Massachusetts Institute of Technology
6.042J/18.062J: Mathematics for Computer Science
Professor David R. Karger

Handout 14
March 10, 1997

Proving Algorithm Correctness

The following handout discusses the three "styles" of algorithm correctness proofs we have studied in this class. In order to highlight the frameworks rather than the content, I have chosen to present a single algorithm (our favorite, Euclid's algorithm) and its correctness proof under all three frameworks.

The three frameworks we have seen are as follows:

Functional (Recursive) Programs. In this model, the algorithm is given as a (typically recursive) functional program without assignments. Such programs are close to the static notion of mathematical functions, and are typically proved correct by induction on the depth of recursive calls made, but can sometimes be proved correct by induction on the values of the inputs.

Procedural (Iterative) Programs. In this model, the algorithm is given as a (typically iterative) procedure that modifies variables. Proof of correctness is typically via an *invariant,* a proposition that is true at all times while the program is executing. We prove that the variant is true at all times by induction on the number of program steps, number of iterations of the main program loop, or some variant thereof.

State Machines. In this model, the system is modeled as a collection of *states.* Each state is often given by a set of *state variables* whose values change from state to state. Computation takes place via a sequence of *transitions* from one state to another. As with procedural programs, we prove correctness by studying *invariants* that are true at every reachable state. We prove the invariant is true at every reachable state by induction on the number of transitions taken to reach a given state.

In the next sections, I will describe Euclid's algorithm in each of the 3 formats. In order to do so, I will need to make use of two lemmas: one which always forms the core of the inductive step in the proof of termination, the other which forms the core of the inductive step in the correct-if-terminates proof. Both these lemmas are from the class notes.

Lemma 0.1 $x \bmod y < y$.

Lemma 0.2 *If* $y > 0$*, then* $GCD(x, y) = GCD(y, x \bmod y)$.

Proof. Notes from lecture 6. ■

1 Functional Programs

In this model, the algorithm is given as a (typically recursive) functional program without assignments. Such programs are close to the static notion of mathematical functions, and are typically proved correct by induction on the depth of recursive calls made to solve the problem. Let's momentarily ignore the issue of termination. The basic inductive hypothesis has the form "the program returns the right answer when the program terminates at recursion depth n," and the inductive step relies on the fact that the recursive call returns the right answer (note that the recursive call will involve a recursion depth one less than that of the original call).

Let's consider the concrete case of Euclid's Algorithm. The function-program formulation of this algorithm is as follows.

> **Procedure** Euclid(x, y), $(x, y \in \mathbb{N},\ x \neq 0)$
>
> **if** $y = 0$, then return x.
> **else** return Euclid$(y, x \bmod y)$.

To prove correctness, we need to show two things:

- If it ever terminates, it gives the right answer (partial correctness).
- It eventually terminates.

Both of these are normally part of correctness proofs (for standard sequential algorithms).

Partial correctness As stated above, partial correctness is usually proved by induction on the depth of recursive calls made by the functional program. If the program, when called with a certain pair of arguments, terminates at a depth of n recursive calls, then by definition recursive subproblem it generates must terminate at a depth of $n-1$ recursive calls. Here's the proof for Euclid's Algorithm.

Theorem 1.1 *If Euclid(x, y) terminates, then its output is GCD(x, y).*

Proof. The proof is by induction on the number of recursive calls. The inductive hypothesis is "for any x and y such that Euclid(x, y) terminates after n recursive calls, the output is GCD(x, y)."

First consider the base case. This is the case $n = 0$, ie, no recursive calls. This case only happens when $y = 0$. But we know that GCD$(x, 0) = x$, and this is what we return in this case. So base case is proven.

Now we prove the inductive step. Suppose the inductive hypothesis is true for any pair of arguments that makes the algorithm terminate at a depth of $n-1$ recursive calls. Consider an instance Euclid(x, y) that terminates in n calls. Then this instance generates a recursive call to Euclid$(y, x \bmod y)$ that terminates at a depth of $n-1$ calls. It follows from the inductive hypothesis that Euclid$(y, x \bmod y)$ returns GCD$(y, x \bmod y)$. This, then, is also the value that we return. But now observe that by Lemma 0.2, $\text{GCD}(y, x \bmod y) = \text{GCD}(x, y)$. It follows that $\text{Euclid}(x, y)$ returns $\text{GCD}(x, y)$. This completes the proof of the inductive step. ∎

We have proven partial correctness, now we prove termination. This time we use induction on the input argument values (we can't use induction on the depth of recursion—this would be circular reasoning, since until we prove the algorithm terminates, we don't know that the depth of recursion is finite).

Theorem 1.2 *Euclid(x, y) terminates at a depth of at most y recursive calls.*

Proof. Strong induction on y. The inductive hypothesis is: $\text{Euclid}(x, y)$ terminates at a depth of at most y recursive calls. Note this looks the same as the theorem. But the theorem actually has an implicit universal quantification: "$(\forall y)$, the statement holds." We will prove this using the inductive hypothesis.

We start with the base case. If $y = 0$, then we don't make any recursive call. So clearly we terminate at a depth of $0 \leq y$ recursive calls.

Now consider the inductive step. Consider a call to $\text{Euclid}(x, y)$ where $y > 0$. This generates a call to $\text{Euclid}(y, x \bmod y)$. By Lemma 0.1, $x \bmod y < y$. It follows from the inductive hypothesis that $\text{Euclid}(y, x \bmod y)$ terminates at a depth of at most $x \bmod y \leq y - 1$ recursive calls. Therefore, $\text{Euclid}(x, y)$ terminates at a recursion depth which is one greater, and thus at most y. ∎

2 Iterative Algorithms

In this model, the algorithm is given in terms of a (typically iterative) procedure that modifies variables. Proof of correctness is typically via an *invariant,* a proposition that is true at all times while the program is executing. In particular, the invariant is true when the program finishes—if we choose the right invariant, this will prove that the program is correct at the end.[1] To prove that the invariant is always true, we use induction on the number of program steps, number of iterations of the main program loop, or some variant thereof. As before, we need to show both partial correctness and termination.

We now reformulate Euclid's Algorithm as an iterative algorithm. In the following code, I am using simultaneous assignments to avoid temporary variables: $(x, y) \leftarrow (a, b)$ means x gets set to a and y to b simultaneously.

[1]In some cases, the key statement will not actually be an invariant; instead, it might be something like "after n iterations of the while loop, the variable a has value n^2." But the proof framework is the same as the one we will see here.

```
Procedure Euclid(x0, y0)

(x, y) ← (x0, y0)
while y ≠ 0
    (x, y) ← (y, x mod y)
return x
```

I've introduced two extra variables, x_0 and y_0. These are convenient handles to let me talk about the "original input values."

To prove partial correctness of this program, we will introduce an *invariant:* a proposition that is true at all times in the program. For this proof, our invariant will be "$\text{GCD}(x, y) = \text{GCD}(x_0, y_0)$." Let's see how this works:

Theorem 2.1 *Euclid*(x_0, y_0) *returns GCD*(x_0, y_0).

Proof. We prove that once x and y have been assigned their initial values, the invariant "$\text{GCD}(x, y) = \text{GCD}(x_0, y_0)$" is true at all times during execution of the program. Why does this suffice? Well, consider the situation when the program terminates. In order for the while loop to exit, it must be the case that $y = 0$. At this point, we return x. However, by our invariant, when we return x, it holds that $\text{GCD}(x, y) = \text{GCD}(x_0, y_0)$. But $y = 0$, meaning that we return $x = \text{GCD}(x, 0) = \text{GCD}(x, y) = \text{GCD}(x_0, y_0)$.

To prove the invariant, we use induction on the number of iterations of the while loop. For the base case, when 0 iterations have been performed, we know that $x = x_0$ and $y = y_0$, so the invariant is obviously true. For the inductive step, suppose the invariant is true after $n-1$ iterations of the while loop. Consider the n^{th} iteration. In this iteration, we set $(x, y) \leftarrow (y, x \bmod y)$. Let x' and y' denote these new values for x and y. By Lemma 0.2, we know that $\text{GCD}(x', y') = \text{GCD}(x, y)$. By the inductive hypothesis, we know that $\text{GCD}(x, y) = \text{GCD}(x_0, y_0)$. It follows that $\text{GCD}(x', y') = \text{GCD}(x_0, y_0)$, proving our inductive step. ∎

Note that this proof used exactly the same content (namely, Lemma 0.2) as did the proof in the functional-program case. Here, we just packaged that content a little differently.

We also need to prove termination. We do so as follows (again, the content will be the same as in the functional program, namely Lemma 0.1, but the packaging will be slightly different).

Theorem 2.2 *Euclid*(x, y) *always terminates in a finite number of steps.*

Proof. We prove by induction that after n iterations of the while loop, $y \leq y_0 - n$. It follows that after y_0 iterations of the while loop, $y \leq 0$, meaning the while loop must terminate at or before this point.

To prove the induction, start with the base case. After no iterations, $y = y_0$, so the inductive hypothesis holds. Now suppose the hypothesis holds after $n - 1$ iterations. In the n^{th} iteration, we replace y with $x \bmod y$. But by Lemma 0.1, $x \bmod y < y$, or in other words, $x \bmod y \leq y - 1$. Thus, the n^{th} iteration decreases y by at least 1. Since by the inductive hypothesis, we had $y \leq y_0 - (n-1)$ in the $(n-1)^{st}$ iteration, it must be that after the n^{th} iteration, $y \leq y_0 - (n-1) - 1 = y_0 - n$. ∎

3 State Machines

In this model, the system is modeled as a collection of *states.* Each state is often given by a set of *state variables* whose values change from state to state. Computation takes place via a sequence of *transitions* from one state to another. As with procedural programs, we prove correctness by studying *invariants* that are true at every reachable state. We prove the invariant is true at every reachable state by induction on the number of transitions taken to reach a given state. If a state is reachable, it is reachable by a finite number of transitions. This means we arrive at it from some immediately preceding state that was reached by one fewer transitions.

Unlike the previous two models, state machine analyses don't really talk about the algorithm running to termination. State machines are often used to model systems that run forever and have transitions at unpredictable rates, so what we really care about is the entire set of reachable states and their properties.

Euclid is therefore a somewhat weird algorithm to study in the state machine model, since all we really care about is what happens at termination. But with appropriate shoehorning, we can use the state machine model to say something pretty close to the previous two models.

Here's a state machine model of Euclid's Algorithm. We can model the algorithm by a state machine as follows. The state set Q consists of values for 5 "state variables":

$x_0 \in \mathbb{N}$
$y_0 \in \mathbb{N}$
$x \in \mathbb{N}$
$y \in \mathbb{N}$
$answer \in \mathbb{N} \cup \{null\}$

The variables x_0 and y_0 represent the "inputs" to the program, whose GCD we want to compute. The two state variables x and y are auxiliary state variables which keep track of the current values on which we are working. The final state variable, *answer* is intended to hold the final answer. It can hold either an answer or *null*, which represents that the answer has not yet been found.

One thing we need to define for a state machine is its start state(s). We specify that the start state is the state in which $x = x_0$, $y = y_0$, and $answer = null$.

The transition relation is defined by the following two code fragments. In this algorithm, only two kinds of things can happen: reductions, and final decisions.

reduction
  Precondition:
    $answer = null$
    $y_1 \neq 0$
  Effect:
    $(x_1, y_1) \leftarrow (y_1, x_1 \bmod y_1)$

decision
  Precondition:
    $answer = null$
    $y_1 = 0$
  Effect:
    $answer \leftarrow x_1$

In a typical execution, a sequence of *reduction* transitions occur, followed by a single *decision* transition.

We will prove the following property of all reachable states of this machine:

Theorem 3.1 *If answer $\neq$ null then answer $= GCD(x, y)$.*

This is the statement of *partial correctness* for the Euclidean GCD algorithm. (Since the transitions happen at arbitrary times, it is a little odd to talk about termination, so we won't. The above property says that the state machine, if it decides on an answer, decides the correct answer).

Proof. To prove the property, we make use of a different invariant—one quite similar to that used in the iterative program. We will use the following convenient notation. For a state q, $q.x$ denotes the value of variable x in state q. The invariant says the following: "in any reachable state q, $\mathrm{GCD}(q.x, q.y) = \mathrm{GCD}(q.x_0, q.y_0)$." We prove this invariant by induction on the number of transitions to the reachable state.

First consider the base case. If we are looking at a state reached in 0 transition steps, we must be looking at a start state. In a start state q, $q.x = q.x_0$ and $q.y = q.y_0$, so the invariant is trivially true.

Now the inductive step. Suppose the invariant is true for any state reached in $n - 1$ transitions. Consider a state q' reached via n transitions. It must be preceded by a state q that was reached via $n - 1$ transitions. The inductive hypothesis therefore applies to q, meaning that $\mathrm{GCD}(q.x, q.y) = \mathrm{GCD}(q.x_0, q.y_0)$. Now consider the transition that takes q to q'. There are two kinds of transitions, so we must consider cases.

If the transition is of type *decision*, then the proof is obvious. Since decision changes only the value of *answer*, we know that $q.x = q'.x$, $q.y = q'.y$, and so on. So $\mathrm{GCD}(q'.x, q'.y) = \mathrm{GCD}(q.x, q.y) = \mathrm{GCD}(q.x_0, q.y_0) = \mathrm{GCD}(q'x_0, q'.y_0)$.

If the transition is of type *reduce*, we again use Lemma 0.2. From the transition, we see that $q'.x = q.y$ and $q'.y = x \bmod y$. while $q.x_0 = q'.x_0$ and $q.y_0 = q'.y_0$. It follows from Lemma 0.2 that $\mathrm{GCD}(q'.x, q'.y) = \mathrm{GCD}(x, y)$. So we deduce that $\mathrm{GCD}(q'.x, q'.y) = \mathrm{GCD}(q.x, q.y) = \mathrm{GCD}(q.x_0, q.y_0) = \mathrm{GCD}(q'.x_0, q'.y_0)$.

So we have proven the invariant claimed at the start of the proof. We need go only a little farther in order to prove the invariant of the theorem. Suppose that in reachable state q', we have $q'.answer \neq null$. Consider the previous state q. What transition could deliver us to q'? Only a *decision* transition. It follows that $q.y = 0$ (since this is a precondition for the *decision* transition to occur). By the invariant that we proved before, we know that $\mathrm{GCD}(q.x, q.y) = \mathrm{GCD}(q.x_0, q.y_0)$. Therefore, since $q.y_0 = 0$, we know that $q.x = \mathrm{GCD}(q.x, q.y) = \mathrm{GCD}(q.x_0, q.y_0)$. Thus in the *decision* transition, we set $q'.answer = q.x = \mathrm{GCD}(q.x_0, q.y_0)$ as desired. This proves the claim. ∎

Note the similarity to our iterative algorithm proof. In fact, this state machine can be seen as a model for the state of the iterative algorithm.