Computer Science 331

Correctness of Algorithms

Mike Jacobson

Department of Computer Science
University of Calgary

Lectures #2-4

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

- $\mathtt{n}$: a positive integer
- $\mathtt{A}$: an integer array of length $\mathtt{n}$, with entries

$$\mathtt{A[0]}, \mathtt{A[1]}, \dots, \mathtt{A[n-1]}$$

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

Postcondition Q_1:

- Output is the integer $\mathtt{i}$ such that $0 \le \mathtt{i} < \mathtt{n}$, $\mathtt{A[j]} \neq \mathtt{key}$ for every integer $\mathtt{j}$ such that $0 \le \mathtt{j} < \mathtt{i}$, and such that $\mathtt{A[i]} = \mathtt{key}$
- 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 $\texttt{A[i]} \neq \texttt{key}$ 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-postcondition pair 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.

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

$$\{P\} \ \ S \ \ \{Q\}$$

Note: Detailed proofs rely heavily on discrete math and logic.

How to Prove Partial Correctness of Algorithms?

Consider algorithm S:

- Divide S into sections $S_1; S_2; \ldots; S_K$
  - assignment statements
  - loops
  - control statements (i.e., if-then-else)
  - (other programming constructs)
- Identify intermediate assertions R_i so that
  - $\{P\}\ S_1\ \{R_1\}$
  - $\{R_1\}\ S_2\ \{R_2\}$
  - ...
  - $\{R_{K-1}\}\ S_K\ \{Q\}$
- After proving each of these, we can then conclude that
  - $\{P\}\ S_1; S_2; \ldots; S_K\ \{Q\}$
  - equivalently, $\{P\}\ S\ \{Q\}$

Example: Proof of Partial Correctness

Problem Definition: Finding the largest entry in an integer array.

Precondition P: Inputs include

- `n`: a positive integer
- `A`: an integer array of length `n`, with entries `A[0]`, ..., `A[n-1]`

Postcondition Q:

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

Example: Pseudocode

`int FindMax(A, n)`
$i = 0$
$j = 1$
while $(j < n)$ **do**
  if $A[j] > A[i]$ **then**
    $i = j$
  end if
  $j = j + 1$
end while
return i

Example: Intermediate Assertion

Intermediate Assertion I:

- `n`: a positive integer
- `A`: an integer array of length `n`
- `i` $= 0$ and `j` $= 1$

Divide into Sections:

$\{P\}$
$i = 0, \quad j = 1$
$\{I\}$
while $(j < n)$ **do**
  if $A[j] > A[i]$ **then**
    $i = j$
  end if
  $j = j + 1$
end while
$\{Q\}$

Example: Proof of each Section

Prove the correctness of each section of the algorithm using the intermediate assertion I:

1. First Section: $\{P\}\ i = 0; j = 0\ \{I\}$
   - correctness is trivial
2. Second Section: $\{I\}$ `while` ... `end while` $\{Q\}$
   - proof is needed

$\Longrightarrow$ *In CPSC 331, we focus on proving correctness of simple loops and recursive programs.*

Correctness of Loops

Problem: Show that

$$\{P\}\ \texttt{while}\ G\ \texttt{do}\ S\ \texttt{end while}\ \{Q\}$$

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*.

A condition R is a Loop Invariant if:

1. **Base Property:** P implies that R is True before the first iteration of the loop
2. **Inductive Property:** If R is True before an iteration and the loop guard G is True, then R is True after the iteration

Example: Loop Invariant

Claim: Assertion R is a loop invariant:

- $1 \leq j \leq n$
- $\texttt{A[i]} \geq \texttt{A[k]}$ for $0 \leq \texttt{k} < j$

Proof.

1. **Base Property:** Before the first iteration of the loop:
   - $i = 0$ and $j = 1$; $A[0] \geq A[0]$
2. **Inductive Property:** If R is True before an iteration and $j < n$, then show that $1 \leq j_{after} \leq n$ and $A[i_{after}] \geq A[0], \ldots, A[j_{after} - 1]$
   - $j_{after} = j_{before} + 1 \Rightarrow 1 \leq j_{after} \leq n$
   - if $A[j_{before}] > A[i_{before}] \Rightarrow i_{after} = j_{before}$ and $A[i_{after}] \geq A[0], \ldots, A[j_{before}] \Rightarrow A[i_{after}] \geq A[0], \ldots, A[j_{after} - 1]$
   - if $A[j_{before}] \leq A[i_{before}] \Rightarrow i_{after} = i_{before}$ and $A[i_{after}] \geq A[0], \ldots, A[j_{before}] \Rightarrow A[i_{after}] \geq A[0], \ldots, A[j_{after} - 1]$ □

Correctness of Loops: Summary

Problem: Prove that

$$\{P\}\ \texttt{while}\ G\ \texttt{do}\ \ S\ \ \texttt{end while}\ \{Q\}$$

Solution:

1. Identify a loop invariant R and prove:
   - **Base Property:** P implies that R is True before the first iteration of the loop
   - **Inductive Property:** If R is True before an iteration and the loop guard G is True, then R is True after the iteration

   Note: essentially a *proof by induction* that the loop invariant holds after zero or more executions of the loop body.
2. Prove the **correctness of the postcondition**:
   - if the loop terminates after zero or more iterations, the Truth of R implies that Q is satisfied

Mathematical Induction

Problem: For all integers $k \geq k_0$, prove that property $P(k)$ is True.

Proof by Induction:

1. **Base Case:** Show that $P(k_0)$ is True
2. **Inductive Step:** Show that if $P(k)$ is True for some arbitrary integer $k \geq k_0$ (the **induction hypothesis**), then $P(k+1)$ is True.
   - choose an arbitrary $k \geq k_0$
   - show that $P(k+1)$ is True if $P(k)$ is True

Example: Mathematical Induction

Claim: $2^{2n} - 1$ is divisible by 3 for all integers $n \geq 1$.

Proof by Induction.

Let $P(n)$: $2^{2n} - 1$ is divisible by 3

1. **Base Case:**
   - $P(1)$: $2^2 - 1 = 3$ is divisible by 3.
2. **Inductive Step:**
   - Assume $P(k)$ is True for some $k \geq 1$, thus $2^{2k} - 1 \bmod 3 = 0$
   - Show that $P(k+1)$ is True:

$$2^{2(k+1)} - 1 \bmod 3 = 2^{2k+2} - 1 \bmod 3 = 4 \cdot 2^{2k} - 1 \bmod 3$$
$$= 3 \cdot 2^{2k} + 2^{2k} - 1 \bmod 3 = 0$$

$\square$

Example 1: Partial Correctness of Loops

Prove the partial correctness of the following algorithm.

Precondition: n and m are positive integers

Postcondition: n and m are unchanged and $p = n \times m$

```
Prod(m, n)
  i = 0
  p = 0
  while i < n do
    i = i + 1
    p = p + m
  end while
```


Example 1, Continued

Claim: Prod(n,m) is partially correct

Proof.

Proof that $0 \leq i \leq n$ and $p = i \times m$ is a loop invariant:

1. True before first iteration: $i = 0$ and $p = 0$
2. If True before an iteration and $i < n$ then True after the iteration:
   - Before the iteration: $0 \leq i_{before} < n$ and $p_{before} = i_{before} \times m$
   - After the iteration: $i_{after} = i_{before} + 1 \Rightarrow 0 \leq i_{after} \leq n$ and $p_{after} = p_{before} + m = i_{before} \times m + m = i_{after} \times m$

Proof of partial correctness:

- preconditions and initial assignment statements imply the loop invariant (trivially)
- upon termination: $i = n$ and $p = n \times m \Rightarrow Q$

$\square$

Example 2: Partial Correctness of Loops

Prove the correctness of the following algorithm.

Precondition: n is a positive integer

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

```
Sum(n)
  i = 1
  s = 1
  while i < n do
    i = i + 1
    s = s + i
  end while
```

Example 2, Continued

Claim: `Sum(n)` is partially correct

Proof.

Proof that $1 \le i \le n$ and $s = \sum_{j=1}^{i} j$ is a loop invariant:

1. True before first iteration: $i = 1$ and $s = 1$
2. If True before an iteration and $i < n$ then True after the iteration:
   - Before the iteration: $1 \le i_{before} < n$, and $s_{before} = \sum_{j=1}^{i_{before}} j$
   - After the iteration: $i_{after} = i_{before} + 1 \Rightarrow 1 \le i_{after} \le n$ and
     $$s_{after} = s_{before} + i_{after} = \sum_{j=1}^{i_{before}} j + i_{after} = \sum_{j=1}^{i_{after}} j$$

Proof of partial correctness: similar to before

- upon termination: $i = n$ and $s = \sum_{j=1}^{n} j \Rightarrow Q$ □

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 $\Rightarrow$ Total 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* totally correct. . . but something interesting can still be established.

Termination of Loops

Problem: Show that if the precondition P is satisfied and the loop

`while G 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.
   (Proof technique is based on the concept of a *Loop Variant*.)

Termination of Loops, Continued

Definition: A *loop variant* for a loop

`while G do` S `end while`

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

1. The value of f_L is decreased *by at least one* every time the loop body S is executed
2. If the value of f_L is less than or equal to zero then the loop guard `G` is `False` (ie., the loop terminates)

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

Termination of Loops, Continued

Problem: Prove that if the precondition P is satisfied and the loop

```
while G do S end while
```

is executed, then the loop eventually terminates.

Solution:

1. Show that if the loop invariant is satisfied and the loop body is executed then the loop body terminates
2. Identify a loop variant f_L:
   - f_L is an integer valued function
   - The value of f_L is decreased *by at least one* every time the loop body is executed
   - If the value of f_L is less than or equal to zero then the loop guard is `False`

Example 1: Termination of Loops

Claim: `Prod(m, n)` terminates.

Proof.

1. Loop body always terminates
2. Loop variant: $f(n,i) = n - i$
   - $f(n,i)$ is an integer valued function
   - after every iteration, i increases by 1 and thus $f(n,i)$ decreases by 1
   - if $f(n,i) \leq 0$ then $i \geq n$ and the loop terminates (number of iterations $= f(n,0) = n$) □

Example 2: Termination of Loops

Claim: `Sum(n)` terminates.

Proof.

1. Loop body always terminates
2. Loop variant: $f(n,i) = n - i$
   - $f(n,i)$ is an integer valued function
   - after every iteration, i increases by 1 and thus $f(n,i)$ decreases by 1
   - if $f(n,i) \leq 0$ then $i \geq n$ and the loop terminates (number of iterations $= f(n,1) = n - 1$) □

Correctness of Recursive Algorithms

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 *strong mathematical induction*, proceeding by induction on the "size" of the inputs.

- **Base Case:** base cases of the recursive algorithm
- **Inductive Step:** algorithm is correct for all inputs of size "up to" n, show that it is correct for inputs of size $n + 1$

Proof proceeds by proving correctness while assuming the induction hypothesis (i.e., every recursive call returns the correct output).

Strong Mathematical Induction

Problem: For all integers $k \geq k_0$, prove that property $P(k)$ is True.

Proof by Strong form of Induction:

1. **Base Case:** Show that $P(k_0)$ is True
2. **Inductive Step:** Show that if $P(i)$ is True for **all** integers $k_0 \leq i \leq k$ then $P(k+1)$ is True.
   - choose an arbitrary $k \geq k_0$
   - show that $P(k+1)$ is True if $P(k), P(k-1), \ldots, P(k_0)$ are True

Example: Partial Correctness of Recursive Algorithms

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

```
long Fib(i)
  if i == 0 then
    return 1
  end if
  if i == 1 then
    return 1
  end if
  return Fib(i-1) + Fib(i-2)
```


Example, Continued

Claim: `Fib(i)` is partially correct.

Proof.

1. **Base Case:** The algorithm is partially correct for $i = 0$ and $i = 1$
2. **Inductive Step:** Assume that $Fib(i)$ for $i = 0, 1, \ldots, k$ $(k \geq 1)$ returns the i-th Fibonacci number denoted by F_i. Show that $Fib(k+1)$ returns the $(k+1)$-th Fibonacci number, F_{k+1}.

   Since $k+1 > 1$, we have:

   $$Fib(k+1) = Fib(k) + Fib(k-1)$$

   Using the induction hypothesis, it follows that

   $$Fib(k+1) = F_k + F_{k-1} = F_{k+1}$$ □

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! 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

Recommended References:

- Susanna S. Epp
  Discrete Mathematics with Applications, Third Edition
  See Section 4.5
- Michael Soltys
  An Introduction to the Analysis of Algorithms
  Chapter 1 contains an introduction to proofs of correctness and is freely available online!