



華東師範大學  
EAST CHINA NORMAL  
UNIVERSITY

# Logic in Computer Science

## Lecture 07

### Hoare Logic and Program Verification

Li Qin

Associate Professor

School of Computer Science and Software Engineering

Verification methods may be classified according to the following main criteria:

- *Proof-based vs. model-based* - if a soundness and completeness theorem holds, then:
  - proof = valid formula = true in **all** models;
  - model-based = check satisfiability in **one** model
- *Degree of automation* - fully automated, partially automated, or manual
- *Full- vs. property-verification* - a single property vs. full behavior
- *Domain of application* - hardware or software; sequential or concurrent; reactive or terminating; etc.
- *Pre- vs. post-development*

Used to verify sequential programs with infinite state and complex data.

- Proof based
- Semi-automatic —some steps cannot be carried out algorithmically by a computer.
- Property-oriented
- Application domain: Sequential, transformational programs
- Pre/post development: the methods can be used during the development process to create small proofs that can be subsequently combined into proofs of larger program fragments.

## Why should we specify and verify code

---

- A formal specification is less ambiguous.
- Experience has shown that verifying programs w.r.t. formal specifications can significantly cut down the duration of software development and maintenance by eliminating most errors in the planning phase.
- Makes debugging easier
- Software built from formal specifications is easier to reuse.
- Verification of safety-critical software *guarantees* safety; testing does not.
- Many examples of software-related catastrophies due to lack of verification.
  - Arienne rocket exploded immediately after launch
  - Lost control of Martian probe
  - Y2K problem

As a software developer, you may get an order from a customer, which provides an informal description of your task.

- Convert the informal description  $D$  of an application domain into an “equivalent” formula  $\Phi_D$  of some symbolic logic.
- Write a program  $P$  which is meant to realize  $\Phi_D$  in the programming environment required by the customer.
- Prove that  $P$  satisfies  $\Phi_D$ .

# A Core Programming Language

We use a language with simple integer and boolean expressions, and simple commands: assignment, if, and while commands.

$$\begin{aligned} E &::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E) \\ B &::= \text{true} \mid \text{false} \mid (!B) \mid (B \& B) \mid (B \mid B) \mid E < E \\ C &::= x = E \mid C; C \mid \text{if } B \{C\} \text{ else } \{C\} \mid \text{while } B \{C\} \end{aligned}$$

Example:

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

We need to be able to express the following statement: “If the execution of a program fragment  $P$  starts in a state satisfying  $\Phi$ , then the execution of  $P$  ends in a state satisfying  $\Psi$ . We denote this by:

$$(\Phi) P (\Psi)$$

and we call this construct a *Hoare triple*.  $\Phi$  is called the *precondition*, and  $\Psi$  is called the *postcondition*.

**Example:** Assume that the specification of a program  $P$  is “*to calculate a number whose square is less than  $x$ .*” Then, the following assertion should hold:

$$(x > 0) P (y \cdot y < x)$$

**It means:** if we start execution in a state where  $x > 0$ , then the execution of  $P$  ends with a state where  $y^2 < x$ .

What happens if the execution starts with  $x \leq 0$ ? We don’t know!

Both these examples realize the specification  $\langle x > 0 \rangle P \langle y \cdot y < x \rangle$ .

$$\begin{array}{l} \langle x > 0 \rangle \\ y = 0 \\ \langle y \cdot y < x \rangle \end{array}$$
$$\begin{array}{l} \langle x > 0 \rangle \\ y = 0 \\ \text{while } (y * y < x) \{ \\ \quad y = y + 1 \\ \} \\ y = y - 1 \\ \langle y \cdot y < x \rangle \end{array}$$



- *Partial correctness*: we **do not require** the program to terminate.
- *Total correctness*: we **do require** the program to terminate.

**Definition (partial correctness):** We say that the triple  $\langle \Phi \rangle P \langle \Psi \rangle$  is satisfied under partial correctness if, for all states which satisfy  $\Phi$ , the state resulting from  $P$ 's execution satisfies the postcondition  $\Psi$ , provided that  $P$  actually terminates. In this case we write

$$\models_{par} \langle \Phi \rangle P \langle \Psi \rangle$$

**Definition (total correctness):** We say that the triple  $\langle \Phi \rangle P \langle \Psi \rangle$  is satisfied total partial correctness if, for all states in which  $P$  is executed and which satisfy the precondition  $\Phi$ ,  $P$  is guaranteed to terminate, and the state resulting from  $P$ 's execution satisfies the postcondition  $\Psi$ . In this case we write

$$\models_{tot} \langle \Phi \rangle P \langle \Psi \rangle$$

The following statement

$$\models_{par} (\Phi) \text{ while true } \{ x = 0; \} (\Psi)$$

holds for all  $\Phi$  and  $\Psi$ . The corresponding total correctness statement does not hold.

Succ:

```
a = x + 1;
if (a - 1 == 0 {
    y = 1;
} else {
    y = a;
}
```

We have:

$$\models_{par} (\top) \text{ Succ } (y = x + 1)$$

and

$$\models_{tot} (\top) \text{ Succ } (y = x + 1)$$

**Remark:**  $\models_{tot} (\Phi) P (\Psi)$  implies  $\models_{par} (\Phi) P (\Psi)$ .

# Program Variables and Logical Variables

Consider the examples:

Fac2:

```
y = 1;
while (x != 0) {
    y = y * x;
    x = x - 1;
}
```

Sum:

```
z = 0;
while (x > 0) {
    z = z + x;
    x = x - 1;
}
```

The values of  $y$  and  $z$  are functions of *the original* values of  $x$ . That value is no longer available as a program variable at the end of the program. We introduce logical variables to handle this situation.

$$\models_{tot} (x = x_0 \wedge x \geq 0) \text{ Fac2 } (y = x_0!)$$

$$\models_{tot} (x = x_0 \wedge x > 0) \text{ sum } \left( \left| z = \frac{x_0(x_0 + 1)}{2} \right| \right)$$

$$\frac{(\phi) C_1 (\eta) \quad (\eta) C_2 (\psi)}{(\phi) C_1; C_2 (\psi)} \text{Composition}$$

$$\frac{}{(\psi[E/x]) x = E (\psi)} \text{Assignment}$$

$$\frac{(\phi \wedge B) C_1 (\psi) \quad (\phi \wedge \neg B) C_2 (\psi)}{(\phi) \text{if } B \{C_1\} \text{ else } \{C_2\} (\psi)} \text{If-statement}$$

$$\frac{(\psi \wedge B) C (\psi)}{(\psi) \text{while } B \{C\} (\psi \wedge \neg B)} \text{Partial-while}$$

$$\frac{\vdash \phi' \rightarrow \phi \quad (\phi) C (\psi) \quad \vdash \psi \rightarrow \psi'}{(\phi') C (\psi')} \text{Implied}$$

## Proof Trees (1)

---

$$\frac{\frac{(1 = 1) \ y = 1 \ (y = 1)}{( \top ) \ y = 1 \ (y = 1)} i}{\frac{( \top ) \ y = 1; \ z = 0 \ (y = 1 \wedge z = 0)}{( \top ) \ y = 1; \ z = 0 \ (y = 1 \wedge z = 0)} c} \frac{\frac{(y = 1 \wedge 0 = 0) \ z = 0 \ (y = 1 \wedge z = 0)}{(y = 1) \ z = 0 \ (y = 1 \wedge z = 0)} i}{( \top ) \ y = 1; \ z = 0 \ (y = 1 \wedge z = 0)} i$$

## Proof Trees (2)

$$\begin{array}{c}
 \frac{(y \cdot (z + 1) = (z + 1)!) \ z = z+1 \ (y \cdot z = z!)}{(y = z! \wedge z \neq x) \ z = z+1 \ (y \cdot z = z!)} \quad i \\
 \frac{\quad (y \cdot z = z!) \ y = y * z \ (y = z!)}{(y = z! \wedge z \neq x) \ z = z+1; \ y = y * z \ (y = z!)} \quad c \\
 \frac{\quad}{(y = z!) \ \text{while} \ (z \neq x) \ \{z = z+1; \ y = y * z\} \ (y = z! \wedge z = x)} \quad w \\
 \frac{\quad}{(y = 1 \wedge z = 0) \ \text{while} \ (z \neq x) \ \{z = z+1; \ y = y * z\} \ (y = x!)} \quad i
 \end{array}$$

Using the rule for composition, we get

$$\frac{(\top) \ y = 1; \ z = 0; \ \text{while } (z \neq x) \ \{z = z+1; \ y = y*z\} \ (y = x!)}{}$$

The rule for sequential composition suggests a more convenient way of presenting proofs in program logic: *proof tableaux*. We can think of any program of our core programming language as a sequence.

Corresponding tableau:

$$\begin{array}{l} C_1; \\ C_2; \\ \vdots \\ C_n \end{array}$$

Each of the transitions

$$\langle \Phi_i \rangle \quad C_{i+1} \quad \langle \Phi_{i+1} \rangle$$

appeals to one of the proof rules

$$\begin{array}{ll} \langle \Phi_0 \rangle & \\ C_1; & \\ \langle \Phi_1 \rangle & \text{justifi cation} \\ C_2; & \\ \langle \Phi_2 \rangle & \text{justifi cation} \\ \vdots & \langle \Phi_{n-1} \rangle \text{ justifi cation} \\ C_n; & \\ \langle \Phi_n \rangle & \text{justifi cation} \end{array}$$



We show  $\vdash_{\text{par}} (y = 5) \ x = y + 1 \ (x = 6)$ :

$(y = 5)$	
$(y + 1 = 6)$	Implied
$x = y + 1$	
$(x = 6)$	Assignment

We prove  $\vdash_{\text{par}} (y < 3) \ y = y + 1 \ (y < 4)$ :

$(y < 3)$	
$(y + 1 < 4)$	Implied
$y = y + 1;$	
$(y < 4)$	Assignment

## Example: If Statement

$\langle \top \rangle$	
$\langle (x + 1 - 1 = 0 \rightarrow 1 = x + 1) \wedge (\neg(x + 1 - 1 = 0) \rightarrow x + 1 = x + 1) \rangle$	Implied
$a = x + 1;$	
$\langle (a - 1 = 0 \rightarrow 1 = x + 1) \wedge (\neg(a - 1 = 0) \rightarrow a = x + 1) \rangle$	Assignment
if (a - 1 == 0) {	
$\langle 1 = x + 1 \rangle$	If-Statement
y = 1;	
$\langle y = x + 1 \rangle$	Assignment
} else {	
$\langle a = x + 1 \rangle$	If-Statement
y = a;	
$\langle y = x + 1 \rangle$	Assignment
}	
$\langle y = x + 1 \rangle$	If-Statement

**Definition:** An *invariant* of the while-statement  $\text{while } B \{C\}$  having guard  $B$  and body  $C$  is a formula  $\eta$  such that  $\models_{par} (\eta \wedge B) \ C \ (\eta)$ ; i.e., if  $\eta$  and  $B$  are true in a state and  $C$  is executed and terminates, then  $\eta$  is again true in the resulting state.

Example:

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

iteration	z	y	B
0	0	1	true
1	1	1	true
2	2	2	true
3	3	6	true
4	4	24	true
5	5	120	true
6	6	720	false

Invariant:  $y = z!$

## Example

$(\top)$	
$(1 = 0!)$	Implied
$y = 1;$	
$(y = 0!)$	Assignment
$z = 0;$	
$(y = z!)$	Assignment
$\text{while } (z \neq x) \{$	
$(y = z! \wedge z \neq x)$	Invariant Hyp. $\wedge$ guard
$(y \cdot (z + 1) = (z + 1)!)$	Implied
$z = z + 1;$	
$(y \cdot z = z!)$	Assignment
$y = y * z;$	
$(y = z!)$	Assignment
$\}$	
$(y = z! \wedge \neg(z \neq x))$	Partial-while
$(y = x!)$	Implied

## While Rule for Total Correctness

---

$$\frac{(\eta \wedge B \wedge 0 \leq E = E_0) \ C \ (\eta \wedge 0 \leq E < E_0)}{(\eta \wedge 0 \leq E) \ \text{while } B \ \{C\} \ (\eta \wedge \neg B)} \quad \text{Total-while}$$

## Example

$(x \geq 0)$	
$(1 = 0! \wedge 0 \leq x - 0)$	Implied
$y = 1;$	
$(y = 0! \wedge 0 \leq x - 0)$	Assignment
$z = 0;$	
$(y = z! \wedge 0 \leq x - z)$	Assignment
<b>while</b> $(x \neq z)$ {	
$(y = z! \wedge x \neq z \wedge 0 \leq x - z = E_0)$	Invariant Hyp. $\wedge$ guard
$(y \cdot (z + 1) = (z + 1)! \wedge 0 \leq x - (z + 1) < E_0)$	Implied
$z = z + 1;$	
$(y \cdot z = z! \wedge 0 \leq x - z < E_0)$	Assignment
$y = y * z;$	
$(y = z! \wedge 0 \leq x - z < E_0)$	Assignment
}	
$(y = z! \wedge x = z)$	Total-while
$(y = x!)$	Implied