CS228 : Logic for Computer Science

Module 4: Hoare Logic and Program Verification

Lecture 19: Introduction to Hoare Logic

Instructor: S. Akshay

IIT Bombay, India

Logistics

Quiz 2

- **Friday, Oct 23rd, 6pm**
- Syllabus:
  - Module 3 (Predicate Logic/FOL): all of Lectures 12–18
  - Module 4 (Hoare logic): from Lecture 19 onwards, covered till Thursday, Oct 15th

Logistics

Quiz 2

- **Friday, Oct 23rd, 6pm**
- Syllabus:
  - Module 3 (Predicate Logic/FOL): all of Lectures 12–18
  - Module 4 (Hoare logic): from Lecture 19 onwards, covered till Thursday, Oct 15th

SMT solver demo

- Postponed to next week.

Topics Covered till date

Module 1: Propositional Logic

1. Syntax
2. Semantics
3. Formal proof systems
4. Normal Forms

Module 2: The Problem of Satisfiability

1. Propositional Resolution
2. DPLL
3. CDCL
4. SAT/SMT tool demo

Topics Covered till date

Module 1: Propositional Logic

1. Syntax
2. Semantics
3. Formal proof systems
4. Normal Forms

Module 2: The Problem of Satisfiability

1. Propositional Resolution
2. DPLL
3. CDCL
4. SAT/SMT tool demo

Module 3: Predicate Logic

1. Syntax of predicate logic
2. Semantics of predicate logic
3. Natural deduction for FOL
4. Structures, expressiveness, and limits of FOL
5. Normal forms for FOL
6. DPLL(T) and SMT solving

Topics Covered till date

Module 1: Propositional Logic

1. Syntax
2. Semantics
3. Formal proof systems
4. Normal Forms

Module 2: The Problem of Satisfiability

1. Propositional Resolution
2. DPLL
3. CDCL
4. SAT/SMT tool demo

Module 3: Predicate Logic

1. Syntax of predicate logic
2. Semantics of predicate logic
3. Natural deduction for FOL
4. Structures, expressiveness, and limits of FOL
5. Normal forms for FOL
6. DPLL(T) and SMT solving

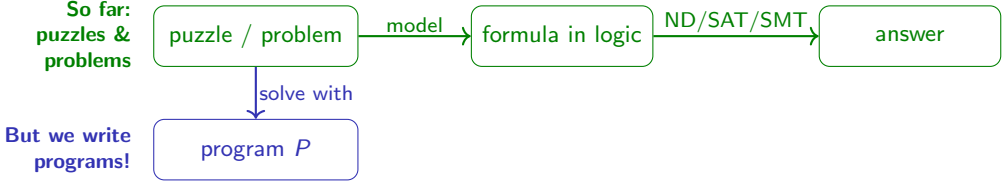
Module 4: Hoare Logic and Program Verification

Today: Introduction to Hoare logic

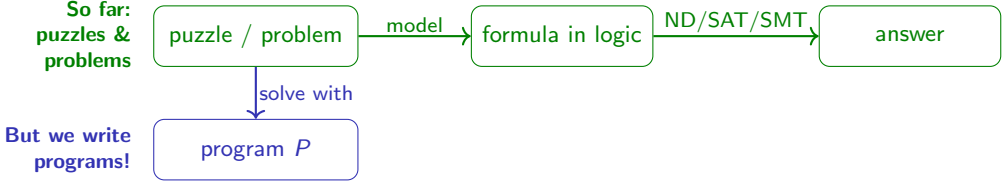
Recap: Hoare Logic and Program Verification



Recap: Hoare Logic and Program Verification

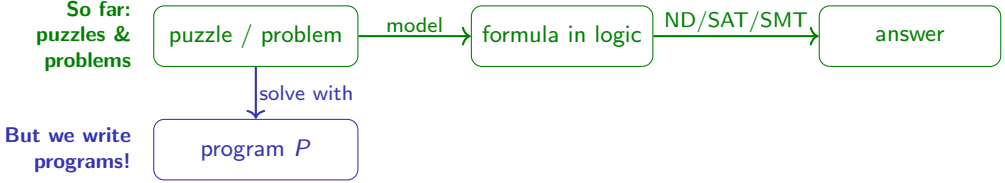


Recap: Hoare Logic and Program Verification



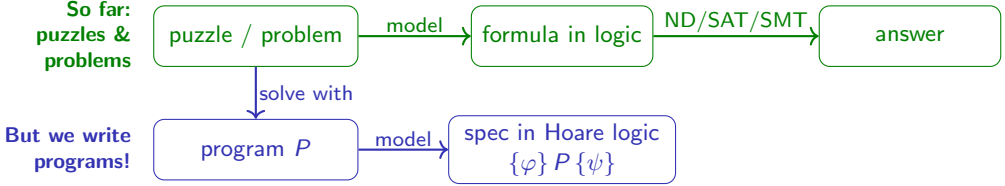
- A logic to reason about programs

Recap: Hoare Logic and Program Verification



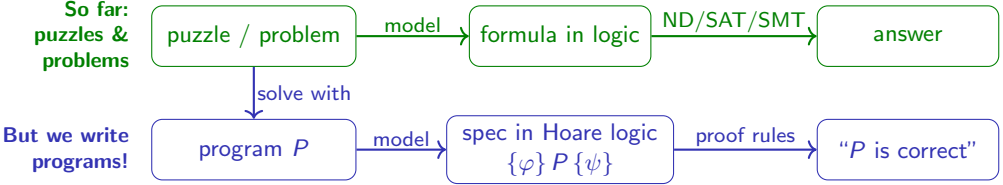
- A logic to reason about programs
- A logic of programs, Hoare logic, built on top of FOL!

Recap: Hoare Logic and Program Verification



- A logic to reason about programs
- A logic of programs, Hoare logic, built on top of FOL!
- In this module: for *simple* imperative programs, we'll see how to *state* properties in logic (pre/postconditions) and *prove* them using proof rules.

Recap: Hoare Logic and Program Verification



- A logic to reason about programs
- A logic of programs, Hoare logic, built on top of FOL!
- In this module: for *simple* imperative programs, we'll see how to *state* properties in logic (pre/postconditions) and *prove* them using proof rules.
- Real program verifiers (e.g., Dafny) largely automate the proof step using SMT solvers!

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
	while $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Let's call this **Fac**$_1$.

- **Exercise 19.1** What is y when the loop stops for input (i) $x = 3$ and (ii) $x = 0$?

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Let's call this **Fac**$_1$.

- **Exercise 19.1** What is y when the loop stops for input (i) $x = 3$ and (ii) $x = 0$? $y = 6 = 3!, y = 1 = 0!$

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Trace for $x = 3$:

after	z	y	$z \neq x$?
start	0	1	yes
iter. 1	1	1	yes
iter. 2	2	2	yes
iter. 3	3	6	no (stop)

Let's call this **Fac**$_1$.

- **Exercise 19.1** What is y when the loop stops for input (i) $x = 3$ and (ii) $x = 0$? $y = 6 = 3!, y = 1 = 0!$

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Trace for $x = 3$:

after	z	y	$z \neq x$?
start	0	1	yes
iter. 1	1	1	yes
iter. 2	2	2	yes
iter. 3	3	6	no (stop)

Let's call this **Fac**$_1$.

- **Exercise 19.1** What is y when the loop stops for input (i) $x = 3$ and (ii) $x = 0$? $y = 6 = 3!, y = 1 = 0!$
- The program is meant to compute the factorial of x and to store that in y.

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Trace for $x = 3$:

after	z	y	$z \neq x$?
start	0	1	yes
iter. 1	1	1	yes
iter. 2	2	2	yes
iter. 3	3	6	no (stop)

Let's call this **Fac**$_1$.

- **Exercise 19.1** What is y when the loop stops for input (i) $x = 3$ and (ii) $x = 0$? $y = 6 = 3!, y = 1 = 0!$
- The program is meant to compute the factorial of x and to store that in y.
- But does it do that? How do we show that it really computes $x!$ in y?

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Trace for $x = 3$:

after	z	y	$z \neq x$?
start	0	1	yes
iter. 1	1	1	yes
iter. 2	2	2	yes
iter. 3	3	6	no (stop)

Let's call this **Fac**$_1$.

- **Exercise 19.1** What is y when the loop stops for input (i) $x = 3$ and (ii) $x = 0$? $y = 6 = 3!, y = 1 = 0!$
- The program is meant to compute the factorial of x and to store that in y.
- But does it do that? How do we show that it really computes $x!$ in y?
- What happens if we start with $x = -2$?

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Trace for $x = 3$:

after	z	y	$z \neq x$?
start	0	1	yes
iter. 1	1	1	yes
iter. 2	2	2	yes
iter. 3	3	6	no (stop)

Let's call this **Fac**$_1$.

- **Exercise 19.1** What is y when the loop stops for input (i) $x = 3$ and (ii) $x = 0$? $y = 6 = 3!, y = 1 = 0!$
- The program is meant to compute the factorial of x and to store that in y.
- But does it do that? How do we show that it really computes $x!$ in y?
- What happens if we start with $x = -2$? It never stops: $z = 1, 2, 3, \ldots$ never equals -2!

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Trace for $x = 3$:

after	z	y	$z \neq x$?
start	0	1	yes
iter. 1	1	1	yes
iter. 2	2	2	yes
iter. 3	3	6	no (stop)

Let's call this **Fac**$_1$.

- **Exercise 19.1** What is y when the loop stops for input (i) $x = 3$ and (ii) $x = 0$? $y = 6 = 3!, y = 1 = 0!$
- The program is meant to compute the factorial of x and to store that in y.
- But does it do that? How do we show that it really computes $x!$ in y?
- What happens if we start with $x = -2$? It never stops: $z = 1, 2, 3, \ldots$ never equals -2!
- In fact, if $x \geq 0$, then **Fac**$_1$ computes $x!$ and stores it in y.

Consider a simple program

What does this program do? Take x as input and y as output.

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

Trace for $x = 3$:

after	z	y	$z \neq x$?
start	0	1	yes
iter. 1	1	1	yes
iter. 2	2	2	yes
iter. 3	3	6	no (stop)

Let's call this **Fac**$_1$.

- **Exercise 19.1** What is y when the loop stops for input (i) $x = 3$ and (ii) $x = 0$? $y = 6 = 3!, y = 1 = 0!$
- The program is meant to compute the factorial of x and to store that in y.
- But does it do that? How do we show that it really computes $x!$ in y?
- What happens if we start with $x = -2$? It never stops: $z = 1, 2, 3, \ldots$ never equals -2!
- In fact, if $x \geq 0$, then **Fac**$_1$ computes $x!$ and stores it in y. Our goal is to prove this!

Proving programs correct

1. If $\mathbf{Fac}_1$ is run with $x \geq 0$, it terminates with $y = x!$.

Proving programs correct

1. If $\mathbf{Fac}_1$ is run with $x \geq 0$, it terminates with $y = x!$.
2. Whenever $\mathbf{Fac}_1$ terminates, we have $y = x!$.

Proving programs correct

1. If $\mathbf{Fac}_1$ is run with $x \geq 0$, it terminates with $y = x!$.
2. Whenever $\mathbf{Fac}_1$ terminates, we have $y = x!$.

Our goal is to prove BOTH these statements!

Proving programs correct

1. If $\mathbf{Fac}_1$ is run with $x \geq 0$, it terminates with $y = x!$.
2. Whenever $\mathbf{Fac}_1$ terminates, we have $y = x!$.

Our goal is to prove BOTH these statements!

- Proof goals mix a program with facts about its input/output values,

Proving programs correct

1. If $\mathbf{Fac}_1$ is run with $x \geq 0$, it terminates with $y = x!$.
2. Whenever $\mathbf{Fac}_1$ terminates, we have $y = x!$.

Our goal is to prove BOTH these statements!

- Proof goals mix a program with facts about its input/output values,
- so we treat both as mathematical/logical objects.

Proving programs correct

1. If $\mathbf{Fac}_1$ is run with $x \geq 0$, it terminates with $y = x!$.
2. Whenever $\mathbf{Fac}_1$ terminates, we have $y = x!$.

Our goal is to prove BOTH these statements!

- Proof goals mix a program with facts about its input/output values,
- so we treat both as mathematical/logical objects.

We explore a paradigm that can do this: Hoare Logic

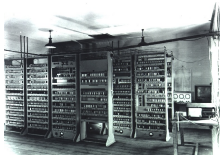
Proving programs correct

1. If $\mathbf{Fac}_1$ is run with $x \geq 0$, it terminates with $y = x!$.
2. Whenever $\mathbf{Fac}_1$ terminates, we have $y = x!$.

Our goal is to prove BOTH these statements!

- Proof goals mix a program with facts about its input/output values,
- so we treat both as mathematical/logical objects.

We explore a paradigm that can do this: Hoare Logic



EDSAC (1948)

In 1949, at a Cambridge conference, Turing asked *"How can one check a routine in the sense of making sure that it is right?"* His example was a factorial program, with assertions for correctness and a quantity that decreases and vanishes for termination – exactly our two statements!



Alan Turing

To reason about programs, we must first write them formally

- To prove things about programs, programs must be mathematical objects, just as formulas were.

To reason about programs, we must first write them formally

- To prove things about programs, programs must be mathematical objects, just as formulas were.
- So, as for logic, we need
  - Syntax: precisely which strings are programs? (a grammar)

To reason about programs, we must first write them formally

- To prove things about programs, programs must be mathematical objects, just as formulas were.
- So, as for logic, we need
  - Syntax: precisely which strings are programs? (a grammar)
  - Semantics: precisely what does a program do? (how it changes the state)

To reason about programs, we must first write them formally

- To prove things about programs, programs must be mathematical objects, just as formulas were.
- So, as for logic, we need
  - Syntax: precisely which strings are programs? (a grammar)
  - Semantics: precisely what does a program do? (how it changes the state)
- Where have we done this before?

To reason about programs, we must first write them formally

- To prove things about programs, programs must be mathematical objects, just as formulas were.
- So, as for logic, we need
  - Syntax: precisely which strings are programs? (a grammar)
  - Semantics: precisely what does a program do? (how it changes the state)
- Where have we done this before? Propositional logic (Lectures 1–2) and predicate logic (Lectures 12–13)!

Topic 19.1

A Simple Programming Language

A simple (imperative) programming language

Integer expressions

Variables $x, y, z, \ldots$, Numerals $0, 1, 2, \ldots, -1, -2, \ldots$, basic operations: addition ($+$), subtraction ($-$), multiplication (*), negation.

A simple (imperative) programming language

Integer expressions

Variables $x, y, z, \ldots$, Numerals $0, 1, 2, \ldots, -1, -2, \ldots$, basic operations: addition ($+$), subtraction ($-$), multiplication (*), negation.

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

n is an integer, x a variable. We drop parentheses using the usual binding conventions: negation, then multiplication, then subtraction and addition.

A simple (imperative) programming language

Integer expressions

Variables $x, y, z, \ldots$, Numerals $0, 1, 2, \ldots, -1, -2, \ldots$, basic operations: addition ($+$), subtraction ($-$), multiplication (*), negation.

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

n is an integer, x a variable. We drop parentheses using the usual binding conventions: negation, then multiplication, then subtraction and addition.

- **Exercise 19.2** Which of these does the grammar generate?
  1. $(4 + (x - 3))$
  2. $(x + ((x * x) * (y - (5 + z))))$
  3. $x + y * 2$

A simple (imperative) programming language

Integer expressions

Variables $x, y, z, \ldots$, Numerals $0, 1, 2, \ldots, -1, -2, \ldots$, basic operations: addition $(+)$, subtraction $(-)$, multiplication $(*)$, negation.

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

n is an integer, x a variable. We drop parentheses using the usual binding conventions: negation, then multiplication, then subtraction and addition.

- **Exercise 19.2** Which of these does the grammar generate?
  1. $(4 + (x - 3))$
  2. $(x + ((x * x) * (y - (5 + z))))$
  3. $x + y * 2$ Yes, because of binding conventions: it abbreviates $(x + (y * 2))$.

A simple (imperative) programming language

Integer expressions

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

Boolean expressions

Why do we need them?

A simple (imperative) programming language

Integer expressions

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

Boolean expressions

Why do we need them? For **if-then-else** and **while** statements!

A simple (imperative) programming language

Integer expressions

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

Boolean expressions

Why do we need them? For **if-then-else** and **while** statements!

$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$

A simple (imperative) programming language

Integer expressions

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

Boolean expressions

$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$

- Note that this builds on integer expressions
- As usual we can define other operators using these.
- **Exercise 19.3** Define $E_1 \neq E_2, E_1 < E_2$ using the above.

A simple (imperative) programming language

Integer expressions

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

Boolean expressions

$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$

Commands, i.e., programs

$$C ::= (x := E) \mid C; C \mid \textbf{if } B \ \{C\} \textbf{ else } \{C\} \mid \textbf{while } B \ \{C\}$$

where { and } mark the start and end of blocks of code

Syntax and semantics of a simple program

Fix variables $x, y, z, \ldots$, numerals/integers $n \in \{0, 1, 2, \ldots, -1, -2, \ldots\}$

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$
$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$
$$C ::= (x := E) \mid C; C \mid \textbf{if } B \ \{C\} \textbf{ else } \{C\} \mid \textbf{while } B \ \{C\}$$

Syntax and semantics of a simple program

Fix variables $x, y, z, \ldots$, numerals/integers $n \in \{0, 1, 2, \ldots, -1, -2, \ldots\}$

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$
$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$
$$C ::= (x := E) \mid C; C \mid \textbf{if } B \ \{C\} \textbf{ else } \{C\} \mid \textbf{while } B \ \{C\}$$

The state of the program is determined by the current value of all its variables. As the program executes, its state changes.

Syntax and semantics of a simple program

Fix variables $x, y, z, \ldots$, numerals/integers $n \in \{0, 1, 2, \ldots, -1, -2, \ldots\}$

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$
$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$
$$C ::= (x := E) \mid C; C \mid \textbf{if } B \ \{C\} \textbf{ else } \{C\} \mid \textbf{while } B \ \{C\}$$

The state of the program is determined by the current value of all its variables. As the program executes, its state changes.

- $(x := E)$ is the usual assignment statement.

Syntax and semantics of a simple program

Fix variables $x, y, z, \ldots$, numerals/integers $n \in \{0, 1, 2, \ldots, -1, -2, \ldots\}$

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$
$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$
$$C ::= (x := E) \mid C; C \mid \textbf{if } B \ \{C\} \textbf{ else } \{C\} \mid \textbf{while } B \ \{C\}$$

The state of the program is determined by the current value of all its variables. As the program executes, its state changes.

- $(x := E)$ is the usual assignment statement.
  - Intuitively, program evaluates integer expression E in current state and overwrites current value stored in x with the value computed by the evaluation.

Syntax and semantics of a simple program

Fix variables $x, y, z, \ldots$, numerals/integers $n \in \{0, 1, 2, \ldots, -1, -2, \ldots\}$

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$
$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$
$$C ::= (x := E) \mid C; C \mid \textbf{if } B \ \{C\} \textbf{ else } \{C\} \mid \textbf{while } B \ \{C\}$$

The state of the program is determined by the current value of all its variables. As the program executes, its state changes.

- $(x := E)$ is the usual assignment statement.
- $C; C$ is sequential composition.

Syntax and semantics of a simple program

Fix variables $x, y, z, \ldots$, numerals/integers $n \in \{0, 1, 2, \ldots, -1, -2, \ldots\}$

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$
$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$
$$C ::= (x := E) \mid C; C \mid \textbf{if } B \ \{C\} \textbf{ else } \{C\} \mid \textbf{while } B \ \{C\}$$

The state of the program is determined by the current value of all its variables. As the program executes, its state changes.

- $(x := E)$ is the usual assignment statement.
- $C; C$ is sequential composition.
  - $C_1; C_2$ means execute C_1 in current state and if it terminates, then execute C_2 from the state at which it terminated.

Syntax and semantics of a simple program

Fix variables $x, y, z, \ldots$, numerals/integers $n \in \{0, 1, 2, \ldots, -1, -2, \ldots\}$

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$
$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$
$$C ::= (x := E) \mid C; C \mid \textbf{if } B \ \{C\} \textbf{ else } \{C\} \mid \textbf{while } B \ \{C\}$$

The state of the program is determined by the current value of all its variables. As the program executes, its state changes.

- $(x := E)$ is the usual assignment statement.
- $C; C$ is sequential composition.
- **if** $B \ \{C_1\}$ **else** $\{C_2\}$ evaluates boolean expression B in current state and if that result is true, then C_1 is executed, else C_2 is executed.

Syntax and semantics of a simple program

Fix variables $x, y, z, \ldots$, numerals/integers $n \in \{0, 1, 2, \ldots, -1, -2, \ldots\}$

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$
$$B ::= \top \mid \bot \mid (\neg B) \mid (B \wedge B) \mid (E \leq E)$$
$$C ::= (x := E) \mid C; C \mid \textbf{if } B \ \{C\} \textbf{ else } \{C\} \mid \textbf{while } B \ \{C\}$$

The state of the program is determined by the current value of all its variables. As the program executes, its state changes.

- $(x := E)$ is the usual assignment statement.
- $C; C$ is sequential composition.
- **if** $B \ \{C_1\}$ **else** $\{C_2\}$ evaluates boolean expression B in current state and if that result is true, then C_1 is executed, else C_2 is executed.
- **while** $B \ \{C\}$ evaluates B at current state. If B evaluates false the command terminates. Else, C is executed. When C terminates we re-evaluate B with updated state and continue. Thus, we repeatedly evaluate C as long as B is true.

The simple programming language

Exercise 19.4

1. Write a program in the above syntax to
   1.1 compute 2^n from n.
   1.2 find greatest y whose square is less than x
2. What features of a common imperative (i.e., state based) programming language like C, java, are not captured? Give 4 examples. Is it easy to modify the language to have these features?

Topic 19.2

Specifying Programs: Hoare Triples

Hoare Logic, sometimes called Floyd–Hoare Logic

Main idea: Use logic to capture what must be true before and after a command!

In general, a **Hoare triple** $\{\varphi\}\, P\, \{\psi\}$ is a formula of Hoare logic:

- φ and ψ are formulas in a base logic, e.g., predicate logic, Presburger arithmetic, etc. In our case, we will use first order logic over integers.

Hoare Logic, sometimes called Floyd–Hoare Logic

Main idea: Use logic to capture what must be true before and after a command!

In general, a **Hoare triple** $\{\varphi\}\, P\, \{\psi\}$ is a formula of Hoare logic:

- φ and ψ are formulas in a base logic, e.g., predicate logic, Presburger arithmetic, etc. In our case, we will use first order logic over integers.
  E.g., for $\mathbf{Fac}_1$: $\{x \geq 0\}\ \mathbf{Fac}_1\ \{y = x!\}$

Hoare Logic, sometimes called Floyd–Hoare Logic

Main idea: Use logic to capture what must be true before and after a command!

In general, a **Hoare triple** $\{\varphi\}\, P\, \{\psi\}$ is a formula of Hoare logic:

- φ and ψ are formulas in a base logic, e.g., predicate logic, Presburger arithmetic, etc. In our case, we will use first order logic over integers.
  E.g., for $\mathbf{Fac}_1$: $\{x \geq 0\}\ \mathbf{Fac}_1\ \{y = x!\}$
- P is an imperative program in our language

Hoare Logic, sometimes called Floyd–Hoare Logic

Main idea: Use logic to capture what must be true before and after a command!

In general, a **Hoare triple** $\{\varphi\}\, P\, \{\psi\}$ is a formula of Hoare logic:

- φ and ψ are formulas in a base logic, e.g., predicate logic, Presburger arithmetic, etc. In our case, we will use first order logic over integers.
  E.g., for $\mathbf{Fac}_1$: $\{x \geq 0\}\ \mathbf{Fac}_1\ \{y = x!\}$
- P is an imperative program in our language
- φ is called the precondition of P and ψ the postcondition.

Hoare Logic, sometimes called Floyd–Hoare Logic

Main idea: Use logic to capture what must be true before and after a command!

In general, a **Hoare triple** $\{\varphi\}\, P\, \{\psi\}$ is a formula of Hoare logic:

- φ and ψ are formulas in a base logic, e.g., predicate logic, Presburger arithmetic, etc. In our case, we will use first order logic over integers.
  E.g., for $\mathbf{Fac}_1$: $\{x \geq 0\}\ \mathbf{Fac}_1\ \{y = x!\}$
- P is an imperative program in our language
- φ is called the precondition of P and ψ the postcondition.



Robert Floyd



Tony Hoare

Floyd 1967, Hoare 1969

Turing Awards: 1978, 1980

States and formulas

States and Formulas

- A state of the program is an environment, i.e., a function α that assigns to each variable x an integer $\alpha(x)$ (as in Lecture 13).
- Assertions φ, ψ are formulas of first order logic over the vocabulary $\mathcal{V}$ with the numerals as constants, function symbols $-$ (unary) and $+, *, -$ (binary), and predicate symbols $\leq$ and $=$.
- Let $M_{\mathbb{Z}} = (\mathbb{Z}, (\ldots, -1, 0, 1, \ldots, -, +, *, \leq))$ be the $\mathcal{V}$-structure in which every symbol has its usual meaning. We write $\alpha \models \varphi$ for $M_{\mathbb{Z}}, \alpha \models \varphi$.

States and formulas

States and Formulas

- A state of the program is an environment, i.e., a function α that assigns to each variable x an integer $\alpha(x)$ (as in Lecture 13).
- Assertions φ, ψ are formulas of first order logic over the vocabulary $\mathcal{V}$ with the numerals as constants, function symbols $-$ (unary) and $+, *, -$ (binary), and predicate symbols $\leq$ and $=$.
- Let $M_{\mathbb{Z}} = (\mathbb{Z}, (\ldots, -1, 0, 1, \ldots, -, +, *, \leq))$ be the $\mathcal{V}$-structure in which every symbol has its usual meaning. We write $\alpha \models \varphi$ for $M_{\mathbb{Z}}, \alpha \models \varphi$.

Exercise 19.5

Let α be a state where $\alpha(x) = -2, \alpha(y) = 5, \alpha(z) = -1$. Which of the following hold?

(a) $\alpha \models \neg(x + y < z)$

(b) $\alpha \models \forall u\,(y < u \rightarrow y * z < u * z)$

States and formulas

States and Formulas

- A state of the program is an environment, i.e., a function α that assigns to each variable x an integer $\alpha(x)$ (as in Lecture 13).
- Assertions φ, ψ are formulas of first order logic over the vocabulary $\mathcal{V}$ with the numerals as constants, function symbols $-$ (unary) and $+, *, -$ (binary), and predicate symbols $\leq$ and $=$.
- Let $M_{\mathbb{Z}} = (\mathbb{Z}, (\ldots, -1, 0, 1, \ldots, -, +, *, \leq))$ be the $\mathcal{V}$-structure in which every symbol has its usual meaning. We write $\alpha \models \varphi$ for $M_{\mathbb{Z}}, \alpha \models \varphi$.

Exercise 19.5

Let α be a state where $\alpha(x) = -2, \alpha(y) = 5, \alpha(z) = -1$. Which of the following hold?

(a) $\alpha \models \neg(x + y < z)$

(b) $\alpha \models \forall u\,(y < u \rightarrow y * z < u * z)$

(a) holds: $x + y = 3$ is not less than $z = -1$. (b) fails: for $u = 6$, $y * z = -5 < -6 = u * z$ is false.

Examples of Hoare Triples

Some examples of correct Hoare triples

- $\{(n \geq 0) \wedge (n * n > 20)\}\ \mathrm{n} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{\neg(m = 50)\}$

Examples of Hoare Triples

Some examples of correct Hoare triples

- $\{(n \geq 0) \wedge (n * n > 20)\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{\neg(m = 50)\}$ quantifier-free first order logic over integers.

Examples of Hoare Triples

Some examples of correct Hoare triples

- $\{(n \geq 0) \wedge (n * n > 20)\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{\neg(m = 50)\}$ quantifier-free first order logic over integers.
- $\{\exists z\,(z + z = x)\}\ \mathrm{y} := \mathrm{x} + 2\ \{\exists z\,(z + z = y)\}$

Examples of Hoare Triples

Some examples of correct Hoare triples

- $\{(n \geq 0) \wedge (n * n > 20)\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{\neg(m = 50)\}$ quantifier-free first order logic over integers.
- $\{\exists z\,(z + z = x)\}\ \mathrm{y} := \mathrm{x} + 2\ \{\exists z\,(z + z = y)\}$ general first order logic over integers: "if x is even, so is y".

Examples of Hoare Triples

Some examples of correct Hoare triples

- $\{(n \geq 0) \wedge (n * n > 20)\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{\neg(m = 50)\}$ quantifier-free first order logic over integers.
- $\{\exists z\,(z + z = x)\}\ \mathrm{y} := \mathrm{x} + 2\ \{\exists z\,(z + z = y)\}$ general first order logic over integers: "if x is even, so is y".

An incorrect Hoare triple

$\{n \geq 0\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{m \geq 4\}$

Examples of Hoare Triples

Some examples of correct Hoare triples

- $\{(n \geq 0) \wedge (n * n > 20)\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{\neg(m = 50)\}$ quantifier-free first order logic over integers.
- $\{\exists z\,(z + z = x)\}\ \mathrm{y} := \mathrm{x} + 2\ \{\exists z\,(z + z = y)\}$ general first order logic over integers: "if x is even, so is y".

An incorrect Hoare triple

$\{n \geq 0\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{m \geq 4\}$ – fails for $n = 0$: then $m = 1$.

Examples of Hoare Triples

Some examples of correct Hoare triples

- $\{(n \geq 0) \wedge (n * n > 20)\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{\neg(m = 50)\}$ quantifier-free first order logic over integers.
- $\{\exists z\,(z + z = x)\}\ \mathrm{y} := \mathrm{x} + 2\ \{\exists z\,(z + z = y)\}$ general first order logic over integers: "if x is even, so is y".

An incorrect Hoare triple

$\{n \geq 0\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{m \geq 4\}$ – fails for $n = 0$: then $m = 1$.

Exercise 19.6

Give 3 "correct" and 3 "incorrect" Hoare triples.

Examples of Hoare Triples

Some examples of correct Hoare triples

- $\{(n \geq 0) \wedge (n * n > 20)\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{\neg(m = 50)\}$ quantifier-free first order logic over integers.
- $\{\exists z\,(z + z = x)\}\ \mathrm{y} := \mathrm{x} + 2\ \{\exists z\,(z + z = y)\}$ general first order logic over integers: "if x is even, so is y".

An incorrect Hoare triple

$\{n \geq 0\}\ \mathrm{m} := \mathrm{n} + 1; \mathrm{m} := \mathrm{m} * \mathrm{m}\ \{m \geq 4\}$ – fails for $n = 0$: then $m = 1$.

Exercise 19.6

Give 3 "correct" and 3 "incorrect" Hoare triples.

How do we formalize the notion of correctness?

Correctness and semantics of Hoare Triples

Partial correctness

A triple $\{\varphi\}P\{\psi\}$ is satisfied under partial correctness, denoted $\models_{par} \{\varphi\}P\{\psi\}$ if, for all states that satisfy φ, the state resulting from P's execution satisfies the postcondition ψ, provided P actually terminates.

Correctness and semantics of Hoare Triples

Partial correctness

A triple $\{\varphi\}P\{\psi\}$ is satisfied under partial correctness, denoted $\models_{par} \{\varphi\}P\{\psi\}$ if, for all states that satisfy φ, the state resulting from P's execution satisfies the postcondition ψ, provided P actually terminates.

Total correctness

A triple $\{\varphi\}P\{\psi\}$ is satisfied under total correctness, denoted $\models_{tot} \{\varphi\}P\{\psi\}$ if, for all states that satisfy φ, P when executed is guaranteed to terminate and the resulting state satisfies the postcondition ψ.

Correctness and semantics of Hoare Triples

Partial correctness

A triple $\{\varphi\}P\{\psi\}$ is satisfied under partial correctness, denoted $\models_{par} \{\varphi\}P\{\psi\}$ if, for all states that satisfy φ, the state resulting from P's execution satisfies the postcondition ψ, provided P actually terminates.

Total correctness

A triple $\{\varphi\}P\{\psi\}$ is satisfied under total correctness, denoted $\models_{tot} \{\varphi\}P\{\psi\}$ if, for all states that satisfy φ, P when executed is guaranteed to terminate and the resulting state satisfies the postcondition ψ.

Examples: Recall the program $\mathbf{Fac}_1$ from earlier,

- $\{\top\}\ \mathbf{Fac}_1\ \{y = x!\}$ satisfies partial correctness, but not total correctness.
- $\{x \geq 0\}\ \mathbf{Fac}_1\ \{y = x!\}$ satisfies partial AND total correctness.

Examples

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Examples

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- $\{\top\}$ **Fac**$_1$ $\{y = x!\}$ satisfies

Examples

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- $\{\top\}$ **Fac**$_1$ $\{y = x!\}$ satisfies partial correctness, but NOT total correctness.

Examples

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- $\{\top\}$ **Fac**$_1$ $\{y = x!\}$ satisfies partial correctness, but NOT total correctness.
- $\{x \geq 0\}$ **Fac**$_1$ $\{y = x!\}$ satisfies

Examples

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- $\{\top\}$ **Fac**$_1$ $\{y = x!\}$ satisfies partial correctness, but NOT total correctness.
- $\{x \geq 0\}$ **Fac**$_1$ $\{y = x!\}$ satisfies partial AND total correctness.

Examples

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Unk$_1$ *inp* x *out* y	$a := x + 1;$ **if** $(a - 1 = 0)\{$ $\quad y := 1;$ $\}$ **else** $\{$ $\quad y := a;$ $\}$

- $\{\top\}$ **Fac$_1$** $\{y = x!\}$ satisfies partial correctness, but NOT total correctness.
- $\{x \geq 0\}$ **Fac$_1$** $\{y = x!\}$ satisfies partial AND total correctness.

Exercise 19.7

1. What does Unk_1 do? Take x as input and y as output.
2. Write a Hoare triple that captures the spec. Does it hold under partial/total correctness?

Program variables and logical variables

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Program variables and logical variables

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Fac$_2$ *inp* x *out* y	$y := 1;$ **while** $(x \neq 0)\{$ $\quad y := y * x;$ $\quad x := x - 1;$ $\}$

Program variables and logical variables

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Fac$_2$ *inp* x *out* y	$y := 1;$ **while** $(x \neq 0)\{$ $\quad y := y * x;$ $\quad x := x - 1;$ $\}$

- **Fac**$_2$ is another program that computes the factorial.
- $\{x \geq 0\}$ **Fac**$_1$ $\{y = x!\}$ satisfies partial and total correctness.
- Qn: What about $\{x \geq 0\}$ **Fac**$_2$ $\{y = x!\}$?

Program variables and logical variables

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Fac$_2$ *inp* x *out* y	$y := 1;$ **while** $(x \neq 0)\{$ $\quad y := y * x;$ $\quad x := x - 1;$ $\}$

- **Fac**$_2$ is another program that computes the factorial.
- $\{x \geq 0\}$ **Fac**$_1$ $\{y = x!\}$ satisfies partial and total correctness.
- Qn: What about $\{x \geq 0\}$ **Fac**$_2$ $\{y = x!\}$?
- How do we correct this?

Program variables and logical variables

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Fac$_2$ *inp* x *out* y	$y := 1;$ **while** $(x \neq 0)\{$ $\quad y := y * x;$ $\quad x := x - 1;$ $\}$

- **Fac**$_2$ is another program that computes the factorial.
- $\{x \geq 0\}$ **Fac**$_1$ $\{y = x!\}$ satisfies partial and total correctness.
- Qn: What about $\{x \geq 0\}$ **Fac**$_2$ $\{y = x!\}$?
- How do we correct this? What is the correct specification? Need to remember initial value!

Program variables and logical variables

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Fac$_2$ *inp* x *out* y	$y := 1;$ **while** $(x \neq 0)\{$ $\quad y := y * x;$ $\quad x := x - 1;$ $\}$

- **Fac**$_2$ is another program that computes the factorial.
- $\{x \geq 0\}$ **Fac**$_1$ $\{y = x!\}$ satisfies partial and total correctness.
- Qn: What about $\{x \geq 0\}$ **Fac**$_2$ $\{y = x!\}$?
- How do we correct this? What is the correct specification? Need to remember initial value!
- we obtain $\{x = x_0 \wedge x \geq 0\}$ **Fac**$_2$ $\{y = x_0!\}$

Program variables and logical variables

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

Fac$_2$ *inp* x *out* y	$y := 1;$ **while** $(x \neq 0)\{$ $\quad y := y * x;$ $\quad x := x - 1;$ $\}$

- **Fac**$_2$ is another program that computes the factorial.
- $\{x \geq 0\}$ **Fac**$_1$ $\{y = x!\}$ satisfies partial and total correctness.
- Qn: What about $\{x \geq 0\}$ **Fac**$_2$ $\{y = x!\}$?
- How do we correct this? What is the correct specification? Need to remember initial value!
- we obtain $\{x = x_0 \wedge x \geq 0\}$ **Fac**$_2$ $\{y = x_0!\}$ x_0 is called a logical variable.