Lecture 9
Axiomatic Semantics
Kinds of Semantics

Operational Semantics

- Describes *how* programs compute
- Relatively easy to define
- Close connection to implementations
Kinds of Semantics

Operational Semantics
- Describes *how* programs compute
- Relatively easy to define
- Close connection to implementations

Denotational Semantics
- Describes *what* programs compute
- Solid mathematical foundation
- Simplifies equational reasoning
Kinds of Semantics

Operational Semantics
- Describes *how* programs compute
- Relatively easy to define
- Close connection to implementations

Denotational Semantics
- Describes *what* programs compute
- Solid mathematical foundation
- Simplifies equational reasoning

Axiomatic Semantics
- Describes the *properties* programs satisfy
- Useful for reasoning about correctness
Axiomatic Semantics

To define an axiomatic semantics we need:

- A language for expressing program properties
- Proof rules for establishing the validity of properties with respect to programs
Axiomatic Semantics

To define an axiomatic semantics we need:

- A language for expressing program properties
- Proof rules for establishing the validity of properties with respect to programs

Assertions:

- The value of $x$ is greater than 0
- The value of $y$ is even
- The value of $z$ is prime
Axiomatic Semantics

To define an axiomatic semantics we need:
- A language for expressing program properties
- Proof rules for establishing the validity of properties with respect to programs

Assertions:
- The value of $x$ is greater than 0
- The value of $y$ is even
- The value of $z$ is prime

Assertion Languages:
- First-order logic: $\forall, \exists, \land, \lor, x = y, R(x), \ldots$
- Temporal or modal logic: $\square, \diamond, X, U, F, \ldots$
- Special-purpose logics: Alloy, Sugar, Z3, etc.
Applications

- Proving correctness
- Documentation
- Test generation
- Symbolic execution
- Translation validation
- Bug finding
- Malware detection
Pre-Conditions and Post-conditions

Assertions are often used (informally) in code

```java
/* Precondition: 0 <= i < A.length */
/* Postcondition: returns A[i] */
public int get(int i) {
    return A[i];
}
```

These assertions are useful as documentation or run-time checks, but there is no guarantee they are correct.

**Idea:** Let’s make this rigorous by defining the semantics of the language in terms of pre-conditions and post-conditions!
Partial Correctness

Here’s the IMP syntax:

\[
\begin{align*}
  a & \in \text{Aexp} & a & ::= x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \\
  b & \in \text{Bexp} & b & ::= \text{true} \mid \text{false} \mid a_1 < a_2 \\
  c & \in \text{Com} & c & ::= \text{skip} \mid x := a \mid c_1; c_2 \\
  & & & | \quad \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c
\end{align*}
\]

A partial correctness statement is a triple:

\[
\{P\} \ c \ \{Q\}
\]

**Meaning:** If \(P\) holds, and then \(c\) executes (and terminates), then \(Q\) holds afterward.
Partial Correctness

\{x = 21\} y := x \times 2 \{y = 42\}
Partial Correctness

\[ \{ x = 21 \} \, y := x \times 2 \, \{ y = 42 \} \]

\[ \{ x = n \} \, y := x \times 2 \, \{ y = 2n \} \]
Total Correctness

Note that partial correctness specifications don’t ensure that the program will terminate—this is why they are called “partial.”

Sometimes we need to know that the program will terminate.

A total correctness statement is a triple written with square brackets:

\[
[P] c [Q]
\]

**Meaning:** if \(P\) holds, then \(c\) will terminate and \(Q\) holds after \(c\).

We’ll focus mostly on partial correctness.
Example: Partial Correctness

\{ foo = 0 \land bar = i \} 

\textbf{baz} := 0; 
\textbf{while } foo \neq bar 
\textbf{do} 
\quad \textbf{baz} := \textbf{baz} - 2; 
\quad \textbf{foo} := \textbf{foo} + 1 
\textbf{do}\}

\{ \textbf{baz} = -2 \times i \} 

\textbf{Intuition: } if we start with a store \( \sigma \) that maps \textbf{foo} to 0 and \textbf{bar} to an integer \( i \), and if the execution of the command terminates, then the final store \( \sigma' \) will map \textbf{baz} to \(-2i\).
Example: Total Correctness

\[
[\text{foo} = 0 \land \text{bar} = i \land i \geq 0]
\]

\[
baz := 0;
\]

\[
\textbf{while } \text{foo} \neq \text{bar} \n\]
\[
\textbf{do}
\]
\[
baz := baz - 2;
\]
\[
\text{foo} := \text{foo} + 1
\]

\[
[baz = -2 \times i]
\]

**Intuition:** if we start with a store $\sigma$ that maps foo to 0 and bar to a non-negative integer $i$, then the execution of the command will terminate in a final store $\sigma'$ will map baz to $-2i$. 
Another Example

\{ \text{foo} = 0 \land \text{bar} = i \} \\
\text{baz} := 0; \\
\textbf{while} \ \text{baz} \neq \text{bar} \\
\textbf{do} \\
\quad \text{baz} := \text{baz} + \text{foo}; \\
\quad \text{foo} := \text{foo} + 1 \\
\{ \text{baz} = i \} \\

Is this partial correctness statement valid?
Assertions

We define a new language syntax to write assertions:

\[ i \in \text{LVar} \]

\[ a \in \text{Aexp} ::= x \mid i \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \]

\[ P, Q \in \text{Assn} ::= \text{true} \mid \text{false} \mid a_1 < a_2 \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid P_1 \Rightarrow P_2 \mid \neg P \mid \forall i. P \mid \exists i. P \]

Assertions can introduce **logical variables**, which are different from program variables.

Note that every boolean expression \( b \) is also an assertion.
Satisfaction

Next we’ll define what it means for a store \( \sigma \) to satisfy an assertion.

To do this, we need an interpretation for the logical variables, which is like the store for program variables:

\[
I : \text{LVar} \rightarrow \text{Int}
\]
Satisfaction

Next we’ll define what it means for a store $\sigma$ to satisfy an assertion.

To do this, we need an interpretation for the logical variables, which is like the store for program variables:

$$l : \text{LVar} \rightarrow \text{Int}$$

And a denotation function for assertion arithmetic expressions, $A_i[a]$, that’s almost the same as for ordinary arithmetic:

$$A_i[n](\sigma, l) = n$$
$$A_i[x](\sigma, l) = \sigma(x)$$
$$A_i[i](\sigma, l) = l(i)$$
$$A_i[a_1 + a_2](\sigma, l) = A_i[a_1](\sigma, l) + A_i[a_2](\sigma, l)$$
Next we define the satisfaction relation for assertions, $\models_i$:

**Definition (Assertion satisfaction)**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Satisfaction Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma \models_i \textbf{true}$</td>
<td>(always)</td>
</tr>
<tr>
<td>$\sigma \models_i a_1 &lt; a_2$</td>
<td>if $\mathcal{A}_i[a_1](\sigma, l) &lt; \mathcal{A}_i[a_2](\sigma, l)$</td>
</tr>
<tr>
<td>$\sigma \models_i P_1 \land P_2$</td>
<td>if $\sigma \models_i P_1$ and $\sigma \models_i P_2$</td>
</tr>
<tr>
<td>$\sigma \models_i P_1 \lor P_2$</td>
<td>if $\sigma \not\models_i P_1$ or $\sigma \models_i P_2$</td>
</tr>
<tr>
<td>$\sigma \models_i P_1 \Rightarrow P_2$</td>
<td>if $\sigma \not\models_i P_1$ or $\sigma \models_i P_2$</td>
</tr>
<tr>
<td>$\sigma \models_i \neg P$</td>
<td>if $\sigma \not\models_i P$</td>
</tr>
<tr>
<td>$\sigma \models_i \forall i. P$</td>
<td>if $\forall k \in \text{Int. } \sigma \models_{l[i \mapsto k]} P$</td>
</tr>
<tr>
<td>$\sigma \models_i \exists i. P$</td>
<td>if $\exists k \in \text{Int. } \sigma \models_{l[i \mapsto k]} P$</td>
</tr>
</tbody>
</table>
Next we define what it means for a command $c$ to satisfy a partial correctness statement.

**Definition (Partial correctness statement satisfiability)**

A partial correctness statement $\{P\} c \{Q\}$ is satisfied in store $\sigma$ and interpretation $I$, written $\sigma \models_I \{P\} c \{Q\}$, if:

$$\forall \sigma'. \text{ if } \sigma \models_I P \text{ and } C[c]_\sigma = \sigma' \text{ then } \sigma' \models_I Q$$
Validity

Definition (Assertion validity)
An assertion $P$ is valid (written $\models P$) if it is valid in any store, under any interpretation: $\forall \sigma, l. \sigma \models_l P$

Definition (Partial correctness statement validity)
A partial correctness triple is valid (written $\models \{P\} \rightarrow \{Q\}$), if it is valid in any store and interpretation: $\forall \sigma, l. \sigma \models_l \{P\} \rightarrow \{Q\}$.

Now we know what we mean when we say “assertion $P$ holds” or “partial correctness statement $\{P\} \rightarrow \{Q\}$ is valid.”
How do we show that \( \{ P \} \ c \ \{ Q \} \) holds?

We know that \( \{ P \} \ c \ \{ Q \} \) is valid if it holds for all stores and interpretations: \( \forall \sigma, I. \sigma \models_I \{ P \} \ c \ \{ Q \} \).

Showing that \( \sigma \models_I \{ P \} \ c \ \{ Q \} \) requires reasoning about the denotation of \( c \) (because of the definition of satisfaction).
Proving Specifications

How do we show that \( \{ P \} c \{ Q \} \) holds?

We know that \( \{ P \} c \{ Q \} \) is valid if it holds for all stores and interpretations: \( \forall \sigma, l. \sigma \models_l \{ P \} c \{ Q \} \).

Showing that \( \sigma \models_l \{ P \} c \{ Q \} \) requires reasoning about the denotation of \( c \) (because of the definition of satisfaction).

We can do this manually, but there is a better way!

We can use a set of inference rules and axioms, called *Hoare rules*, to directly derive valid partial correctness statements without having to reason about stores, interpretations, and the execution of \( c \).