Program Verification

Notes by Jonathan Buss
Based in part on materials prepared by B. Bonakdarpour
from Huth & Ryan text,
and material by Anna Lubiw
Additional thanks to D. Maftuleac, R. Trefler, and P. Van Beek
Modified by Lila Kari
Outline

- Introduction: What and Why?
- Pre- and Post-conditions
- Conditionals
- while-Loops and Total Correctness
Reference: Huth & Ryan, Chapter 4

**Program correctness**: does a given program satisfy its specification—does it do what it is supposed to do?

**Techniques for showing program correctness:**
- **inspection**, code walk-throughs
- **testing**
  - black box: tests designed independent of code
  - white box: tests designed based on code
- **formal verification**
"Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

[E. Dijkstra, 1972.]

Testing is not proof!
Testing versus Formal Verification

- **Testing:**
  - check a program for carefully chosen inputs (e.g., boundary conditions, etc.)
  - in general: cannot be exhaustive

- **Formal verification:**
  - formally state the specification of a problem (using the formalism of logic), and
  - **prove** that a program satisfies the specification for all inputs
    - although undecidable (= no algorithm) in general, we will study some useful techniques
    - part of *Software Engineering*
Why formally specify and verify programs?

- Reduce bugs
- Safety-critical software or important components (e.g., brakes in cars, nuclear power plants)
- Documentation
  - necessary for large multi-person, multi-year software projects
  - good documentation facilitates code re-use
- Current Practice
  - specifying software is widespread practice
  - formally verifying software is less widespread
  - hardware verification is common

Why Formal Verification?

Reduce bugs

Safety-critical software or important components (e.g., brakes in cars, nuclear power plants)

Documentation
  - necessary for large multi-person, multi-year software projects
  - good documentation facilitates code re-use

Current Practice
  - specifying software is widespread practice
  - formally verifying software is less widespread
  - hardware verification is common
Some Software Bugs

- **Therac-25, X-ray, 1985**
  - overdosing patients during radiation treatment, 5 dead
  - *reason*: race condition between concurrent tasks

- **AT&T, 1990**
  - long distance service fails for 9 hours
  - *reason*: wrong BREAK statement in C code

- **Patriot-Scud, 1991**
  - 28 dead and 100 injured
  - *reason*: rounding error

- **Pentium Processor, 1994**
  - error in division algorithm
  - *reason*: incomplete entries in a look-up table
Some Software Bugs

- **Ariane 5, 1996**
  - exploded 37 seconds after takeoff
  - *reason*: data conversion of a too large number

- **Mars Climate Orbiter, 1999**
  - destroyed on entering atmosphere of Mars
  - *reason*: mixing up pounds with kilograms

- **Power black-out, 2003**
  - 50 million people in Canada and US without power
  - *reason*: programming error

- **Royal Bank, 2004**
  - financial transactions disrupted for 5 days
  - *reason*: programming error
Some Software Bugs

- **UK Child Support Agency, 2004**
  - overpaid 1.9 million people, underpaid 700,000, cost to taxpayers over $1 billion
  - *reason*: more than 500 bugs reported

- **Science (a prestigious scientific journal), 2006**
  - retraction of research papers due to erroneous research results
  - *reason*: program incorrectly flipped the sign (+ to -) on data

- **Toyota Prius, 2007**
  - 160,000 hybrid vehicles recalled due to stalling unexpectedly
  - *reason*: programming error

- **Knight Capital Group, 2012**
  - high-frequency trading system lost $440 million in 30 min
  - *reason*: programming error
Framework for software verification

The steps of formal verification:

1. Convert the informal description $R$ of requirements for an application domain into an “equivalent” formula $\Phi_R$ of some symbolic logic,

2. Write a program $P$ which is meant to realise $\Phi_R$ in some given programming environment, and

3. Prove that the program $P$ satisfies the formula $\Phi_R$.

We shall consider only the third part in this course.

Note that Step 1, of finding a suitable formalization $\Phi_R$ of $R$ (translation from English into logic) is very important; otherwise it is always possible that $\Phi_R$ specifies behaviour which is different from the one described in $R$. 
We shall use a subset of C/C++ and Java. It contains their core features:

- integer and Boolean expressions
- assignment
- sequence
- if-then-else (conditional statements)
- while-loops
- for-loops
- arrays
- functions and procedures
We are verifying imperative, sequential, transformational programs.

- **imperative**: sequence of commands which modify the values of variables
- **sequential**: no concurrency
- **transformational**: given inputs, compute outputs and terminate
Imperative programs

- Imperative programs manipulate variables.
- The **state** of a program consists of a vector of the **values of all the variables** at a particular time in the execution of the program.
- Expressions evaluate relative to the current state of the program.
- Commands change the state of the program.
Example

We shall use the following code as an example.

Compute the factorial of input $x$ and store in $y$.

```java
y = 1;
z = 0;
\rightarrow \text{while } (z \neq x) \{ 
    z = z + 1;
    y = y * z;
\}
```

State at the “while” test:
- Initial state $s_0$: $z=0$, $y=1$
- Next state $s_1$: $z=1$, $y=1$
- State $s_2$: $z=2$, $y=2$
- State $s_3$: $z=3$, $y=6$
- State $s_4$: $z=4$, $y=24$
  
  ...

Note: the order of “$z = z + 1$” and “$y = y * z$” matters!
We shall use the following code as an example.

Compute the factorial of input $x$ and store in $y$.

```
y = 1;
z = 0;
→ while (z != x) {
    z = z + 1;
y = y * z;
}
```

**State** at the “while” test:
- Initial state $s_0$: $z=0$, $y=1$
- Next state $s_1$: $z=1$, $y=1$
- State $s_2$: $z=2$, $y=2$
- State $s_3$: $z=3$, $y=6$
- State $s_4$: $z=4$, $y=24$
  ...

Note: the order of “$z = z + 1$” and “$y = y * z$” matters!
Informal Requirement $R$:

Compute a number $y$ whose square is less than the input $x$.

An appropriate specification for it may be $y \cdot y < x$.

But what if the input $x$ is $-4$?

It is not possible to write a program that will work for all inputs.

Revised informal requirement $R$:

If the input $x$ is a positive number, compute a number whose square is less than $x$.

This means that we need information not just about the state after the program executes, but also about the state before it executes.
Hoare Triples

Our assertions about programs will have the form

\[ (P) \quad \text{— precondition} \]
\[ C \quad \text{— program or code} \]
\[ (Q) \quad \text{— postcondition} \]

The meaning of the triple \((P)\ C\ (Q)\)

If program \(C\) is run starting in a state that satisfies the formula \(P\), then the resulting state after the execution of \(C\) will satisfy the formula \(Q\).

An assertion \((P)\ C\ (Q)\) is called a Hoare triple.
Conditions $P$ and $Q$ are written in predicate logic of integers.

Use predicates $<, =, \text{ functions } +, -, \ast$ and others derivable from these.

Tony Hoare (C.A. R. Hoare), b. 1934

famous for Quicksort and program verification.
A *specification* of a program $C$ is a Hoare triple with $C$ as the second component: $(\implies P \land C \land Q)$.

**Example.** The requirement

If the input $x$ is a positive number, compute a number whose square is less than $x$

might now be expressed as the Hoare triple

$$(\implies x > 0 \land C \land y \cdot y < x)$$.

Often we do not want to put any constraints on the initial state. In that case, the precondition can be set to *true*, which is a formula which is true in any state.
Specification Is Not Behaviour

A triple (\(|x > 0|\) C \(|y \cdot y < x|\)) specifies neither a unique program \(C\) nor a unique behaviour; E.g., both \(C_1\) and \(C_2\) satisfy the specifications:

\begin{align*}
C_1: & \quad y = 0 ; \\
C_2: & \quad y = 0 ; \\
& \quad \text{while} \ (y \cdot y < x) \{ \\
& \quad \quad y = y + 1 ; \\
& \quad \} \\
& \quad y = y - 1 ;
\end{align*}

\(C_2\) has a different behaviour from \(C_1\); For example, if \(x = 22\), then \(C_1\) outputs \(y = 0\), while \(C_2\) outputs \(y = 4\).

To avoid programs like \(C_1\) from being solutions, a better postcondition would be \((y \cdot y < x) \land \forall z ((z \cdot z < x) \rightarrow z \leq y)\)
We want to develop a notion of proof that will allow us to prove that a program $C$ satisfies the specification given by the precondition $P$ and the postcondition $Q$.

The proof calculus is different from the proof calculus in first-order (predicate) logic, since it is about proving triples, which are built from two different kinds of things:

- logical formulas: $P$, $Q$, and
- code $C$
Partial correctness

Definition. A Hoare triple $(\| P \|) C (\| Q \|)$ is satisfied under partial correctness, denoted

\[ \models_{\text{par}} (\| P \|) C (\| Q \|), \]

if and only if

for every state $s$ that satisfies condition $P$,

if the execution of $C$ starting from state $s$ terminates in a state $s'$,

then the state $s'$ satisfies condition $Q$.

(Alternatively, if the above conditions are satisfied, we say that the program $C$ satisfies the specification $(\| P \|) C (\| Q \|)$ under partial correctness.)
In particular, the program

```c
while true {
    x = 0;
}
```

satisfies all specifications!

It is an endless loop and never terminates, but partial correctness only says what must happen if the program terminates.
**Total correctness**

**Definition.** A Hoare triple \((P \mid C \mid Q)\) is **satisfied under total correctness**, denoted

\[ \models_{\text{tot}} (P \mid C \mid Q), \]

if and only if

for every state \(s\) that satisfies \(P\),

execution of \(C\) starting from state \(s\) terminates,

and the resulting state \(s'\) satisfies \(Q\).

(Alternatively, if the above conditions are satisfied, we say that the program \(C\) satisfies the specification \((P \mid C \mid Q)\) under total correctness.)

**Total Correctness = Partial Correctness + Termination**
Example 1.

\[ (\{ x = 1 \} ) \]
\[ y = x ; \]
\[ (\{ y = 1 \} ) \]

This Hoare triple is satisfied under total correctness.

Example 2.

\[ (\{ x = 1 \} ) \]
\[ y = x ; \]
\[ (\{ y = 2 \} ) \]

This Hoare triple is satisfied under neither total nor partial correctness.
Example 3.

\[ (x = 1) \]
while (true) {
    x = 0 ;
}
\[ (x > 0) \]

The program is an infinite loop, so this Hoare triple is satisfied under partial correctness.
Example 4.

\[ (\{ x \geq 0 \} ) \]
\[ y = 1 ; \]
\[ z = 0 ; \]
\[ \text{while} \ (z \neq x) \ { \{ } \]
\[ \quad z = z + 1 ; \]
\[ \quad y = y \times z ; \]
\[ { \} } \]
\[ (\{ y = x! \}) \]

This Hoare triple is satisfied under total correctness.

What happens if we remove pre-condition (replace with “true”)?

The new Hoare triple would be satisfied under partial correctness, but not under total correctness (C loops forever on negative input).
Example 5.

\[ (x \geq 0) \]
\[ y = 1 ; \]
\[ \text{while } (x \neq 0) \{ \]
\[ \quad y = y \times x ; \]
\[ \quad x = x - 1 ; \]
\[ \} \]
\[ (y = x!) \]

This Hoare triple is satisfied under neither partial nor total correctness because the input altered (“consumed”).
Partial correctness is really weak

Give a program that is partially satisfied under partial correctness for any pre- and post-conditions

\[
(P) \quad \text{while (true)} \{ \\
\quad x = 0 \\
\} \quad (Q)
\]

The program never terminates so the partial correctness is vacuously true.
Partial correctness is really weak

At the other extreme, consider

\[(| true |) C (| true |)\]

Suppose

- \(C\) never terminates \(\implies\) \(C\) satisfies the specification under partial correctness
- \(C\) sometimes terminates \(\implies\) \(C\) satisfies the specification under partial correctness
- \(C\) always terminates \(\implies\) \(C\) satisfies the specification under total correctness
Sometimes in our specifications (pre- and post- conditions) we will need additional variables that do not appear in the program.

These are called **logical variables**.

**Example.**

\[
\begin{align*}
(x = x_0 \land x_0 &\geq 0) \\
y &= 1; \\
\text{while } (x \neq 0) \{ \\
\quad y &= y \times x; \\
\quad x &= x - 1; \\
\} \\
(y = x_0! &)
\end{align*}
\]

For a Hoare triple, its set of logical variables are those variables that are free in \( P \) or \( Q \) and do not occur in \( C \).
We can write the pre- and postconditions for partial and total correctness in predicate logic:

- **States(s)** - **Predicate**: “s is an element of the set of states”
- **Satisfies(s, P)** - **Predicate**: “State s satisfies condition P”
- **Terminates(C, s)**: **Predicate**: “code C terminates when execution begins in state s”
- **result(C, s)**: **function**: the state that results from executing code C beginning in state s, if C terminates (undefined otherwise)
Partial and Total Correctness in Logic

- **Partial correctness** of Hoare triple $(P)\ C\ (Q)$:
  \[
  \forall s [\text{States}(s) \rightarrow (\text{Satisfies}(s, P) \land \text{Terminates}(C, s) \rightarrow \text{Satisfies}(\text{result}(C, s), Q))]
  \]

- **Total correctness** of Hoare triple
  \[
  \forall s [\text{States}(s) \rightarrow (\text{Satisfies}(s, P) \rightarrow \text{Terminates}(C, s) \land \text{Satisfies}(\text{result}(C, s), Q))]
  \]
Total correctness is our goal.

We usually prove it by proving partial correctness and termination separately.

- For partial correctness, we shall introduce sound inference rules.
- Proving termination is often easy, but not always (in general, it is undecidable)
Why do we separate into partial/total correctness?

Both are undecidable, i.e., there is no algorithm to solve them.

There are different techniques for partial and total correctness.

We will look at a proof system for proving partial correctness.
Proving Partial Correctness

Recall the definition of Partial Correctness:

For every starting state which satisfies $P$ and for which $C$ terminates, the final state satisfies $Q$.

How do we show this, if there are a large or infinite number of possible states?

Answer: **Inference rules** (proof rules, like in formal deduction)

Rules for each construct in our programming language.
What will a “Formal Proof” for partial correctness look like?

An annotated program with conditions before and after every program statement. Each Hoare triple (condition, program statement, condition) will have a justification.

```
( precondition  )
y = 1;
( ... )
while (x != 0) {
( ... )
y = y * x;
( ... )
x = x - 1;
( ... )
}
( postcondition )
```
Inference Rule for Assignment

$$\frac{\left( Q[E/x] \right)}{x = E \left( \neg Q \right)}$$

(assignment)

Intuition:

$Q(x)$ will hold after assigning (the value of) $E$ to $x$ if $Q(E)$ was true initially.

Note: Normally, $Q$ will be a formula with variable $x$ in it, $Q(x)$;
Example.

\[ \vdash_{\text{par}} (y + 1 = 7) \ x = y + 1 \ (x = 7) \]

by one application of the assignment rule.

- The Assignment rule is best applied backwards: The right way to understand it is to think about what we would have to prove about the initial state, in order to prove that \( Q \) holds in the resulting state.
- Since \( Q \) will in general be a function of \( x \), whatever it says about \( x \) must have been true for \( E \), since in the resulting state the value of \( x \) is \( E \).
- Thus, \( Q \) with \( E \) in place of \( x \) must be true of the initial state.
Instances of the rule *Assignment*

Example 1.

\[
\begin{align*}
(| y = 2 |) & \quad (| Q[E/x] |) \\
x = y & \quad x = E; \\
(| x = 2 |) & \quad (| Q |)
\end{align*}
\]

If we want to prove \( x = 2 \) after the assignment \( x = y \), then we must be able to prove that \( y = 2 \) before it. Here \( Q(x) \) is “\( x = 2 \)”, \( E = y \), \( Q[y/x] \) is “\( y = 2 \)”.

Example 2.

\[
\begin{align*}
(| 0 < 2 |) & \quad (| Q[E/x] |) \\
x = 2 & \quad x = E; \\
(| 0 < x |) & \quad (| Q |)
\end{align*}
\]

If we want to prove that \( 0 < x \) after the assignment \( x = 2 \), we must be able to prove \( 0 < 2 \) before it. Here \( Q(x) \) is “\( 0 < x \)”, \( E = 2 \), \( Q[2/x] \) is “\( 0 < 2 \)”. 
Instance of the rule Assignment

Example 3.

\[(x + 1 = 2) \leftrightarrow ((x = 2)[(x + 1)/x])\]
\[x = x + 1; \quad x = x + 1;\]
\[(x = 2) \leftrightarrow (x = 2)\]

If we want to prove that \(x = 2\) after the assignment \(x = x + 1\), we must be able to prove \(x + 1 = 2\) before it. Here \(Q(x)\) is “\(x = 2\)”, \(E = x + 1\)

Example 4.

\[(x + 1 = n + 1)\]
\[x = x + 1;\]
\[(x = n + 1)\]

If we want to prove that \(x = n + 1\) after the assignment \(x = x + 1\), we must be able to prove \(x + 1 = n + 1\) before it. Here \(Q(x)\) is “\(x = n + 1\)”, \(E = x + 1\)
In program correctness proofs, we usually work backwards from the postcondition:

```
??                 (\( P[E/x] \) )
x = y;             x = E;
(\( x > 0 \) )      (\( P \) )
```
Inference Rules about Implications

Precondition strengthening:

\[
\frac{P \rightarrow P'}{(P') \mid C \mid Q)} (\text{implied})
\]

Postcondition weakening:

\[
\frac{(P) \mid C \mid Q'\mid Q'}{(P') \mid C \mid Q)} (\text{implied})
\]

The *Implied* rules allow us to import proofs in predicate logic, \( P \rightarrow P' \), \( Q' \rightarrow Q \), (enhanced with basic facts of arithmetic) into the proofs in program logic.
Example of using the *Implied* rule

\[
P \rightarrow P' \quad (P') \quad C \quad (Q) \quad (implied)
\]

Show that the program “\( x = y + 1 \)’’ satisfies the specification \((y = 6) \quad x = y + 1 \quad (x = 7)\) under partial correctness.

\((y = 6)\)
\((y + 1 = 7)\) implied
\(x = y + 1\)
\((x = 7)\) assignment

Here: \(P\) is \(y = 6\), \(P'\) is \(y + 1 = 7\), \(C\) is \(x = y + 1\), \(Q\) is \(x = 7\)

Note that here \(P \leftrightarrow P'\)
Example of using the *Implied* rule

\[
\frac{(P) \quad C \quad (Q')}{(P) \quad C \quad (Q)} \quad Q' \rightarrow Q \quad (implied)
\]

Show that the program “\(x = y + 1\)” satisfies the specification \((y + 1 = 7) \land x = y + 1 \land x \leq 7\) under partial correctness.

\[
(y + 1 = 7) \quad x = y+1 \\
(x = 7) \quad (x \leq 7) \quad \text{implied}
\]

Here: \(P\) is \(y + 1 = 7\), \(C\) is \(x = y + 1\), \(Q'\) is \(x = 7\), \(Q\) is \(x \leq 7\).

In this case, \(Q' \rightarrow Q\) but the converse is not true.
Inference Rule for Sequences of Instructions

\[
\frac{(P) \quad C_1 \quad (Q), \quad (Q) \quad C_2 \quad (R)}{(P) \quad C_1; \ C_2 \quad (R)} \quad \text{(composition)}
\]

In order to prove \((P) \quad C_1; \ C_2 \quad (R)\), we need to find a midcondition \(Q\) for which we can prove \((P) \quad C_1 \quad (Q)\) and \((Q) \quad C_2 \quad (R)\).

(In our examples, the mid-condition will usually be determined by a rule, such as assignment. But in general, a mid-condition might be very difficult to determine.)
Inference rules with sequence of instructions allow us to string together pre/postconditions and lines of code.

Each condition is the **postcondition of the previous line of code** and the **precondition of the next line of code**.
Interleave program statements with assertions (= conditions), each justified by an inference rule.

The composition rule is implicit.

Each assertion should hold whenever the program reaches that point in its execution.

Each assertion is justified by an inference rule
- If implied inference rule is used, we also need to prove the implication. This is done after annotating the program.
- don’t simplify assertions in the annotated program. Do them as implied inferences.
Example: Composition of Assignments

To show that the following Hoare triple (whose program has 3 lines of code) is satisfied under partial correctness.

We work bottom-up for assignments...

\[
\begin{align*}
( & x = x_0 \land y = y_0 ) \\
( & y = y_0 \land x = x_0 ) \\
( & t = x ) \\
( & y = y_0 \land t = x_0 ) \\
( & x = y ) \\
( & x = y_0 \land t = x_0 ) \\
( & y = t ) \\
( & x = y_0 \land y = x_0 )
\end{align*}
\]

Finally, show \( ( x = x_0 \land y = y_0 ) \) implies \( ( y = y_0 \land x = x_0 ) \).
Show that the program \( x = y + 1 \) satisfies the specification 
\((y = 5)\) \( x = y + 1 \) \((x = 6)\) under partial correctness.

\((y = 5)\) 
\((y + 1 = 6)\) implied 
x = y + 1;
\((x = 6)\) assignment

- The proof is constructed from the **bottom upwards**
- We start with \( x = 6 \) and, using the **assignment** rule, we push it upwards through (the assignment) \( x = y + 1 \)
- This means substituting \( y + 1 \) for all occurrences of \( x \), resulting in \( y + 1 = 6 \)
- Now compare this with the given precondition \( y = 5 \).
- The given precondition and the arithmetic fact that \( 5 + 1 = 6 \) imply it, so we have finished the proof
Although the proof is constructed bottom-up, its justifications make sense when read top-down.

- The 2nd line is implied by the 1st line.
- The 4th line follows from the 2nd, by the intervening assignment $x = y + 1$.

Note that implied always refers to the immediately preceding line.

Proofs in program logic generally combine two logical levels:

- The 1st level is directly concerned with proof rules for programming constructs, such as the assignment statement.
- The 2nd level is ordinary logic derivations (as familiar from propositional and predicate logic) plus facts from arithmetic.
Example 2 and Comments

Show that the program “\(y = y + 1\)” satisfies the specification
\((\lnot y < 3) \land y = y + 1 \land (\lnot y < 4)\) under partial correctness.

\((\lnot y < 3)\)
\((\lnot y + 1 < 4)\) \hspace{1em} \text{implied}
\(y = y+1;\)
\((\lnot y < 4)\) \hspace{1em} \text{assignment}

- We may use ordinary logical and arithmetic implications to change a certain condition \(\varphi\) to any condition \(\varphi'\) which is implied by \(\varphi\) (that is, \(\varphi \rightarrow \varphi'\)) for reasons which have nothing to do with the code.
- Here, \(\varphi\) was \(y < 3\) and the implied formula \(\varphi'\) was \(y + 1 < 4\).
- The validity of this implication is rooted in general facts about integers and the relation \(<\).
- Completely formal proofs would require separate proofs attached to all instances of the rule \textit{implied}.
- We will not always do that.
Programs with Conditional Statements
if-then-else:

\[
\frac{(P \land B)}{\text{if } (B) C_1} \quad \frac{(P \land \neg B)}{(Q)} \quad (P \land \neg B) \rightarrow Q
\]

\[
\frac{(Q)}{\text{else } C_2}
\]

\[
\frac{(Q)}{(P) \text{ if } (B) C_1 \text{ else } C_2}
\]

(if-then-else)

if-then (without else):

\[
\frac{(P \land B)}{\text{if } (B) C} \quad \frac{(P \land \neg B)}{(Q)} \quad (P \land \neg B) \rightarrow Q
\]

\[
\frac{(Q)}{(P) \text{ if } (B) C}
\]

(if-then)
Annotated program template for if-then-else:

\[
\begin{align*}
\{ P \} \\
\text{if } ( B ) \{ \\
\quad ( P \land B ) & \quad \text{if-then-else} \\
\quad C_1 \\
\quad \{ Q \} & \quad (\text{justify depending on } C_1—\text{a “subproof”}) \\
\} \quad \text{else } \{ \\
\quad ( P \land \neg B ) & \quad \text{if-then-else} \\
\quad C_2 \\
\quad \{ Q \} & \quad (\text{justify depending on } C_2—\text{a “subproof”}) \\
\} \\
\{ Q \} & \quad \text{if-then-else [justifies this } Q, \text{ given previous two]}
\end{align*}
\]
Annotated program template for if-then:

\[
\begin{align*}
&\text{if } ( B ) \{ \\
&\quad \text{if-then } ( P \land B ) \\
&\quad \text{if-then } C \\
&\quad \text{[add justification based on } C \text{]} \\
&\}\text{ if-then } ( Q ) \\
&\text{Implied: Proof of } P \land \neg B \rightarrow Q
\end{align*}
\]
Example 1 for conditionals

**Example:** Prove that the program $\text{CODE1}$ below (in blue), satisfies the specifications ($\text{true}$) $\text{CODE1}$ ($\text{max} \geq x$) under partial correctness.

\[
\begin{align*}
\text{\{true\}} & \quad \text{\{P\}} \\
\text{if ( max < x ) \{ & \text{\{Q\}} \\
\quad \text{max} = x ; & \quad \text{C} \\
\} & \\
\text{\{max} \geq x \text{\}} & \\
\end{align*}
\]

First, let's recall our proof method....
The Steps of Creating a Proof

Three steps in doing a proof of partial correctness:

1. First **annotate** using the appropriate inference rules.

2. Then **“back up” in the proof**: add an assertion before each assignment statement, based on the assertion following the assignment.

3. Finally **prove any “implieds”**:
   - Annotations from (1) above containing implications
   - Adjacent assertions created in step (2).

Proofs here can use predicate logic, basic arithmetic, or other appropriate reasoning.
Doing the Steps

1. Add annotations for the if-then statement.
2. “Push up” for the assignments.
3. Identify “implieds” to be proven.

\[
\text{(true)} \\
\text{if ( max < x )} \left\{ \\
\text{(true \land max < x)} \quad \text{if-then} \\
\text{(x \geq x)} \quad \text{Implied (a)} \\
\text{max = x ;} \\
\text{(max \geq x)} \\
\right\}
\]

\[
\text{Implied: } (\text{true \land \neg (max < x)}) \rightarrow max \geq x
\]
The auxiliary “implied” proofs can be done by Natural Deduction (and assuming the necessary arithmetic properties). We will use it informally.

Proof of Implied (a):

\[ \vdash \left( (true \land (max < x)) \right) \rightarrow x \geq x. \]

Clearly \( x \geq x \) is a tautology and the implication holds.
Proof of Implied (b): Show ⊢ (P ∧ ¬B) → Q, which is

\[ \emptyset \vdash (true \land \neg (max < x)) \rightarrow (max \geq x) . \]

1. \( (true \land \neg (max < x)) \vdash (true \land \neg (max < x)) \) (∈)
2. \( (true \land \neg (max < x)) \vdash \neg (max < x) \) (1, ∧ −)
3. \( (true \land \neg (max < x)) \vdash (max \geq x) \) (def. of ≥)
4. \( \emptyset \vdash (true \land \neg (max < x)) \rightarrow (max \geq x) \)
Example 2 for Conditionals

Prove that the program CODE2 (in blue) satisfies the specifications

\((|true| \ CODE2 \ (|(x > y \land max = x) \lor (x \leq y \land max = y)|)|) \) under partial correctness.

\(|true|

if (x > y) {
    max = x;
} else {
    max = y;
}

(|(x > y \land max = x) \lor (x \leq y \land max = y)|)
Example 2: Annotated Code

\[
\begin{align*}
&\text{(true)} \\
&\text{if (x > y) { } } \\
&\quad \text{(x > y)} \quad \text{if-then-else} \\
&\quad (x > y \land x = x) \lor (x \leq y \land x = y) \quad \text{implied (a)} \\
&\quad \text{max = x ; } \\
&\quad (x > y \land \text{max} = x) \lor (x \leq y \land \text{max} = y) \quad \text{assignment} \\
&\text{}} \text{else { } } \\
&\quad \text{(\neg(x > y))} \quad \text{if-then-else} \\
&\quad (x > y \land y = x) \lor (x \leq y \land y = y) \quad \text{implied (b)} \\
&\quad \text{max = y ; } \\
&\quad (x > y \land \text{max} = x) \lor (x \leq y \land \text{max} = y) \quad \text{assignment} \\
&\text{}} \\
&\quad (x > y \land \text{max} = x) \lor (x \leq y \land \text{max} = y) \quad \text{if-then-else}
\end{align*}
\]
Example 2: Implied Conditions

(a) Prove $\emptyset \vdash x > y \rightarrow (x > y \land x = x) \lor (x \leq y \land x = y)$

1. $x > y \vdash x > y$ ($\in$)
2. $\emptyset \vdash x = x$ ($\approx +$)
3. $x > y \vdash x = x$ (2, $+$)
4. $x > y \vdash x > y \land x = x$ (1, 3, $\land +$)
5. $x > y \vdash (x > y \land x = x) \lor (x \leq y \land x = y)$ (4, $\lor +$)
6. $\emptyset \vdash x > y \rightarrow (x > y \land x = x) \lor (x \leq y \land x = y)$ (4, $\rightarrow +$)
Example 2 for Conditionals

(b) Prove \( x \leq y \rightarrow ((x > y \land x = x) \lor (x \leq y \land y = y)) \).

1. \( x \leq y \vdash x \leq y \ (\in) \)
2. \( \emptyset \vdash y = y \ (\approx +) \)
3. \( x \leq y \vdash y = y \ (2,+ ) \)
4. \( x \leq y \vdash x \leq y \land y = y \ (1,3,\land +) \)
5. \( x \leq y \vdash (x > y \land x = x) \lor (x \leq y \land y = y) \ (4,\lor +) \)
6. \( \emptyset \vdash x \leq y \rightarrow (x > y \land x = x) \lor (x \leq y \land y = y) \ (5,\rightarrow +) \)
While-Loops and Total Correctness
Inference Rule: Partial-while

“Partial while”: do not (yet) require termination.

\[
\frac{(I \land B) \ C \ (I \land B)}{(I \land B) \ \text{while} \ (B) \ C \ (I \land \neg B)} \quad \text{(partial-while)}
\]

In words:

If the code \(C\) satisfies the triple \((I \land B) \ C \ (I \land B)\), and \(I\) is true at the start of the while-loop, then no matter how many times we execute \(C\), condition \(I\) will still be true.

Condition \(I\) is called a loop invariant.

After the while-loop terminates, \(\neg B\) is also true.
Annotations for Partial-while

\[
\begin{align*}
&\langle P \rangle \\
&\langle I \rangle \\
\text{while ( } B \text{ )} \\
&\langle I \land B \rangle \\
&\quad C \\
&\langle I \rangle \\
\end{align*}
\]
\[
\begin{align*}
\text{partial-while} \\
\langle I \land \neg B \rangle \\
\langle Q \rangle \\
\end{align*}
\]

(a) Prove \( P \rightarrow I \) (precondition \( P \) implies the loop invariant)

(b) Prove \(( I \land \neg B ) \rightarrow Q\) (exit condition implies postcondition)

We need to determine \( I \)!!
A *loop invariant* is an assertion (condition) that is true both *before* and *after* each execution of the body of a loop.

- True before the *while*-loop begins.
- True after the *while*-loop ends.
- Expresses a relationship among the variables used within the body of the loop. Some of these variables will have their values changed within the loop.
- An invariant may or may not be useful in proving termination (to discuss later).
Example: Finding a loop invariant

\[ (\forall x \geq 0) \]
\[ y = 1 ; \]
\[ z = 0 ; \]
\[ \rightarrow \text{while (z} \neq \text{x}) \{ \]
\[ \quad z = z + 1 ; \]
\[ \quad y = y \times z ; \]
\[ \} \]
\[ (\forall y = x!) \]

At the while statement:

\[
\begin{array}{cccc}
x & y & z & z \neq x \\
5 & 1 & 0 & \text{true} \\
5 & 1 & 1 & \text{true} \\
5 & 2 & 2 & \text{true} \\
5 & 6 & 3 & \text{true} \\
5 & 24 & 4 & \text{true} \\
5 & 120 & 5 & \text{false} \\
\end{array}
\]

From the trace and the post-condition, a candidate loop invariant is \( y = z! \).

Why are \( y \geq z \) or \( x \geq 0 \) not useful?

These do not involve the loop-termination condition.
Annotations Inside a while-Loop

1. First annotate code using the while-loop inference rule, and any other control rules, such as if-then.
2. Then work bottom-up (“push up”) through program code.
   - Apply inference rule appropriate for the specific line of code, or
   - Note a new assertion (“implied”) to be proven separately.
3. Prove the implied assertions using the inference rules of ordinary logic.
Example: annotations for partial-while

Annotate by partial-while, with chosen invariant \((y = z)\). Annotate assignment statements (bottom-up). Note the required implied conditions.

\[
\begin{align*}
(\lnot x &\geq 0) \\
(\lnot 1 &\geq 0) \\
y & = 1 \quad \text{assignment} \\
(\lnot y &\geq 0) \\
z & = 0 \quad \text{assignment} \\
(\lnot y &\geq z) \\
\text{while } (z &\neq x) \{ \\
(\lnot (y = z) &\land \lnot (z = x)) \\
(\lnot y(z+1) &\geq (z+1)) \\
z & = z + 1 \quad \text{assignment} \\
(\lnot yz &\geq z) \\
y & = y \ast z \quad \text{assignment} \\
(\lnot y &\geq z) \\
\} \\
(\lnot y &\geq z \land z = x) \\
(\lnot y &\geq x) 
\end{align*}
\]

implied (a)  
implied (b)  
implied (c)  

\text{CS245 (Winter 2018) Program Verification  March 28, 2018  72 / 88}
Example: implied conditions (a) and (c)

Proof of implied (a): \((x \geq 0) \vdash (1 = 0!)\).

By definition of factorial.

Proof of implied (c): \(((y = z!) \land (z = x)) \vdash (y = x!)\).

1. \((y = z!) \land (z = x) \vdash (y = z!) \land (z = x)\) (\(\in\))
2. \((y = z!) \land (z = x) \vdash (y = z!)\) (1, \(\land \rightarrow\))
3. \((y = z!) \land (z = x) \vdash (z = x)\) (1, \(\land \rightarrow\))
4. \((y = z!) \land (z = x) \vdash (y = x!)\) (2, 3, \(\approx \rightarrow\))
Example: implied condition (b)

Proof of implied (b):

\[ ((y = z!) \land \neg(z = x)) \vdash (z + 1)y = (z + 1)! \]

1. \( y = z! \land z \neq x \vdash y = z! \land z \neq x \) (\( \in \))
2. \( y = z! \land z \neq x \vdash y = z! \) (1, \( \land \neg \))
3. \( (z + 1)y = (z + 1)z! \) (2, algebra)
4. \( (z + 1)z! = (z + 1)! \) (def. of factorial)
5. \( (z + 1)y = (z + 1)! \) (3, 4, transitivity of equality)
Example 2 (Partial-while)

Prove the following is satisfied under partial correctness.

\[ (| n \geq 0 \land a \geq 0 |) \]

\[
\begin{align*}
  s &= 1 ; \\
  i &= 0 ; \\
  \text{while} (i < n) \{ \\
    &\quad s = s * a ; \\
    &\quad i = i + 1 ; \\
  \} \\
  (| s = a^n |)
\end{align*}
\]

Trace of the loop:

<table>
<thead>
<tr>
<th>a</th>
<th>n</th>
<th>i</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1*2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1<em>2</em>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1<em>2</em>2*2</td>
</tr>
</tbody>
</table>

Candidate for the loop invariant: \( s = a^i \).
Example 2: Testing the invariant

Using \( s = a^i \) as an invariant yields the annotations shown at right.

Next, we want to
- Push up for assignments
- Prove the implications

But: implied (c) is false!

We must use a different invariant.

\[
\begin{align*}
& (\| n \geq 0 \land a \geq 0 \|) \\
& (\| \ldots \|) \\
& s = 1 ; \\
& (\| \ldots \|) \\
& i = 0 ; \\
& (\| s = a^i \|) \\
\text{while } (i < n) \{ \\
& (\| s = a^i \land i < n \|) \quad \text{partial-while} \\
& (\| \ldots \|) \\
& s = s \ast a ; \\
& (\| \ldots \|) \\
& i = i + 1 ; \\
& (\| s = a^i \|) \\
\} \\
(\| s = a^i \land i \geq n \|) \quad \text{partial-while} \\
(\| s = a^n \|) \quad \text{implied (c)}
\end{align*}
\]
Example 2: Adjusted invariant

Try a new invariant:

\[ s = a^i \land i \leq n \, . \]

Now the “implied” conditions are actually true, and the proof can succeed.

\[
\begin{align*}
( n \geq 0 \land a \geq 0 ) & \quad \text{implied (a)} \\
( 1 = a^0 \land 0 \leq n ) & \quad \text{assignment} \\
 s = 1 & \\
( s = a^0 \land 0 \leq n ) & \quad \text{assignment} \\
i = 0 & \\
( s = a^i \land i \leq n ) & \quad \text{assignment} \\
\text{while (} i < n \text{) } \{ & \\
( s = a^i \land i \leq n \land i < n ) & \quad \text{partial-while} \\
( s \cdot a = a^{i+1} \land i + 1 \leq n ) & \quad \text{implied (b)} \\
s = s \ast a ; & \\
( s = a^{i+1} \land i + 1 \leq n ) & \quad \text{assignment} \\
i = i + 1 ; & \\
( s = a^i \land i \leq n ) & \quad \text{assignment} \\
\} & \\
( s = a^i \land i \leq n \land i \geq n ) & \quad \text{partial-while} \\
( s = a^n ) & \quad \text{implied (c)}
\end{align*}
\]
Total Correctness (Termination)

Total Correctness = Partial Correctness + Termination

Only while-loops can be responsible for non-termination in our programming language.

(In general, recursion can also cause it).

Proving termination:
For each while-loop in the program,

Identify an integer expression which is always non-negative and whose value decreases every time through the while-loop.
Example For Total Correctness

The code below has a “loop guard” of $z \neq x$, which is equivalent to $x - z \neq 0$.

What happens to the value of $x - z$ during execution?

```plaintext
y = 1 ;
z = 0 ;

while ( z != x ) {
    z = z + 1 ;
    y = y * z ;
}
```

At start of loop: $x - z = x \geq 0$

$x - z$ decreases by 1

$x - z$ unchanged

Thus the value of $x - z$ will eventually reach 0. The loop then exits and the program terminates.
Proof of Total Correctness

We chose an expression $x - z$ (called the \textit{variant}).

At the start of the loop, $x - z \geq 0$:

- Precondition: $x \geq 0$.
- Assignment $z \leftarrow 0$.

Each time through the loop:

- $x$ doesn’t change: no assignment to it.
- $z$ increases by 1, by assignment.
- Thus $x - z$ decreases by 1.

Thus the value of $x - z$ will eventually reach 0.

When $x - z = 0$, the guard $z \neq x$ ends the loop.
Total Correctness Problem

Total Correctness Problem: Given a program $C$, does it satisfy the specification $(|P|)C(|Q|)$ under total correctness?

Theorem The Total Correctness Problem is undecidable.

Proof:

- Reduce the Blank-Tape Halting Problem to our problem.
- Suppose we have an algorithm $A$ to solve the Total Correctness Problem.
- We can use it to solve the Blank-Tape Halting Problem.
- Given program $C$ as input, we can use our algorithm $A$ to test if $(|true|)C(|true|)$ is totally correct.
- Claim: The program $C$ halts on a blank tape iff this Hoare triple is totally correct.
- Contradiction since the Blank-Tape Halting Problem is undecidable.
Partial Correctness Problem

Partial Correctness Problem: Given a program $C$, does it satisfy the specification ($| P |$ $C$ $| Q |$) under partial correctness?

Theorem The Partial Correctness Problem is undecidable.

Proof:

- Reduce the Blank-Tape Halting Problem to our problem.
- Suppose we have an algorithm $A$ to solve the Partial Correctness Problem. We can use it to solve the Blank-Tape Halting Problem for any program $C$ as follows.
- Given program $C$ as input, make a new program $C'$ by adding a new line at the end of the program $C$ (here $x$ is a new variable):

\[ x = 1; \]

- **Claim**: The Hoare Triple ($| true |$ $C'$ $| x = 0 |$) is partially correct iff $C'$ does not halt.
- **Contradiction** since the Blank-Tape Halting Problem is undecidable.
Where did our method for proving partial/total correctness fail to be an algorithm?

- finding an invariant for while loops
- finding a variant to prove that while loops terminate
- proving the implied conditions - recall that validity in first order (predicate) logic is undecidable.
Logic and Computation:
Summary
Propositional Logic

- Translations from English to propositional logic formulas
- Syntax - well-formed formulas, structural induction
- Semantics (truth tables, value assignments)
- Proving validity of arguments expressed in propositional logic (by truth tables or by contradiction)
- Propositional calculus laws and normal forms (CNF, DNF)
- Adequate sets of connectives
- Applications of propositional logic: Logic gates, circuits, code simplification
- Formal (natural) deduction, 11 rules, its soundness and completeness
- Automated theorem-proving: Resolution, Davis Putnam Procedure
- Solving the Satisfiability problem (SAT) with DNA computing
Predicate logic (first-order logic)

- **Translations** from English to predicate logic formulas, and implications for program specifications
- **Syntax** - well-formed formulas in predicate logic
- **Semantics** - interpretations, domains, satisfiability
- Proving **validity of arguments** expressed in predicate logic
- **Formal deduction** for predicate logic (17 rules)
- **Applications**: Automated proof verifiers, Resolution
Undecidability; Other Logic Applications

- Undecidability, Halting Problem, other undecidable problems
- Applications and implications of predicate logic
  - Peano Arithmetic
  - Godel’s Incompleteness Theorem
  - Program Verification
- Solve logical puzzles and debug invalid arguments

All cats have four legs.
I have four legs.
Therefore I am a cat? ("Dog Thoughts" by King)

What’s wrong with this argument?
Use Logic Wisely!

INTELLIGENCE IS SPOTTING THE FLAW IN THE BOSS’S REASONING... WISDOM IS REFRAINING FROM POINTING IT OUT!

- THE END -