Hoare Logic

Last Time

• Predicate calculus review

Today

• Verification frameworks

• Proving programs correct using Hoare logic
Verification Frameworks

This course is about verifying systems mathematically.

In order to verify systems, need a verification framework, which consists of following components.

**Class Sys of system descriptions.** Sys must be given mathematically, with (at least) semantics and (usually) syntax.

**Class Spec of system specifications/requirements.** Spec must also be given mathematically.

**Relation** sat $\subseteq$ Sys $\times$ Spec.

Once a verification framework has been defined, verifying a specific systems $S \in$ Sys against a specific requirement $R \in$ Spec means proving that $S$ sat $R$. 
Given a verification framework, how does one establish whether or not $S$ sat $R$? Two main approaches.

**Proof-based:** Develop proof rules for proving $S$ sat $R$ and use them to prove correctness.

**Algorithmic:** Give decision procedures for computing if $S$ sat $R$ holds.

In this class we will study several different verification frameworks, including ones that are proof-based and others that are algorithmic.
Hoare Logic and Program Verification

The first verification framework we will study: *Hoare Logic*.

- **Sys** consists of programs written in a simple “guarded commands” programming notation.

- **Spec** consists of pairs “predicates” given in first-order logic (= *predicate calculus*); predicates typically refer to program variables.

- $S$ sat $R$ holds if, whenever program is started in state satisfying first predicate and program terminates, the final state satisfies the second predicate.

- Verification conducted using proof rules.

Logic is sometimes called *Floyd-Hoare* logic and was a big topic of study in 70s and 80s for sequential and parallel programs.
Systems in Hoare Logic are given as programs in a small programming language. We assume existence of following sets.

**Var:** Program variables (assume integer-valued).

**AE:** Arithmetic expressions built using constants, variables, operators, etc. (e.g. $x + 1$)

**BE:** Boolean expressions (e.g. $x = 0$, $(y = 0) \land (x = 2)$).
Let $v_1, \ldots, v_n \subseteq \text{Var}$, $e_1, \ldots, e_n \subseteq \text{AE}$, and $G_1, \ldots, G_n \subseteq \text{BE}$.

- skip: No-op
- halt: Abort
- $v_1, \ldots, v_n := e_1, \ldots, e_n$: Assignment
- $S_1; S_2$: Sequential composition
- if $G_1 \rightarrow S_1[] \ldots [] G_n \rightarrow S_n$ fi: Alternative composition
- do $G_1 \rightarrow S_1[] \ldots [] G_n \rightarrow S_n$ od: Iteration
The semantics (or meaning) of program statements interprets them as mappings of states to states. More formally,

- \( \mathbb{Z} = \{ \ldots, -1, 0, 1, \ldots \} \) is the set of integers.
- \( \mathbb{B} = \{ 0, 1 \} \) is the set of booleans.
- \( \Sigma \) is the set of states i.e. mappings in \( \text{Var} \rightarrow \mathbb{Z} \).
- State \( S \) will be interpreted as a function \( \left[ S \right] \in \Sigma \rightarrow 2^\Sigma \). \( \left[ S \right](\sigma) \) records the states \( S \), when begun in state \( \sigma \), can terminate in.
We can define $[S]$ *inductively* on the structure of $S$. Assumptions:

- A function $[-] - : AE \times \Sigma \rightarrow \mathbb{Z}$; $[e](\sigma)$ returns the value of $e$ in state $\sigma$.
- A function $[-] - : BE \times \Sigma \rightarrow \mathbb{B}$; $[G](\sigma)$ returns (boolean) value of $G$ in $\sigma$.
- If $\sigma \in \Sigma$ and $v_1, \ldots, v_n, k_1, \ldots, k_n$ are sequences of variables/values, then $\sigma[v_1 \mapsto k_1, \ldots, v_n \mapsto k_n]$ is the state given as follows.

\[
(\sigma[v_1 \mapsto k_1, \ldots, v_n \mapsto k_n])(x) = \begin{cases} 
  k_i & \text{if } x = v_i \\
  \sigma(x) & \text{otherwise}
\end{cases}
\]
Defining $[S]$ (cont.)

$[\text{skip}](\sigma) = \{\sigma\}$

$[\text{halt}](\sigma) = \emptyset$

$[v_1, \ldots, v_n := e_1, \ldots, e_n](\sigma) = \{\sigma[v_1 \mapsto [e_1](\sigma), \ldots, v_n \mapsto [e_n](\sigma)]\}$

$[S_1; S_2](\sigma) = \bigcup_{\sigma' \in [S_1](\sigma)} [S_2](\sigma')$

$[\text{if } G_1 \rightarrow S_1[\ldots] G_n \rightarrow S_n \text{ fi}](\sigma) = \bigcup_{\{i \mid [G_i](\sigma) = 1\}} [S_i](\sigma)$
What about do ... od?

\[\left[ \text{do } G_1 \rightarrow S_1 \right] \cdots \left[ G_n \rightarrow S_n \right] \] is defined iteratively. Consider the sequence \( f_i : \Sigma \rightarrow 2^\Sigma \):

\[
\begin{align*}
  f_0(\sigma) &= \emptyset \\
  f_{i+1}(\sigma) &= \{ \sigma \mid [G_1 \lor \cdots \lor G_n](\sigma) = 0 \} \cup \\
  &\quad \bigcup \{ f_i(\sigma') \mid \sigma' \in \bigcup \{ i \mid [G_i](\sigma) = 1 \} \bigcup [S_i](\sigma) \}
\end{align*}
\]

Intuitively, \( f_i \) “unrolls” the do ... od loop \( i - 1 \) times.

Then we can define the semantics of do ... od as follows.

\[
\left[\text{do } G_1 \rightarrow S_1 \right] \cdots \left[ G_n \rightarrow S_n \text{ od} \right](\sigma) = \bigcup_{i=0}^{\infty} f_i(\sigma)
\]
In the Hoare Logic setting we have now defined precisely what \( Sys \) is. What about \( Spec \)?

- \( Spec \) consists of pairs of “state predicates”: a precondition and a postcondition.
- What are “state predicates”? For us: first-order logical formulas over arithmetic, with elements of \( Var \) allowed to appear free.

### Examples

1. \( \text{true} \)

2. \( \forall j \in \mathbb{N}. 0 \leq j < i \rightarrow (\min \leq A[j]) \)

Let \( \mathcal{P} \) be the set of state predicates.

Semantically, state predicates are interpreted with respect to states via a relation \( \models \subseteq \Sigma \times \mathcal{P} \); \( \sigma \models P \) means “\( P \) is true in state \( \sigma \).”
We can now define the set of specifications in Hoare Logic. 

\[ Spec = \mathcal{P} \times \mathcal{P} \]

In a specification \( \langle P, Q \rangle \):

- \( P \) is called the *precondition*. 
- \( Q \) is called the *postcondition*. 

©2007 Rance Cleaveland. All rights reserved.
The Satisfaction Relation in Hoare Logic

The third component in the Hoare Logic verification framework is “sat”: when does a program satisfy a specification?

**Definition** Let $S$ be a program and $\langle P, Q \rangle$ be a precondition/postcondition pair. Then $S$ satisfies $\langle P, Q \rangle$ if whenever $\sigma$ is such that $\sigma \models P$, then every $\sigma' \in [S](\sigma)$ satisfies: $\sigma' \models Q$.

**Notation** Instead of formulating correctness criteria as “$S$ satisfies $\langle P, Q \rangle$,” researchers more normally write $\{P\} S \{Q\}$; this is called a Hoare triple.

A Hoare triple is valid, notation $\models \{P\} S \{Q\}$, if it is the case that $S$ satisfies $\langle P, Q \rangle$.

**Note** The notation of satisfaction here is called *partial correctness*, because programs are not required to terminate. *Total correctness* imposes an additional termination requirement.
We have now defined the Hoare Logic verification framework. How do we now verify programs?

Traditional approach relies on proofs using a collection of *inference rules*.

- Rules have form $\frac{H_1, \ldots, H_n}{C}$ where the $H_i$ are *hypotheses* and $C$ is the *conclusion*.
- If a rule has no hypotheses, it is sometimes called an *axiom*.
- A rule encodes a single step of reasoning; it can be applied once its hypotheses have been proved.

**Notation**

- If $P \in \mathcal{P}$ and $v_1, \ldots, v_n; e_1, \ldots, e_n$ are sequences of variables, expressions, then $P_{e_1, \ldots, e_n}^{v_1, \ldots, v_n}$ is $P$ with the $v_i$ textually replaced by the $e_i$.
- If $P, Q \in \mathcal{P}$ then we write $P \Rightarrow Q$ if for every $\sigma$ such that $\sigma \models P$, it is also the case that $\sigma \models Q$. 

Axioms of Hoare Logic

\[ \begin{align*}
\text{Skip} & \quad \vdash \quad \{ P \} \text{skip} \{ P \} \\
\text{Halt} & \quad \vdash \quad \{ P \} \text{halt} \{ Q \} \\
\text{Asg} & \quad \vdash \quad \{ P^{v_1, \ldots, v_n} \} \quad v_1, \ldots, v_n := e_1, \ldots, e_n \{ P \}
\end{align*} \]
Inference Rules of Hoare Logic: Program Constructs

**Seq**

\[
\[
\{P\} S_1 \{Q\}, \{Q\} S_2 \{R\} \\
\{P\} S_1; S_2 \{R\}
\]

**If**

\[
\{P \land G_1\} S_1 \{Q\}, \{P \land G_2\} S_2 \{Q\}, \ldots, \{P \land G_n\} S_n \{Q\} \\
\{P\} \text{ if } G_1 \rightarrow S_1 \mid G_2 \rightarrow S_2 \mid \ldots \mid G_n \rightarrow S_n \text{ fi } \{Q\}
\]

**Do**

\[
\{I \land G_1\} S_1 \{I\}, \{I \land G_2\} S_2 \{I\}, \ldots, \{I \land G_n\} S_n \{I\} \\
\{I\} \text{ do } G_1 \rightarrow S_1 \mid G_2 \rightarrow S_2 \mid \ldots \mid G_n \rightarrow S_n \text{ od } \{I \land \neg G_1 \land \neg G_2 \ldots \land \neg G_n\}
\]
Inference Rules of Hoare Logic: State Predicate Reasoning

\[
\begin{align*}
\text{RoC} & \quad \frac{P' \Rightarrow P, \{P\}S\{Q\}, Q \Rightarrow Q'}{\{P'\}S\{Q'\}}
\end{align*}
\]
Example

Consider the following program Pr, which should calculate the quotient and remainder of dividing x by y.

\[
\begin{align*}
& r, q := x, 0; \\
& \text{do} \\
& \quad \text{if } y \leq r \text{ then } r, q := r - y, q + 1 \\
& \text{od}
\end{align*}
\]

We would like to prove that

\[
\{x \geq 0\} \text{ Pr } \{x = q \cdot y + r \land 0 \leq r < y\}
\]

is valid.

In what follows, define \( I \triangleq (x = q \cdot y + r \land 0 \leq r) \).
Example (cont.)

By Asg, we have:

\[
\{I_{r-y, q+1}^{r, q}\} \equiv (x = (q+1) \cdot y + (r-y) \land 0 \leq r - y) \ r, \ q := r-y, \ q+1 \ \{I\} \tag{1}
\]

Since \(I_{r-y, q+1}^{r, q}\) \(\Rightarrow I \land y \leq r\), RoC and (1) give us:

\[
\{I \land y \leq r\} \ r, \ q := r-y, \ q+1 \ \{I\} \tag{2}
\]

(2) and Do now give us:

\[
\{I\} \text{ do } y \leq r \to r, \ q := r-y, \ q+1 \text{ od } \{I \land \neg(y \leq r)\}\tag{3}
\]

Asg ensures:

\[
\{I_{x, 0}^{r, q}\} \equiv (x = 0 \cdot y + x \land 0 \leq x) \ r, \ q := x, \ 0 \ \{I\} \tag{4}
\]

From (3) and (4) Seq produces:

\[
\{x = 0 \cdot y + x \land 0 \leq x\} \text{ Pr } \{I \land \neg(y \leq r)\}\tag{5}
\]

Since \((x = 0 \cdot y + x \land 0 \leq x) \Rightarrow x \geq 0\) and \((I \land \neg(y \leq r)) \Rightarrow (x = q \cdot y + r \land 0 \leq r < y)\) RoC allows us to conclude the desired result.
Reasoning in Practice: Proof Outlines

In practice proofs are more usually given as “proof outlines”.

1. State predicates are inserted into program text so that every statement (simple and compound) has a pre- and postcondition.

2. A proof outline is valid if: every embedded triple is a valid and adjacent state predicates related by implication.

   \[
   \begin{align*}
   \{x \geq 0\} \\
r, q := x, 0; \\
\{I\} \\
do \\
\{I \land y \leq r\} \\
y \leq r \rightarrow r, q := r - y, q + 1 \\
\{I\} \\
od \\
\{I \land \neg(y \leq r)\} & \{x = q \cdot y + r \land 0 \leq r\}
   \end{align*}
   \]
Reasoning in Practice: Where do Preconditions Come from?

- Begin with idea of what the result of the program $Pr$ should be, represented as a postcondition $Q$.
- Use axioms and inference rules to reason backwards to obtain a precondition $P$ for $Pr$.
- Result is a of the triple $\{P\} Pr \{Q\}$.
Reasoning in Practice: Loop Invariants

- The inference rule for do-loops requires the creation of a loop invariant which must hold each time through the loop.

- Coming up with the right loop invariants is often the trickiest aspect of sequential program verification.

- Invariants generally capture “design information” and are very useful as documentation, even if you don’t prove your programs correct.
Soundness and Relative Completeness

For general inference systems one speaks of *soundness* and *completeness*.

**Soundness**: Can only “true” things be proved?

**Completeness**: Can all “true” things be proved?

We can study these issues for Hoare Logic. Write $\vdash \{P\} S \{Q\}$ if the triple $\{P\} S \{Q\}$ can be proved. We can then state the following.

**Theorem (Soundness)** Suppose $\vdash \{P\} S \{Q\}$ holds. Then $\models \{P\} S \{Q\}$.

**Theorem (Relative Completeness)** Suppose that there is a complete proof system for establishing $P \Rightarrow Q$. Then $\models \{P\} S \{Q\}$ implies $\vdash \{P\} S \{Q\}$.

**Question** Why is only “relative completeness” possible, in general?