22c:111 Programming Language Concepts

Fall 2008

Program Correctness I

These notes were originally developed by Allen Tucker, Robert Noonan and modified by Cesare Tinelli. They are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.
Program Correctness

18.1 Axiomatic Semantics
18.1.1 Fundamental Concepts
18.1.2 The Assignment Rule
18.1.3 Rules of Consequence
18.1.4 Correctness of the Max Function
18.1.5 Correctness of Programs with Loops

18.2 Formal Methods Tools: JML

18.3 Correctness of Object-Oriented Programs

18.4 Correctness of Functional Programs
Motivation

A correct program is one that does exactly what it is intended to do, no more and no less.

A formally correct program is one whose correctness can be proved mathematically.

- This requires a language for specifying precisely what the program is intended to do.
- Specification languages are based in mathematical logic.
- Hoare invented “axiomatic semantics” in 1969 as a tool for specifying program behavior and proving correctness.

Until recently, correctness has been an academic exercise.

- Now it is a key element of critical software systems.
Correctness Tools

- Theorem provers
  - PVS
- Modeling languages
  - UML and OCL
- Specification languages
  - JML
- Programming language support
  - Eiffel
  - Java
  - Spark/Ada
- Specification Methodology
  - Design by contract
18.1 Axiomatic Semantics

Axiomatic semantics is a language for specifying what a program is supposed to do.

Based on the idea of an assertion:

- An assertion is a predicate that describes the state of a program at a point in its execution.

A postcondition is an assertion that states the program’s result.
A precondition is an assertion that states what must be true before the program begins running.
A "Hoare Triple" has the form \{P\} s \{Q\}

\{true\}
int Max (int a, int b) {
    int m;
    if (a >= b)
        m = a;
    else
        m = b;
    fi
    return m;
}
\{m = \text{max}(a, b)\}

Precondition \(P\): there are no constraints on the input for this particular function.

Program body \(s\)

Postcondition \(Q\): max is the mathematical idea of a maximum.
Partial correctness

There is no guarantee that an arbitrary program will terminate normally. That is, for some inputs,

- It may enter an infinite loop, or
- It may fail to complete its calculations.

E.g., consider a C-like factorial function $n!$ whose parameter $n$ and result are int values. Passing 21 as an argument should return $n! = 51090942171709440000$. But that value cannot be stored as an int, so the function fails.

A program $s$ is *partially correct* for pre- and postconditions $P$ and $Q$ if, whenever $s$ begins in a state that satisfies $P$, it terminates in state that satisfies $Q$. 
Proving Partial Correctness

Program $s$ is *partially correct* for pre- and postconditions $P$ and $Q$ if the Hoare triple $\{P\} s \{Q\}$ is *valid*.

There are seven *rules of inference* that can be used to prove the validity of $\{P\} s \{Q\}$:

1. The Assignment Rule
2. The Sequence Rule
3. The Skip Rule
4. The Conditional Rule
5. The Loop Rule
6. Precondition Consequence Rule
7. Postcondition Consequence Rule

- **Used for basic statement types**
- **Used to simplify intermediate triples**
Proof Methodology

1. A proof is naturally represented as a proof tree.
2. The proof starts with this triple \{P\}s\{Q\}. E.g.,
   \{true\} if (a \geq b) m = a; else m = b; fi\{m = \text{max}(a, b)\}
3. An inference is written as \(\frac{p, q}{r}\), and means
   “if \(p\) and \(q\) are valid, then \(r\) is inferred to be valid.”
4. Using appropriate rules of inference, break the triple
   into a group of inferences in which:
   1. Each triple is individually valid, and
   2. The inferences (logically) combine to form a tree whose root
      is the program’s original Hoare triple.
### The Assignment Rule

\[
\begin{align*}
\text{true} & \\
\{Q[v \leftarrow e]\} & \quad v = e \quad \{Q\}
\end{align*}
\]

If \( Q \) is a postcondition of an assignment, then replacing all occurrences of \( v \) in \( Q \) by \( e \) is a valid precondition.

**Examples:**

\[
\begin{align*}
v &= e\{Q\} & Q[v \leftarrow e] \\
\{?\} \quad x &= 1 \quad \{x = 1 \land y = 4\} & \{1 = 1 \land y = 4\} \\
\{?\} \quad m &= a \quad \{m = \max(a,b)\} & \{\max(a,b) = a\} \\
\{?\} \quad i &= i + 1 \quad \{0 \leq i \land i < n\} & \{0 \leq i + 1 \land i + 1 < n\} \\
\{?\} \quad i &= i + 1 \quad \{f \ast i = i!\} & \{f \ast (i + 1) = (i + 1)!\}
\end{align*}
\]
The Conditional Rule

We can infer \textbf{this} when we reason backwards from \textbf{here}.

\[
\begin{align*}
\{test \land P\} s_1 \{Q\}, & \quad \{\neg test \land P\} s_2 \{Q\} \\
\{P\} \text{ if (test) } s_1 \text{ else } s_2 \{Q\}
\end{align*}
\]

\[
\{a > b \land true\} m = a; \quad \{m = \text{max}(a,b)\}, \quad \{(a > b) \land true\} m = b; \quad \{m = \text{max}(a,b)\}
\]

\[
\{true\} \text{ if (a > b) } m = a; \text{ else } m = b; \quad \{m = \text{max}(a,b)\}
\]

E.g.,
Rules of Consequence

Precondition strengthening: \[ P \supset P', \{ P' \} s \{ Q \} \]
\[ \{ P \} s \{ Q \} \]

Postcondition weakening: \[ \{ P \} s \{ Q' \}, \quad Q' \supset Q \]
\[ \{ P \} s \{ Q \} \]

E.g.,

\[ a > b \supset a = \max(a,b), \quad \{ a = \max(a,b) \} \quad m = a; \quad \{ m = \max(a,b) \} \]
\[ \{ a > b \} \quad m = a; \quad \{ m = \max(a,b) \} \]
Correctness of the Max Function

Its proof tree has the following form:

\[
\frac{P_1 \quad P_3}{P_2}, \quad \frac{P_4}{\{true\} \text{ if } (a > b) \ m = a; \ else \ m = b; \{m = \max(a, b)\}}
\]

where

\[P_1 = a > b \land true \supset a = \max(a, b), \{a = \max(a, b)\} \ m = a; \{m = \max(a, b)\}\]
\[P_2 = \{a > b \land true\} \ m = a; \{m = \max(a, b)\}\]
\[P_3 = \neg(a > b) \land true \supset b = \max(a, b), \{b = \max(a, b)\} \ m = b; \{m = \max(a, b)\}\]
\[P_4 = \neg(a > b) \land true \ m = b; \{m = \max(a, b)\}\]

Note: the assignment rule, the precondition strengthening rule, and the conditional rule are all used in this proof.
The Sequence Rule

\[ \{P\} s_1 \{R\}, \quad \{R\} s_2 \{Q\} \]
\[ \{P\} s_1; s_2 \{Q\} \]

Here, the challenge is to find an \( R \) that will allow us to break a triple into two valid triples.

E.g.,

\[ \{i < n \land 1 \leq i \leq n \land f = i!\} \quad i = i + 1; \quad \{R\}, \quad \{R\} \quad f = f \times i; \quad \{1 \leq i \land i \leq n \land f = i!\} \]
\[ \{i < n \land 1 \leq i \land i \leq n \land f = i!\} \quad i = i + 1; \quad f = f \times i; \quad \{1 \leq i \land i \leq n \land f = i!\} \]

where

\[ R = \{1 \leq i \land i \leq n \land f \times i = i!\} \]

Notes:
1. The \textit{second triple} above the line is valid because of the assignment rule.
2. The \textit{first triple} is valid because both \( f \times i = i! \supset f \times (i + 1) = (i + 1)! \) and \( \{i + 1 \leq n\} \supset \{i < n\} \quad i = i + 1; \quad \{i \leq n\} \) are valid, using precondition strengthening and the assignment rule.
The Loop Rule

\[
\{test \land R\} s_1 \{R\}, \quad \neg test \land R \supseteq Q \quad \frac{}{\{R\} \text{ while (test)} s_1 \{Q\}}
\]

R is called the loop invariant.

The loop invariant remains true before and after each iteration. E.g., in

\[
\{R\} \text{ while } (i < n) \ i = i + 1; \ f = f \ast i; \ \{f = n!\}
\]

\[
R = 1 \leq i \land i \leq n \land f = i!
\]

is a good choice for a loop invariant.

Note: when the test \(i < n\) finally becomes false, the only value for \(i\) that satisfies \(R\) is \(i = n\), justifying the postcondition \(\{f = n!\}\)
Correctness of Programs with Loops

Consider the following triple for a factorial calculation:

\[
\{1 \leq n\}
\]

\[
\text{int } f = 1;
\]

\[
\text{int } i = 1;
\]

\[
\text{while } (i < n) \{ \\
\quad i = i + 1; \\
\quad f = f \times i;
\%
\}
\]

\[
\{f = n!\}
\]

Below is a sketch of its correctness proof, as two proof trees:

– part 1 for the first two statements, and
– part 2 for the loop.
Proof tree for factorial (part 1 of 2)

\[ 1 \leq n \supset P_1 \quad , \quad \{ P_1 \} \ f =1; \ \{ P_2 \} \quad , \quad \{ P_2 \} \ i =1; \ \{ 1 \leq i \land i \leq n \land f = i! \} \]

\[ \{ 1 \leq n \} \ f =1; \ \{ P_2 \} \quad , \quad \{ 1 \leq i \land i \leq n \land f = i! \} \]

where

\[ P_1 = 1 \leq 1 \land 1 \leq n \land 1 = 1! \]

\[ P_2 = 1 \leq 1 \land 1 \leq n \land f = 1! \]

This part uses precondition strengthening, the assignment rule, and the sequence rule.

It also establishes the loop invariant as: \( R = 1 \leq i \land i \leq n \land f = i! \)
Proof tree for factorial (part 2 of 2)

\[
\{i < n \land R\} \quad i = i + 1; \; f = f \cdot i; \; \{R\} \quad , \quad \neg(i < n) \land R \supset f = n!
\]

\[\{R\} \text{ while } (i < n) \quad \{i = i + 1; \; f = f \cdot i; \} \quad \{f = n!\}\]

where

\[R = 1 \leq i \land i \leq n \land f = i!\]

Notes:

1. The left-hand premise was proved earlier.
2. The right-hand premise is true when the loop terminates. *Its validity can be shown mathematically.* I.e., if \(i \leq n\) and \(\neg(i < n)\) then \(i = n\). But if \(f = i!\) and \(i = n\), then \(f = n!\)
3. The conclusion follows from the loop rule.
Perspectives on Formal Methods

• Theory developed in the ‘60s and ‘70s
• Effectively used to verify hardware design
• Not widely used in software design, but
  – New tools are emerging (e.g., JML, Spark/Ada)
  – Techniques have been effective in some critical systems (e.g., the Paris metro system).
• Many software designers reject formal methods:
  – Too complex for programmers
  – Time better spent with alternative testing methods.
• Current research in FM is shooting for a middle ground.