CSC B36 Additional Notes
proving program correctness

© Nick Cheng

⋆ Introduction

These notes summarize the main points on proving program correctness from chapter 2 of the course notes. Here we extend the meaning of "program" to include a segment of code.

◦ Notation

Given a variable x and an iteration of a loop, we use the convention of letting x denote the value of the variable *before* the iteration, and x' denote the value of the variable *after* the iteration.

Also, given an algebraic expression e that includes variables used in a loop, we let e denote the value of the expression *before* the iteration, and e' denote the value of the expression *after* the iteration.

⋆ What does it mean to say a program is correct?

When we say that a program (or a segment of code) is correct, we mean

if the proper condition to run the program holds, and the program is run, then the program will halt, and when it halts, the desired result follows.

- The proper condition to run a program is called its *precondition.*
- That the program halts is called *termination.*
- The desired result following a program's termination is called its *postcondition.*
- Note that program correctness is always described with respect to its precondition and postcondition. I.e., correctness also depends on precondition and postcondition.
- The pre/post-condition pair is called the program's *specification.*

With the above terminology, we can think of program correctness as follows.

Program correctness: *if precondition then (termination and postcondition).*

So proving correctness means proving

precondition ⇒ (termination and postcondition).

Sometimes it is convenient to prove separately the termination part and the postcondition part. In this case we divide the proof into two parts.

(a) precondition ⇒ termination — this part is sometimes just called *termination*,

(b) (precondition and termination) ⇒ postcondition — this part is called *partial correctness.*

So proving correctness means proving partial correctness and termination.

◦ **Useful result for proving termination (for iterative programs)**

When proving termination for an iterative program, we will make use of the following corollary to the Principle of Well-Ordering (PWO).

Corollary:
Every (strictly) decreasing sequence of natural numbers is finite.

Proof:
As suggested by its introduction, the proof uses PWO.
This is left as an exercise to the reader.

⋆ Proving correctness of iterative programs

Iterative programs are programs with loops. When proving that a loop (or program with a loop) is correct (with respect to some pre/post-condition pair), we prove partial correctness and termination separately. For both parts we need a *loop invariant*, which describes how the variables in the loop are used to achieve the postcondition.

◦ **Loop invariants**

A loop invariant (LI) is a statement that is true on entering the loop, and after every iteration (assuming the precondition holds).

To prove an LI, we use a form of induction, where

(a) [Basis:] we prove that the LI holds on entering the loop,

(b) [Induction Step:] we prove that if the LI holds *before* an iteration, then it also holds *after* that iteration.

Aside: Do you see how proving the above amounts to proving that the LI holds on entering the loop and after every iteration?

◦ **Steps in proving iterative code correct**

1. Formulate a loop invariant (LI). This usually requires understanding how the code works. Sometimes it helps to trace it with various input. The LI should describe the purpose of each variable.

2. Use induction to prove the LI from step 1. I.e., Prove that the LI holds on entering the loop, and after every iteration (assuming the precondition holds).

3. Use the LI from step 1 to prove partial correctness. This means proving that if the loop halts, then the postcondition follows. Since the loop halts exactly when its exit condition (negation of the condition in the **while** loop) is satisfied, what we prove can be summarized as

   (loop exit condition and LI) $\Rightarrow$ postcondition

4. Use LI from step 1 to prove termination. We start by finding an expression e that uses the variables in the loop. We want e so that

   (A) the value of e is a natural number on entering the loop, and after every iteration,

   (B) the value of e decreases with every iteration.

5. Prove that the sequence of values of e after each iteration is a decreasing sequence of natural numbers. This means proving:

   (A) the value of e is always a natural number,

   (B) the value of e decreases with every iteration. I.e.,
   for any iteration, $e' < e$.

Note: In doing steps 2 to 5, it is not unusual to find that the LI from step 1 needs to be modified. So sometimes all the steps need to be revisited multiple times before a complete proof is obtained.

⋆ Proving correctness of recursive programs

Recursive programs are programs that makes recursive calls. When proving that a recursive program is correct, we do not need to prove termination and partial correctness separately. Instead we encapsulate both ideas in a single predicate $P(n)$ where n is a natural number that somehow captures the "size" of the input, and we prove that $P(n)$ holds for all possible input sizes. This is usually done by complete induction.

∘ Predicate for recursive code and input "size"

The predicate for a recursive program always has this form.

$P(n)$: If precondition holds, and the program runs, and its input size is n, then
the program halts and the postcondition holds after it halts.

Note A: See how termination and partial correctness are both captured in $P(n)$.

Note B: By *input*, we mean the arguments to a recursive function/method/subprogram. By *size*, we mean some natural number that we associate with each possible input. E.g., for a factorial function FACT(n), the size could be the value of n. For a sort method SORT(A) where A is an array or a list, the size could be the length of A. See how the number associated with the input is a measurement of the size of the input.

∘ Steps in proving recursive code correct

1. Understand how the code works. Trace it with various input if needed. In particular describe how the input for any recursive calls of the code is smaller than the input for the initial call of the code.

2. Use the work from step 1 to formulate a predicate $P(n)$ where n is a natural which describes the size of the code's input.

3. Use complete induction to prove that $P(n)$ holds for every $n \in \mathbb{N}$. Proving this means we prove that the code works for all possible input sizes, which is a way of saying that the code is correct. The basis of the induction proof should correspond to input whose size is so small that no further recursive calls of the code is triggered. The induction step should correspond to input that does trigger more recursive calls of the code.