A Short Introduction to Hoare Logic

Supratik Chakraborty

I.I.T. Bombay

June 23, 2008

Motivation

- Assertion checking in (sequential) programs
- Interesting stuff happens when heap memory allocated, freed and mutated
  - Absolutely thrilling stuff happens if you throw in concurrency!
- Hoare Logic: A logic for reasoning about programs and assertions
  - Program as a mathematical object
  - Inference system: Properties of program from properties of sub-programs
- This lecture primarily about sequential programs that don't change heap.
  - Highlight problems that arise when dealing with heap
  - Hongseok Yang will show how **separation logic** allows Hoare-style reasoning on heap-manipulating programs
  - Can also be used to reason about concurrent programs sharing resources

Example programs

```
int foo(int n) {
  local int k, int j;
  k := 0;
  j := 1;
  while (k != n) {
    k := k + 1;
    j := 2*j;
  }
  return(j)
}
```

```
int bar(int n) {
  local int k, int j;
  k := 0;
  j := 1;
  while (k != n)  {
    k := k + 1;
    j := 2 + j;
  }
  return(j);
}
```

Wish to prove:

1. If `foo` is called with parameter `n` greater than 0, it returns 2^n
2. If `bar` is called with parameter `n` greater than 0, it returns $1 + 2n$

Note: No heap manipulation by above programs.

Some observations

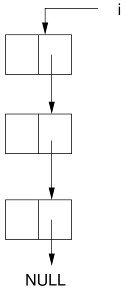
Proof goals from previous slide:

1. If `foo` is called with parameter `n` greater than 0, it returns 2^n
2. If `bar` is called with parameter `n` greater than 0, it returns $1 + 2n$

- What we want to prove involves **both** the program and properties of input and output values
- Our proof goal (and subgoals) and proof technique must therefore refer to both program and input/output values "at par" (equally important)
- Program must therefore be treated as much a mathematical object as formulas like $(n > 0)$
- This is what Hoare logic does very elegantly

We will be able to prove properties of both programs by end of today!

A list reversal program




```
ptr list Reverse(ptr list i) {
 local ptr list j, ptr list k;
 j := NULL;
 while (i != NULL) {
    k := i->next;
    i->next := j;
    j := i;
    i := k;
 }
 return(i);
}
```

Wish to prove: If `i` points to an acyclic list before `Reverse` executes, it also points to an acyclic list after `Reverse` returns.

- Requires specifying properties of/reasoning about heap memory!
- We should be able to prove this by end of this week!
  - Not by end of today

A simple storage model

Assume integer and pointer types.

$$
\begin{aligned}
\text{Vars} = \{x, y, z, \ldots\} & \quad \ldots \quad \text{User-defined variables} \\
\text{Locs} = \{1, 2, 3, \ldots\} & \quad \ldots \quad \text{Memory addresses in heap} \\
\text{Vals} \supseteq \text{Locs} & \quad \ldots \quad \text{Values in variables/heap locations} \\
\text{Vals} = \mathbf{Z} & \quad \quad \text{(for our purposes)}
\end{aligned}
$$

$$
\begin{aligned}
\text{Heaps} &= \text{Locs} \rightarrow_{fin} \text{Vals} \\
\text{Stacks} &= \text{Vars} \rightarrow \text{Vals} \\
\text{States} &= \text{Stacks} \times \text{Heaps}
\end{aligned}
$$

Example:

- $\text{Vars} = \{x, y\}$, $\text{Locs} = \{97, 200, 1371\}$
- Stack : $x \rightarrow 1, y \rightarrow 29$
- Heap : $97 \rightarrow 29, 200 \rightarrow 235, 1371 \rightarrow 46$

A simple imperative language

E	$::=$	$x \mid n \mid E + E \mid -E \mid \ldots$	Heap-indepedent expr
B	$::=$	$E = E \mid E \geq E \mid B \wedge B \mid \neg B$	Boolean condn
P	$::=$	$x := E \mid P; P \mid \texttt{if}\ B\ \texttt{then}\ P\ \texttt{else}\ P \mid$	Standard constructs
		$\texttt{while}\ B\ P \mid$	Looping construct
		$x := \texttt{new}(E) \mid$	Allocation on heap
		$x := *E \mid$	Lookup of heap
		$*E = E \mid$	Mutation of heap
		$\texttt{free}(E)$	Deallocation of heap

- This lecture primarily discusses how to reason about programs without heap-related constructs
- Hongseok Yang will show how this reasoning can be extended to programs with heap-related constructs

A simple assertion language

- Assertion: A logic formula describing a set of states with some "interesting" property
- Recall States = Stacks × Heaps
- Assertions can refer to both stack and heap

E	$::=$	$x \mid n \mid E + E \mid -E \mid \ldots$	Heap-indepedent expr
B	$::=$	$E = E \mid E \geq E$	Boolean conditions
A	$::=$	B	Atomic predicates on stack
		$\mathrm{emp} \mid E \mapsto E$	Atomic predicates on heap
		$A \star A \mid A \mathrel{-\!\!\star} A$	Wait for a day !!!
		$\mathrm{true} \mid A \wedge A \mid \neg A \mid \forall v.\, A$	Logical connectives

Assertion semantics

- As program executes, its state changes.
- At some point during execution, let state be (s, h)
- Program satisfies assertion A at this point iff $(s, h) \models A$

$$
\begin{array}{rcl}
(s,h) \models B & \text{iff} & [\![B]\!]_s = \text{true} \\
(s,h) \models \neg A & \text{iff} & (s,h) \not\models A \\
(s,h) \models A_1 \wedge A_2 & \text{iff} & (s,h) \models A_1 \text{ and } (s,h) \models A_2 \\
(s,h) \models \forall v.\, A & \text{iff} & \forall x \in \mathbf{Z}.\, (s[v \leftarrow x], h) \models A \\
(s,h) \models \text{emp} & \text{iff} & dom(h) = \emptyset \\
(s,h) \models E_1 \mapsto E_2 & \text{iff} & [\![E_1]\!]_s \in dom(h) \text{ and } h([\![E_1]\!]_s) = [\![E_2]\!]_s \\
(s,h) \models A_1 \star A_2 & \text{iff} & \exists h_0, h_1.\, (dom(h_0) \cap dom(h_1) = \emptyset \wedge h = h_0 * h_1 \\
 & & \quad \wedge (s, h_0) \models A_1 \wedge (s, h_1) \models A_2) \\
(s,h) \models A_1 \mathrel{-\!\!\star} A_2 & \text{iff} & \forall h'.\, (dom(h') \cap dom(h) = \emptyset \wedge (s,h') \models A_1) \\
 & & \quad \text{implies } (s, h * h') \models A_2
\end{array}
$$

Examples of assertions in programs

- Consider program with two variables `x` and `y`, both initialized to 0.
- Assertions $A_1 : x \mapsto y$, $A_2 : y^2 \geq 28$

pc	**Program**	**Stack**	**Heap**	**Sat** $A1$	**Sat** A_2
1	`x = new(1);`	$x : 237, y : 0$	$237 : 123456$	No	No
2	`y = 37;`	$x : 237, y : 37$	$237 : 123456$	No	Yes
3	`*x = 37;`	$x : 237, y : 37$	$237 : 37$	Yes	Yes
4	`x = new(1);`	$x : 10, y : 37$	$237 : 37, 10 : 54$	No	Yes

It therefore makes sense to talk of assertions at specific pc values and how program statements and control flow affect validity of assertions

Hoare logic

In honour of Prof. Tony Hoare who formalized the logic in the way we know it today

A Hoare triple $\{\varphi_1\}P\{\varphi_2\}$ is a formula

- φ_1, φ_2 are formulae in a base logic (e.g., full predicate logic, Presburger logic, separation logic, quantifier-free fragment of predicate logic, etc.)
- P is a program in our imperative language
- Note how programs and formulae in base logic are intertwined
- Terminology: Precondition φ_1, Postcondition φ_2

Examples of syntactically correct Hoare triples:

- $\{(n \geq 0) \wedge (n^2 > 28)\}$ `m := n + 1; m := m*m` $\{\neg(m = 36)\}$
  - Quantifier-free fragment of predicate logic
  - Interpretted predicates and functions over integers
- $\{\exists x, y. (y > 0) \wedge (n = x^y)\}$ `n := n*(n+1)` $\{\exists x, y. (n = x^y)\}$
  - Above fragment augmented with quantifiers

Semantics of Hoare triples

The partial correctness specification $\{\varphi_1\}P\{\varphi_2\}$ is valid iff starting from a state (s_1, h_1) satisfying φ_1,

- No execution of P accesses an unallocated heap cell (no memory error)
- Whenever an execution of P terminates in state (s_2, h_2), then $(s_2, h_2) \models \varphi_2$

The total correctness specification $[\varphi_1]P[\varphi_2]$ is valid iff starting from a state (s_1, h_1) satisfying φ_1,

- No execution of P accesses an unallocated heap cell
- Every execution of P terminates
- When an execution of P terminates in state (s_2, h_2), then $(s_2, h_2) \models \varphi_2$

- For programs without loops, both semantics coincide
- Memory error checking unnecessary in well-specified programs

Hoare logic for a subset of our language

- We will use partial correctness semantics
- Base logic: Predicate logic (with quantifiers) with usual interpretted functions and predicates over integers
- Programs without any heap-manipulating instructions
  - Reasoning about heap: Hongseok's lasting lectures over next 5 days

Restricted program constructs:

E	$::=$	$x \mid n \mid E + E \mid -E \mid \ldots$	Heap-indepedent expr
B	$::=$	$E = E \mid E \geq E \mid B \wedge B \mid \neg B$	Boolean condn
P	$::=$	$x := E \mid P; P \mid \texttt{if}\ B\ \texttt{then}\ P\ \texttt{else}\ P \mid$	Standard constructs
		$\texttt{while}\ B\ P$	Looping construct

Hoare logic: Assignment rule

Program construct:

E	$::=$	$x \mid n \mid E + E \mid -E \mid \ldots$	Heap-indepedent expr
P	$::=$	$x := E$	Assignment statement

Hoare inference rule: If x is free in φ

$$\frac{}{\{\varphi([x \leftarrow E])\} \quad x := E \quad \{\varphi(x)\}}$$

Examples:

- $\{(x + z.y)^2 > 28\}$ x := x + z*y $\{x^2 > 28\}$
  - Identical to weakest precondition computation
- $\{(z.y > 5) \wedge (\exists x.\ y = x^x)\}$ x := z*y $\{(x > 5) \wedge (\exists x.\ y = x^x)\}$
  - x must be free in φ

Hoare logic: Sequential composition rule

Program construct:

$P \quad ::= \quad P; P$ Sequencing of commands

Hoare inference rule:

$$\frac{\{\varphi\}\ P_1\ \{\eta\} \qquad \{\eta\}\ P_2\ \{\psi\}}{\{\varphi\}\ P_1;\ P_2\ \{\psi\}}$$

Example:

$$\frac{\{y+z>4\}\ \mathtt{y := y + z - 1}\ \{y>3\} \qquad \{y>3\}\ \mathtt{x := y + 2}\ \{x>5\}}{\{y+z>4\}\ \mathtt{y := y + z - 1; x := y + 2}\ \{x>5\}}$$

Hoare logic: Strengthening precedent, weakening consequent

Hoare inference rule:

$$\frac{\varphi \Rightarrow \varphi_1 \qquad \{\varphi_1\}\ P\ \{\varphi_2\} \qquad \varphi_2 \Rightarrow \psi}{\{\varphi\}\ P\ \{\psi\}}$$

- $\varphi \Rightarrow \varphi_1$ and $\varphi_2 \Rightarrow \psi$ are implications in base (predicate) logic
- Applicable to arbitrary program P

Example:

$$\frac{((y > 4) \wedge (z > 1)) \Rightarrow (y + z > 5) \quad \{y + z > 5\}\ y := y + z\ \{y > 5\} \quad (y > 5) \Rightarrow (y > 3)}{\{(y > 4) \wedge (z > 1)\}\ y := y + z\ \{y > 3\}}$$

Hoare logic: Conditional branch

Program construct:

E	$::=$	$x \mid n \mid E + E \mid -E \mid \ldots$	Heap-indepedent expr
B	$::=$	$E = E \mid E \geq E \mid B \wedge B \mid \neg B$	Boolean condn
P	$::=$	$\texttt{if}\ B\ \texttt{then}\ P\ \texttt{else}\ P$	Conditional branch

Hoare inference rule:

$$\frac{\{\varphi \wedge B\}\ \ P_1\ \ \{\psi\} \qquad\qquad \{\varphi \wedge \neg B\}\ \ P_2\ \ \{\psi\}}{\{\varphi\}\ \ \texttt{if}\ B\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \ \{\psi\}}$$

Example:

$$\frac{\{(y > 4) \wedge (z > 1)\}\ \ \texttt{y} := \texttt{y} + \texttt{z}\ \ \{y > 3\} \qquad \{(y > 4) \wedge \neg(z > 1)\}\ \ \texttt{y} := \texttt{y} - \texttt{1}\ \ \{y > 3\}}{\{y > 4\}\ \texttt{if (z > 1) then y := y+z else y := y-1}\ \{y > 3\}}$$

Hoare logic: Partial correctness of loops

Program construct:

E	$::=$	$x \mid n \mid E + E \mid -E \mid \ldots$	Heap-indepedent expr
B	$::=$	$E = E \mid E \geq E \mid B \wedge B \mid \neg B$	Boolean condn
P	$::=$	$\texttt{while}\, B\ P$	Looping construct

Hoare inference rule:

$$\frac{\{\varphi \wedge B\}\ \ P\ \ \{\varphi\}}{\{\varphi\}\ \ \texttt{while}\, B\ P\ \ \{\varphi \wedge \neg B\}}$$

- φ is a **loop invariant**
- Partial correctness semantics
  - If loop does not terminate, Hoare triple is vacuously satisfied
  - If it terminates, $(\varphi \wedge \neg B)$ must be satisfied after termination

Hoare logic: Partial correctness of loops

Hoare inference rule:

$$\frac{\{\varphi \wedge B\}\ \ P\ \ \{\varphi\}}{\{\varphi\}\ \ \texttt{while}\, B\ P\ \ \{\varphi \wedge \neg B\}}$$

Example:

$$\frac{\{(y = x + z) \wedge (z \neq 0)\}\ \ \mathtt{x} := \mathtt{x} + 1; \mathtt{z} := \mathtt{z} - 1\ \ \{y = x + z\}}{\{y = x + z\}\ \ \texttt{while (z != 0)}\{\text{x := x+1; z := z-1}\}\ \ \{(y = x + z) \wedge (z = 0)\}}$$

$$\frac{\{(y = x + z) \wedge \text{true}\}\ \ \mathtt{x} := \mathtt{x} + 1; \mathtt{z} := \mathtt{z} - 1\ \ \{y = x + z\}}{\{y = x + z\}\ \ \texttt{while (true)}\{\text{x := x+1; z := z-1}\}\ \ \{(y = x + z) \wedge \text{false}\}}$$

$\{\varphi\}$ `while (true) P` $\{\psi\}$ holds vacuously for all φ, `P` and ψ

Summary of Hoare rules for our (sub-)language

$$\frac{}{\{\varphi([x \leftarrow E])\} \quad x := E \quad \{\varphi(x)\}}$$

Assignment

$$\frac{\{\varphi\}\ P_1\ \{\eta\} \qquad \{\eta\}\ P_2\ \{\psi\}}{\{\varphi\}\ P_1;P_2\ \{\psi\}}$$

Seq. Composition

$$\frac{\{\varphi \wedge B\}\ P_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ P_2\ \{\psi\}}{\{\varphi\}\ \texttt{if}\ B\ \texttt{then}\ P_1\ \texttt{else}\ P_2\ \{\psi\}}$$

Conditional branch

$$\frac{\{\varphi \wedge B\}\ P_1\ \{\varphi\}}{\{\varphi\}\ \texttt{while}\,(B)\ P_1\ \{\varphi \wedge \neg B\}}$$

While loop

$$\frac{\varphi_1 \rightarrow \varphi \qquad \{\varphi\}\ P\ \{\psi\} \qquad \psi \rightarrow \psi_1}{\{\varphi_1\}\ P\ \{\psi_1\}}$$

Precedent-strengthen

Antecedent-weaken

Proof system **sound** and **relatively complete**

Proving properties of simple programs

```
int foo(int n) {
  local int k, int j;
  k := 0;
  j := 1;
  while (k != n) {
    k := k + 1;
    j := 2*j;
  }
  return(j)
}
```

```
int bar(int n) {
  local int k, int j;
  k := 0;
  j := 1;
  while (k != n)  {
    k := k + 1;
    j := 2 + j;
  }
  return(j);
}
```

Can we apply our rules to prove that if `bar` is called with `n` greater than 0, it returns $1 + 2n$?

- Function `bar` has a while loop
- Partial correctness: If `bar` is called with `n` greater than 0, and if `bar` terminates, it returns $1 + 2n$.

Proving properties of simple programs

Let P: Sequence of executable statements in `bar`

```
P ::   k := 0;
       j := 1;
       while (k != n)  {
         k := k + 1;
         j := 2 + j;
       }
```

Our goal is to prove the validity of $\{n > 0\}\ \ P\ \ \{j = 1 + 2.n\}$

A Hoare logic proof

Sequential composition rule will give us a proof if we can fill in the template:

$\{n > 0\}$	Precondition
`k := 0`	
$\{\varphi_1\}$	Midcondition
`j := 1`	
$\{\varphi_2\}$	Midcondition
`while (k != n)` $\{$ `k := k+1; j := 2+j` $\}$	
$\{j = 1 + 2.n\}$	Postcondition

- How do we prove $\{\varphi_2\}$ `while (k != n)` $\{$ `k := k+1; j := 2+j` $\}$ $\{j = 1 + 2.n\}$?
- Recall rule for loops requires a loop invariant
- "Guess" a loop invariant $(j = 1 + 2.k)$

A Hoare logic proof

To prove:

$\{\varphi_2\}$ `while (k != n) k := k+1; j := 2+j` $\{j = 1 + 2.n\}$

using loop invariant $(j = 1 + 2.k)$

If we can show:

- $\varphi_2 \Rightarrow (j = 1 + 2.k)$
- $\{(j = 1 + 2.k) \wedge (k \neq n)\}$ `k := k+1; j:= 2+j` $\{j = 1 + 2.k\}$
- $((j = 1 + 2.k) \wedge \neg(k \neq n)) \Rightarrow (j = 1 + 2.n)$

then

By inference rule for loops

$$\frac{\{(j = 1 + 2.k) \wedge (k \neq n)\}\ \texttt{k := k+1; j:= 2+j}\ \{j = 1 + 2.k\}}{\{j = 1 + 2.k\}\ \texttt{while (k != n) k := k+1; j:= 2+j}\ \{(j = 1 + 2.k) \wedge \neg(k \neq n)\}}$$

By inference rule for strengthening precedents and weakening consequents

$$\frac{\begin{array}{c}\varphi_2 \Rightarrow (j = 1 + 2.k) \\ \{j = 1 + 2.k\}\ \texttt{while (k != n) k := k+1; j:= 2+j}\ \{(j = 1 + 2.k) \wedge \neg(k \neq n)\} \\ ((j = 1 + 2.k) \wedge \neg(k \neq n)) \Rightarrow (j = 1 + 2.n)\end{array}}{\{\varphi_2\}\ \texttt{while (k != n) k := k+1; j:= 2+j}\ \{(j = 1 + 2.n)\}}$$

A Hoare logic proof

How do we show:

- $\varphi_2 \Rightarrow (j = 1 + 2.k)$
- $\{(j = 1 + 2.k) \wedge (k \neq n)\}$ `k := k+1; j:= 2+j` $\{j = 1 + 2.k\}$
- $((j = 1 + 2.k) \wedge \neg(k \neq n)) \Rightarrow (j = 1 + 2.n)$

Note:

- $\varphi_2 \Rightarrow (j = 1 + 2.k)$ holds trivially if φ_2 is $(j = 1 + 2.k)$
- $((j = 1 + 2.k) \wedge \neg(k \neq n)) \Rightarrow (j = 1 + 2.n)$ holds trivially in integer arithmetic

Only remaining proof subgoal:
$\{(j = 1 + 2.k) \wedge (k \neq n)\}$ `k := k+1; j:= 2+j` $\{j = 1 + 2.k\}$

A Hoare logic proof

To show:
$\{(j = 1 + 2.k) \wedge (k \neq n)\}$ `k := k+1; j:= 2+j` $\{j = 1 + 2.k\}$

Applying assignment rule twice

$$\{2 + j = 1 + 2.k\}\ \texttt{j := 2+j}\ \{j = 1 + 2k\}$$
$$\{2 + j = 1 + 2.(k + 1)\}\ \texttt{k := k+1}\ \{2 + j = 1 + 2.k\}$$

Simplifying and applying sequential composition rule

$$\frac{\begin{array}{c}\{1 + j = 2.k\}\ \texttt{j := 2+j}\ \{j = 1 + 2k\} \\ \{j = 1 + 2.k)\}\ \texttt{k := k+1}\ \{1 + j = 2.k\}\end{array}}{\{j = 1 + 2.k\}\ \texttt{k := k+1; j := 2+j}\ \{j = 1 + 2.k\}}$$

Applying rule for strengthening precedent

$$\frac{\begin{array}{c}(j = 1 + 2.k) \wedge (k \neq n)\} \Rightarrow (j = 1 + 2.k) \\ \{j = 1 + 2.k\}\ \texttt{k := k+1; j := 2+j}\ \{j = 1 + 2.k\}\end{array}}{\{(j = 1 + 2.k) \wedge (k \neq n)\}\ \texttt{k := k+1; j := 2+j}\ \{j = 1 + 2.k\}}$$

A Hoare logic proof

We have thus shown that with φ_2 as $(j = 1 + 2.k)$
$\{\varphi_2\}$ `while (k != n) k := k+1; j := 2+j` $\{j = 1 + 2.n\}$ is valid

Recall our template:

$\{n > 0\}$	Precondition
`k := 0`	
$\{\varphi_1\}$	Midcondition
`j := 1`	
$\{\varphi_2 : j = 1 + 2.k\}$	Midcondition
`while (k != n) k := k+1; j := 2+j`	
$\{j = 1 + 2.n\}$	Postcondition

The only missing link now is to show
$\{n > 0\}$ `k := 0` $\{\varphi_1\}$ and
$\{\varphi_1\}$ `j := 1` $\{j = 1 + 2.k\}$

A Hoare logic proof

To show

$\{n > 0\}$ `k := 0` $\{\varphi_1\}$ and

$\{\varphi_1\}$ `j := 1` $\{j = 1 + 2.k\}$

Applying assignment rule twice and simplifying:

$\{0 = k\}$ `j := 1` $\{j = 1 + 2.k\}$

$\{\text{true}\}$ `k := 0` $\{0 = k\}$

Choose φ_1 as $(k = 0)$, so $\{\varphi_1\}$ `j := 1` $\{j = 1 + 2.k\}$ holds.

Applying rule for strengthening precedent:

$$\frac{(n > 0) \Rightarrow \text{true} \qquad \{\text{true}\}\ \texttt{k := 0}\ \{\varphi_1 : k = 0\}}{\{n > 0\}\ \texttt{k := 0}\ \{\varphi_1 : k = 0\}}$$

We have proved partial correctness of function `bar` in Hoare Logic !!!

Exercise

Try proving the other program using the following template:

$\{n > 0\}$	Precondition
`k := 0`	
$\{\varphi_1\}$	Midcondition
`j := 1`	
$\{\varphi_2\}$	Midcondition
`while (k != n) k := k+1; j := 2*j`	
$\{j = 2^n\}$	Postcondition

Hint: Use the loop invariant $(j = 2^k)$

A few sticky things

- We "guessed" the right loop invariant
  - A weaker invariant than $(j = 1 + 2.k)$ would not have allowed us to complete the proof.
  - Finding the strongest invariant of a loop: Undecidable in general!
- Annotations can help
  - Programmer annotates her intended loop invariant
  - This is not the same as giving a proof of correctness
  - But can significantly simplify constructing a proof
  - Checking whether a formula is a loop invariant much simpler than finding a loop invariant
- Tools to infer midconditions from annotations exist
- Some tools claim to infer midconditions directly from code
  - Cannot infer strong enough invariants in all cases
  - Otherwise we could check if a Turing Machine halts !!!
- A powerful technique for proving program correctness, but requires some help from user (by way of providing annotations)

Some structural rules in Hoare logic

Structural rules do not depend on program statements

$$\frac{\{\varphi_1\}\ P\ \{\psi_1\} \quad \{\varphi_2\}\ P\ \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\}\ P\ \{\psi_1 \wedge \psi_2\}}$$ Conjunction

$$\frac{\{\varphi_1\}\ P\ \{\psi_1\} \quad \{\varphi_2\}\ P\ \{\psi_2\}}{\{\varphi_1 \vee \varphi_2\}\ P\ \{\psi_1 \vee \psi_2\}}$$ Disjunction

$$\frac{\{\varphi\}\ P\ \{\psi\}}{\{\exists v.\, \varphi\}\ P\ \{\exists v.\, \psi\}}$$ Exist-quantification (v not free in P)

$$\frac{\{\varphi\}\ P\ \{\psi\}}{\{\forall v.\, \varphi\}\ P\ \{\forall v.\, \psi\}}$$ Univ-quantification (v not free in P)

- We have not given an exhaustive listing of rules
- Just sufficient to get a hang of Hoare-style proofs
- Other rules exist for procedure calls and even concurrency!

What breaks with heap accesses?

Consider a code fragment

```
L0:     local ptr int x, ptr int y;
L1:     x := y;
L2:     *x := 5;
L3:     *y := 7;
L4:     *x := 10;
```

- When control flow reaches L4, the assertion $(x \mapsto 7) \wedge (y \mapsto 7)$ holds.
- Only `*y` is assigned to in statement at L3.
- However, the following Hoare triple (in the spirit of the assignment rule) is not valid:

$$\{(x \mapsto 7) \wedge (7 = 7)\} \quad \texttt{*y := 7} \quad \{(x \mapsto 7) \wedge (y \mapsto 7)\}$$

  - Although `*x` is not explicitly assigned to by statement at L3, the truth of predicate $(x \mapsto 7)$ changes

What breaks with heap accesses?

Without heap (shared resource) accesses, the following **Rule of Constancy** holds in Hoare Logic:

$$\frac{\{\varphi\} \;\; P \;\; \{\psi\}}{\{\varphi \wedge \xi\} \;\; P \;\; \{\psi \wedge \xi\}}$$

where no free variable of ξ is modified by P.

This rule fails with heap (shared resource) accesses due to aliasing

$$\frac{\{\exists t.\, x \mapsto t\} \;\; \texttt{*x := 5} \;\; \{x \mapsto 5\}}{\{(\exists t.\, x \mapsto t) \wedge (y \mapsto 5)\} \;\; \texttt{*x := 5} \;\; \{(x \mapsto 5) \wedge (y \mapsto 5)\}}$$

is not a sound inference rule if $x = y$.

- This motivates the need for special rules for heap accesses
- We'll learn about **separation logic** in the next few days.

Conclusion

- We saw a brief glimpse of Hoare logic and Hoare-style proofs
- Hoare-style proofs have been extensively used over the past few decades to prove subtle properties of complicated programs
- This approach works best with programmer-provided annotations
- The use of automated theorem provers and programmer annotations has allowed application of Hoare-style reasoning to medium sized programs quite successfully.
- Key-Hoare (from Chalmers University): A tool suite for teaching/learning about Hoare logic
- Scalability of Hoare-style reasoning is sometimes an issue
- Yet, this is one of the most elegant techniques available for proving properties of programs