It is the responsibility of the user who runs the program to ensure that the inputs are valid.

10.2 Recursion

The main idea behind recursion is that we can often reduce the solution of a problem to easier instances of the same problem. This concept applies to definitions as well as to algorithms or programs. For example, the constructor rule of an inductive definition can be thought of as a recursive definition since it defines one instance of a concept in terms of smaller instances. For programs, recursion means that we call the program from itself, but on a smaller input. We call such a call a *recursive call*.

It is key that each recursive call to the program is a call that uses a smaller input, so that some call is eventually made with an input for which the problem solved by the program is trivial and for which the program can return the answer right away without any additional recursive calls. This causes an end to the chain of recursive calls, and the recursively called instances of the program terminate, one by one, with the instance called last returning first. If the recursive calls don't use smaller inputs, the program could keep calling itself forever and never terminate.

The next three sections give examples of recursive algorithms for problems you have seen. The goal of these examples is to ease you into thinking in terms of recursion.

10.2.1 Fibonacci Numbers

We can view the definition of the n-th Fibonacci number as a recursive definition. The constructor rule says that $F_n = F_{n-1} + F_{n-2}$. In recursion terms, the "smaller instances" in the recursive definition of the n-th Fibonacci number are Fibonacci numbers with a lower index than n. The foundation rules $F_1 = 1$ and $F_2 = 1$ correspond to the base cases where we don't need another application of recursion.

We can turn the recursive definition of the n-th Fibonacci number into a recursive program that calculates the n-th Fibonacci number. Before we do so (and before we write any program), we should have a specification available. In this case, our program receives a positive integer, n, as input, and returns the n-th Fibonacci number. We give the program as a Function called Fib.

10.2.2 Greatest Common Divisor

We can also rewrite the GCD algorithm (Algorithm 1) using recursion. Observe that the behavior of the function GCD is the same as the behavior of Algorithm 1.

```
Algorithm Fib($n$)
    Input: $n$ - A positive integer
    Output: $F_n$ - The $n$-th Fibonacci number
(1) if $n = 1$ or $n = 2$ then return 1
(2)                  else return Fib($n$-1) + Fib($n$-2)
```

```
Algorithm GCD($a, b$)
    Input: $a, b$ - positive integers
    Output: gcd($a, b$) - their greatest common divisor
(1) if $a = b$ then return $a$
(2) if $a < b$ then return GCD($a, b - a$)
(3) if $a > b$ then return GCD($a - b, b$)
```

10.2.3 Grade School Multiplication Algorithm

Finally, we rewrite the grade school multiplication algorithm from last lecture using recursion. We offer some intuition behind the recursive description.

Write $b = 2q + r$ where $q = \lfloor b/2 \rfloor$ and $r \in \{0, 1\}$. Then r corresponds to the last bit in the binary representation of b, and q is formed by the all the remaining bits. To get q out of the binary representation of b, we just "cut off" the last bit from the binary representation of b. Thus, it is possible to obtain binary representations of q and r from the binary representation of b. Also notice that to multiply a number x by 2, we just append a zero after the last bit of the binary representation of x.

If $b = 2q + r$, we can write $ab = a(2q + r) = 2(aq) + ar$. This reduces the multiplication ab into three multiplications, namely ar, aq, and multiplying aq by 2. Let's argue that these multiplication problems are easier so as to give ourselves some confidence that we don't just keep increasing the number of multiplications without end.

First, multiplication by r is easy because r is either 0 or 1. In the former case, $ar = 0$, and in the latter case, $ar = a$, so we don't need an additional recursive call here. Second, the multiplication aq is a multiplication of a by a number that is smaller than b, so this is a multiplication problem with a smaller input than the original problem. Finally, we saw in the previous paragraph how to multiply by 2 in binary: We just append a zero at the end of the binary representation of the number we multiply by 2 and return right away without another recursive call. Thus, we have reduced the problem of multiplying a and b into three easier multiplication problems.

Now that we have some intuition, let's write down the algorithm. It is not an exact transcript of our intuition, but is close. It also allows for multiplication by zero.

```
Algorithm MULT($a$,$b$)
    Input: $a, b$ - integers
    Output: $ab$ - their product
(1) if $b = 0$ then return 0
(2) if $b$ is even then return $2 \cdot$ MULT($a$, $\lfloor b/2 \rfloor$)
(3)             else return $2 \cdot$ MULT($a$, $\lfloor b/2 \rfloor$) + $a$
```

10.3 Next Time

To argue correctness of recursive algorithms, we usually use induction like we did to argue program correctness for non-recursive programs. As before, we will prove partial correctness and termination.