Computer Science 331

Proofs of Correctness of Algorithms

Mike Jacobson

Department of Computer Science
University of Calgary

Lectures #2–3

Outline

How Do We Specify a Computational Problem?

Recall: a computational problem is specified by one (or more) pairs of *preconditions* and *postconditions*.

- *Precondition:* A condition that one might expect to be satisfied when the execution of a program begins. This generally involves the algorithm's *inputs* as well as initial values of *global variables*.
- *Postcondition:* A condition that one might want to be satisfied when the execution of a program ends. This might be
  - A set of relationships between the values of inputs (and the values of global variables when execution started) and the values of outputs (and the values of global variables on a program's termination), or
  - A description of output generated, or exception(s) raised.

Example: Specification of a "Search" Problem

Precondition P_1: Inputs include

- `n`: a positive integer
- `A`: an integer array of length `n`, with entries

$$\texttt{A[0]}, \texttt{A[1]}, \ldots, \texttt{A[n-1]}$$

- `key`: An integer found in the array (ie, such that `A[i]` $=$ `key` for at least one integer `i` between 0 and `n-1`)

Postcondition Q_1:

- Output is the integer `i` such that $0 \leq$ `i` $<$ `n`, `A[j]` $\neq$ `key` for every integer `j` such that $0 \leq$ `j` $<$ `i`, and such that `A[i]` $=$ `key`
- Inputs (and other variables) have not changed

This describes what should happen for a "successful search."

Example: Specification of a "Search" Problem

Precondition P_2: Inputs include

- `n`: a positive integer
- `A`: an integer array of length `n`, with entries

$$\texttt{A[0]}, \texttt{A[1]}, \ldots, \texttt{A[n-1]}$$

- `key`: An integer not found in the array (ie, such that `A[i]` $\neq$ `key` for every integer `i` between 0 and `n-1`)

Postcondition Q_2:

- A `notFoundException` is thrown
- Inputs (and other variables) have not changed

This describes what should happen for an "unsuccessful search."

Example: Specification of a "Search" Problem

A problem can be specified by multiple precondition-postcondition pairs

$$(P_1, Q_1); (P_2, Q_2); \ldots; (P_k, Q_k)$$

as long as it is not possible for more than one of the preconditions

$$P_1, P_2, \ldots, P_k$$

to be satisfied at the same time.

For example, if P_1, Q_1, P_2, and Q_2 are as in the previous slides then the pair of precondition-postcondition pairs

$$(P_1, Q_1); (P_2, Q_2)$$

could specify a "search problem" in which the input is expected to be *any* positive integer `n`, integer array `A` of length `n`, and integer `key`.

When is an Algorithm Correct?

Suppose, first, that a problem is specified by a *single* precondition-postcondition pair (P, Q).

An algorithm (that is supposed to solve this problem) is *correct* if it satisfies the following condition: If

- inputs satisfy the given precondition P and
- the algorithm is executed

then

- the algorithm eventually halts, and the given postcondition Q is satisfied on termination.

Note: This does not tell us *anything* about what happens if the algorithm is executed when P is *not* satisfied.

When is an Algorithm Correct?

Suppose, next, that $k \geq 2$ and that a problem is specified by a sequence of k precondition-postcondition pairs

$$(P_1, Q_1); (P_2, Q_2); \ldots; (P_k, Q_k)$$

where it is impossible for more than one of the preconditions to be satisfied at the same time.

An algorithm (that is supposed to solve this problem) is *correct* if the following is true for *every* integer i between 1 and k: If

- inputs satisfy the given precondition P_i and
- the algorithm is executed

then

- the algorithm eventually halts, and the given postcondition Q_i is satisfied on termination.

When is an Algorithm Correct?

A consequence of the previous definitions: Consider a problem specified by a sequence of k precondition-postcondition pairs

$$(P_1, Q_1); (P_2, Q_2); \ldots; (P_k; Q_k).$$

Then an algorithm that is supposed to solve *this* problem is *correct* if and only if it is a *correct* solution for each of the k problems that are each specified by the single precondition-postcondtion paiir P_i and Q_i, for i between 1 and k.

$\Longrightarrow$ It is sufficient to consider problems that are specified by a single precondition and postcondition (and we will do that, from now on).

Why are Proofs of Correctness Useful?

Who Generates Proofs of Correctness?

- Algorithm designers (whenever the algorithm is not obvious). Other people need to see evidence that this new algorithm really *does* solve the problem!
- Note that testing *cannot* do this (in general).

Who Uses Proofs of Correctness?

- Anyone coding, testing, or otherwise maintaining software implementing any nontrivial algorithm need to know *why* (or *how*) the algorithm does what it is supposed in order to do their jobs well.

What Will Happen in CPSC 331?

The instructor will be presenting proofs of correctness of several nontrivial algorithms in this course.

You will be expected to use these proofs to

- document your code more effectively,
- make your code easier to test and maintain,
- design test suites that are more comprehensive, and
- debug software more effectively.

You will be asked to develop a few *really, really easy* proofs in this course, just to show that you know how. You won't be asked to do more than that here.

What Will Happen Later?

Computer Science majors will be asked to *write* proofs of correctness of algorithms in senior courses they take later on.

- In CPSC 313 students are asked to develop some *very* simple *kinds* of algorithms (given as very simple "automata" or "machines"). Students may be asked to prove the correctness of these, in CPSC 313.
- In CPSC 413 students are asked to show that they know how to use some "algorithm design techniques" by developing algorithms (that use these techniques) for specific problems. They might be asked to show that their algorithms are correct in CPSC 413.

Overview of the Rest of the Lecture(s)

1. We will consider proofs of correctness of algorithms written using a "tiny programming language" that includes
   - `continue` statement — does nothing
   - assignment statements
   - `if-then-else` statements — "branching"
   - `while-do` loops

   and where programs can be *sequences* of the above
2. We will see how to handle additional control structures
3. We will introduce nonrecursive procedures
4. We will introduce simple recursion
5. Tutorial exercises will help you to apply these ideas to software development using Java

Necessary Background

Propositional Logic: We will use the following operators on boolean values.

$\neg p$	"not p:" `true` if and only if p is `false`
$p \wedge q$	"p and q:" `true` when p and q are *both* `true`
$p \vee q$	"p or q:" `true` when at least one of p or q is `true`
$p \Rightarrow q$	"p `implies` q:" `true` unless p is `true` *and* q is `false`
$p \iff q$	"p `if and only if` q:" `true` when p and q have the same truth value

Note: Some texts use different symbols for these operators!

One Part of a Proof of Correctness: Partial Correctness

Partial Correctness: If

- inputs satisfy the precondition p, and
- algorithm or program S is executed,

then *either*

- S halts and its inputs and outputs satisfy the postcondition q

or

- S does not halt, at all.

Generally written as

$$\{p\} \;\; S \;\; \{q\}$$

Proof Rule: `continue`

Statement: `continue`

Effect: No change

Conclusion: In order to show that

$$\{p\} \;\texttt{continue}\; \{q\}$$

you should prove that

$$p \Rightarrow q$$

Wait a Minute!

Q: What (if anything) have we accomplished here?

A: strategy for proof of empty code blocks

Q: Why would you want to have a `continue` statement anyway?

A: Main reason:

- flexibility in representing other code structures using our limited operations
- Eg. use `continue` to represent empty `else` condition in `if-then-else` structure

Proof Rule: Assignment Statements

Statement: `x := e`

Effect: The value of the variable `x` is reset to be that of the expression `e`

Conclusion: A bit complicated...

Notation for Substitution

Definition: Let

- q be a condition,
- `x` be a variable, and let
- `e` be an expression — of the same type as `x`.

Then $q^{\mathtt{x}}_{\mathtt{e}}$ is the condition produced from q by replacing every occurrence of the variable `x` with the expression `e`.

Examples:

q	`e`	$q^{\mathtt{x}}_{\mathtt{e}}$
`x` is odd	3	3 is odd
`x` is odd	`x` + 1	`x` + 1 is odd

Note: q is `true` *after* a statement "`x := e`" is executed if and only if $q^{\mathtt{x}}_{\mathtt{e}}$ was `true` *before*!

Proof Rule: Assignment Statements, Continued

Statement: `x := e`

Effect: The value of the variable `x` is reset to be that of the expression `e`

Conclusion: In order to show that

$$\{p\} \quad \mathtt{x} := \mathtt{e} \quad \{q\}$$

you should prove that

$$p \Rightarrow q^{\mathtt{x}}_{\mathtt{e}}$$

Q. Why do this?

A. Clarity — helps make what needs to be proved more explicit (simple implication expressed in x only

Proof Rule: Branching

Statement: `if c then` S_1 `else` S_2 `end if`

Effect: If c is `true` then S_1 is executed; S_2 is executed otherwise

Conclusion: In order to show that

$$\{p\} \ \texttt{if c then } S_1 \texttt{ else } S_2 \texttt{ end if} \ \{q\}$$

prove *both* of the following:

- $\{p \wedge (\texttt{c})\} \ \ S_1 \ \ \{q\}$
- $\{p \wedge \neg(\texttt{c})\} \ \ S_2 \ \ \{q\}$

Hold On, Now!

Q: We've gone from proving the partial correctness of one program to proving the partial correctness of *two* of them! How is this progress?

A: a compound statement/program has been broken into two statements, each of which can be proved separately.

Proof Rule: Loops

Problem: Show that

$$\{p\} \ \texttt{while c do } S \texttt{ end while} \ \{q\}$$

Observation: There is generally some condition that we expect to hold at the beginning of *every* execution of the body of the loop. Such a condition is called a *loop invariant*.

To Prove That a Condition r is a Loop Invariant for the Above:

1. Prove that $(p \wedge (\texttt{c})) \Rightarrow r$
2. Identify another condition $\widehat{r}$ and prove that
   - $\{r\} \ \ S \ \ \{\widehat{r}\}$
   - $(\widehat{r} \wedge (\texttt{c})) \Rightarrow r$

Note: essentially a *proof by induction* that the loop invariant holds after zero of more executions of the loop body.

Proof Rule: Loops, Continued

Statement: `while c do` S `end while`

Proof Rule: In order to prove that

$$\{p\} \ \texttt{while c do } S \texttt{ end while} \ \{q\}$$

it is sufficient to do the following:

1. Prove that $(p \wedge \neg(\texttt{c})) \Rightarrow q$
2. Identify a condition r and prove that r is a loop invariant using the previously given process for this
3. As part of the previous step you identified another condition, $\widehat{r}$, such that $\{r\} \ \ S \ \ \{\widehat{r}\}$. Complete this process by proving that $(\widehat{r} \wedge \neg(\texttt{c})) \Rightarrow q$

Proof Rule: Sequences of Programs

Statement: $S_1; S_2$

Effect: Program S_1 is executed, and then program S_2 is executed after that

Observation: There should be some well-defined objective that is achieved by the initial program S_1

Proof Rule: Do the following order to prove that

$$\{p\} \quad S_1; S_2 \quad \{q\}$$

1. Identify an intermediate assertion r
2. Prove that $\{p\} \quad S_1 \quad \{r\}$
3. Prove that $\{r\} \quad S_2 \quad \{q\}$

Another Part: Termination

Termination: If

- inputs satisfy the precondition p, and
- algorithm or program S is executed,

then

- S is guaranteed to halt!

Note: Partial Correctness + Termination $\Rightarrow$ Correctness!

Partial Correctness and Termination are often (but not always) considered separately because ...

- Different — independent — arguments are used for each
- Sometimes one condition holds, but not the other! Then the algorithm is *not* correct... but something interesting can still be established.

Proof Rule: `continue` and Assignment Statements

Problem: Prove that each of the statements

`continue` and `x` := `e`

halt (for any variable `x` and expression `e`) when a given precondition p is satisfied

Solution: There's nothing we need to do! These statements always halt.

Proof Rule: Branching

Statement: `if c then` S_1 `else` S_2 `end if`

Effect: If `c` is `true` then S_1 is executed; S_2 is executed otherwise

Conclusion: In order to prove that if p is satisfied and the above statement is executed, then it halts, prove *both* of the following:

- If $p \wedge$ (`c`) is satisfied and S_1 is executed then this program halts
- If $p \wedge \neg$(`c`) is satisfied and S_2 is executed then this program halts

Proof Rule: Loops

Problem: Show that if a condition p is satisfied and a loop

`while c do` S `end while`

is executed, then the loop eventually terminates.

Suppose that a *loop invariant* r for the precondition p and the above loop has already been found. You should have done this when proving the partial correctness of this loop — also useful to prove termination.

Proof Rule: To establish the above termination property, prove *each* of the following.

1. If the loop invariant r is satisfied and the loop body S is executed then the loop body terminates.
2. The loop body is only executed a finite number of times. (Information about how to prove *this* is on the next slide.)

Proof Rule: Loops, Continued

Definition: A *loop variant* for a loop

`while c do` S `end while`

is a *function* f_L from program variables to the set of *integers* that satisfies the following additional properties:

- The value of f_L is decreased *by at least one* every time the loop body S is executed
- If the value of f_L is less than or equal to zero then the loop test c is `false`

Note: The *initial* value of f_L is an upper bound for the number of executions of the loop body before the loop terminates.

Proof Rule: Loops, Concluded

Problem: Find an upper bound on the number of executions of the loop body when a condition p is satisfied and the loop

`while c do S end while`

is executed

Solution:

1. Identify an integer-valued function f_L of the program variables
2. Prove that f_L is a loop variant for this loop
3. Return the *initial* value of f_L as an upper bound on the number of executions of the loop body

Note: this proves termination!

Proof Rule: Sequences of Programs

Objective: Show that if condition p is satisfied and the program S_1; S_2 is executed then this program halts

Assumption: While proving partial correctness we have found an intermediate assertion r such that

$$\{p\} \;\; S_1 \;\; \{r\}$$

Proof Rule: In order to prove that S_1; S_2 halts when executed with condition p satisfied

1. Prove that S_1 halts when executed with condition p satisfied
2. Prove that S_2 halts when executed with condition r satisfied, for r as above

Example

Prove the correctness of the following algorithm.

Precondition: n is a positive integer

Postcondition: n is unchanged and $\texttt{sum} = \sum_{j=1}^{\texttt{n}} j$

Algorithm:

```
i := 1
sum := 1
while (i < n) do
  i := i + 1
  sum := sum + i
end while
```

For detailed solution, see Lecture Supplement online.

Additional Control Structures

Any real programming language is likely to include additional control structures...

Example: Consider the statement `if c then` S `end if`

... but these can generally be rewritten using the statements and control structures we have already ...

Example, Continued: This has the same effect as

`if c then` S `else continue end if`

...so we can replace the code that uses a "new" structure with the equivalent code (that we know how to analyze) and then proceed.

Additional Control Structures, Continued

Example, Continued: Suppose we wish to prove that if a condition p is satisfied and the statement

`if c then` S `end if`

is executed, then this program terminates and the condition q is satisfied on termination.

Then it is sufficient to prove that if the condition p is satisfied and the "equivalent" statement

`if c then` S `else continue end if`

is executed, then the program terminates and the condition q is satisfied on termination.

Additional Control Structures, Concluded

Example, Concluded: Referring to the previous notes we can deduce the following about this example.

Necessary and Sufficient To Prove Partial Correctness:

- prove $\{p \wedge (\texttt{c})\}\ \ S\ \ \{q\}$
- prove $(p \wedge \neg(\texttt{c})) \Rightarrow q$

Necessary and Sufficient to Prove Termination:

- prove that S terminates

Nonrecursive Methods

Suppose a program consists of a set of methods, some of which call each other, but recursion is not used.

- Then one of these methods — say, "method A" — does not call any methods at all.
- Prove that method A is correct.
- Suppose method A has precondition p_A and postcondition q_A. Add a new proof rule:

  > *If precondition p_A is satisfied and method A is called then the method terminates and postcondition q_A is satisfied.*

  Continue as if method A did not exist and calls to the method were analyzed using the above.
- Iterate until the correctness of all methods (including the main method) has been established.

Simple Recursive Methods

Suppose method A calls itself (but does not call any other methods).

In this case it is often possible to prove the correctness of this method using *mathematical induction*, proceeding by induction on the "size" of the inputs.

- Base cases of inductive proof: base cases of the recursive algorithm
- Induction hypothesis: algorithm is correct for inputs of "size smaller that n"
- Proof proceeds by proving correctness while assuming the induction hypothesis (i.e., every recursive call returns the correct output)

Example

Prove the correctness of the following algorithm.

Precondition: `i` is a positive integer

Postcondition: the value returned is the i^{th} Fibonacci number, F_i

Algorithm:

```
long fib (int i)
 if (i == 0) return 1
 if (i == 1) return 1
 return fib(i-1) + fib(i-2)
```

See Self-Study Exercises #2 (Problem 1)

Applications to Java Development

A proof of correctness of an algorithm includes detailed information about the expected state of inputs and variables at every step during the computation.

This information can be included in documentation as an aid to other developers. It also facilitates effective testing and debugging.

Self-study exercises can be used to learn more about this.

Can This All Be Automated?

The following questions might come to mind.

Q: Is it possible to write a program that decides whether a given program is correct, providing a proof of correctness of the given program if it is?

A: No! Computer science students will see in CPSC 313 that the simpler problem of determining whether a given program *halts* on a given input is "undecidable:" It has been *proved* that no computer program can solve this problem!

Q: Can a computer program be used to *check* a proof of correctness?

A: See our courses in "Artificial Intelligence" for information about this!

References

Discrete Mathematics textbooks sometimes include "proofs of correctness" of algorithms as an application of *mathematical induction:*

Recommended References:

- Susanna S. Epp
  Discrete Mathematics with Applications, Third Edition
  See Section 4.5
- Kenneth H. Rosen
  Discrete Mathematics and Its Applications, Sixth Edition
  See Section 4.5

Note: Epp's presentation of this topic is quite different from these notes.

For Further Reading

Proofs of correctness are discussed briefly in the textbook (Section 4.3). Also, Chapter 1 of

- Michael Soltys
  An Introduction to the Analysis of Algorithms

contains a nice introduction to proofs of correctness and is freely available online!

Each of the following references is available in the library:

- Edsger W. Dijkstra
  A Discipline of Programming
- David Gries
  The Science of Programming

These may be challenging, especially for students who have not already completed PHIL 279 (or taken another course in mathematical logic)!