CS 4110

Programming Languages & Logics

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: $\forall, \exists, \wedge, \vee, x = y, R(x), \ldots$
- Temporal or modal logic: $\Box, \Diamond$, X, U, F, ...
- 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

```
/* 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:

$$
\begin{array}{llll}
a \in \textbf{Aexp} & a & ::= & x \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \\
b \in \textbf{Bexp} & b & ::= & \textbf{true} \mid \textbf{false} \mid a_1 < a_2 \\
c \in \textbf{Com} & c & ::= & \textbf{skip} \mid x := a \mid c_1; c_2 \\
 & & \mid & \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c
\end{array}
$$

A partial correctness statement is a triple:

$$\boxed{\{P\}\, c\, \{Q\}}$$

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

$$\{x = 21\}\ y := x \times 2\ \{y = 42\}$$

$$\{x = n\}\ y := x \times 2\ \{y = 2n\}$$

Question

Given the following partial correctness specification,

$$\{P\}\ \textbf{while}\ x < 0\ \textbf{do}\ x := x + 1\ \{x \geq 0\}$$

which P makes it valid?

A. **true**
B. **false**
C. $x \geq 0$
D. All of the above.
E. None of the above.

$\forall \quad P(\sigma) \wedge \langle \sigma, c \rangle \Downarrow \sigma' \Rightarrow Q(\sigma')$

Question

Given the following partial correctness specification,

$$\{P\}\ \textbf{while}\ x < 0\ \textbf{do}\ x := x + 1\ \{\textbf{false}\}$$

which P makes it valid?

A. **true**

B. **false**

C. $x \geq 0$

D. All of the above.

E. None of the above.

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:

$$\boxed{[\,P\,]\; c\; [\,Q\,]}$$

Meaning: if P holds, then c will terminate and Q holds after c.

We'll focus mostly on partial correctness.

Example: Partial Correctness

$\{\text{foo} = 0 \wedge \text{bar} = i\}$

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

$\{\text{baz} = -2 \times i\}$

Intuition: if we start with a store σ that maps foo to 0 and bar to an integer i, and if the execution of the command terminates, then the final store σ' will map baz to $-2i$.

Example: Total Correctness

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

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

$[\mathsf{baz} = -2 \times i]$

Intuition: if we start with a store σ 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 σ' will map baz to $-2i$.

Another Example

$$\{\mathsf{foo} = 0 \wedge \mathsf{bar} = i\}$$

$$\begin{array}{l} \mathsf{baz} := 0; \\ \textbf{while } \mathsf{baz} \neq \mathsf{bar} \\ \textbf{do} \\ \quad \mathsf{baz} := \mathsf{baz} + \mathsf{foo}; \\ \quad \mathsf{foo} := \mathsf{foo} + 1 \end{array}$$

$$\{\mathsf{baz} = i\}$$

Is this partial correctness statement valid?

Assertions

We define a new language syntax to write assertions:

$$
\begin{aligned}
i &\in \mathbf{LVar} \\
a \in \mathbf{Aexp} &::= x \mid i \mid n \mid a_1 + a_2 \mid a_1 \times a_2 \\
P, Q \in \mathbf{Assn} &::= \mathbf{true} \mid \mathbf{false} \\
&\mid a_1 < a_2 \\
&\mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \Rightarrow P_2 \\
&\mid \neg P \mid \forall i.\, P \mid \exists i.\, P
\end{aligned}
$$

$\forall i. \ (i < x+1)$

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 σ to satisfy an assertion.

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

$$I : \textbf{LVar} \rightarrow \textbf{Int}$$

$\forall I$

Satisfaction

Next we'll define what it means for a store σ to satisfy an assertion.

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

$$I : \mathbf{LVar} \to \mathbf{Int}$$

And a denotation function for assertion arithmetic expressions, $\mathcal{A}_i[\![a]\!]$, that's almost the same as for ordinary arithmetic:

$$\begin{aligned}
\mathcal{A}_i[\![n]\!](\sigma, I) &= n \\
\mathcal{A}_i[\![x]\!](\sigma, I) &= \sigma(x) \\
\mathcal{A}_i[\![i]\!](\sigma, I) &= I(i) \\
\mathcal{A}_i[\![a_1 + a_2]\!](\sigma, I) &= \mathcal{A}_i[\![a_1]\!](\sigma, I) + \mathcal{A}_i[\![a_2]\!](\sigma, I)
\end{aligned}$$

Satisfaction

Next we define the satisfaction relation for assertions, $\models_I$:

Definition (Assertation satisfaction)

$\sigma \models_I \textbf{true}$	(always)
$\sigma \models_I a_1 < a_2$	if $\mathcal{A}_i[\![a_1]\!](\sigma, I) < \mathcal{A}_i[\![a_2]\!](\sigma, I)$
$\sigma \models_I P_1 \wedge P_2$	if $\sigma \models_I P_1$ and $\sigma \models_I P_2$
$\sigma \models_I P_1 \vee P_2$	if $\sigma \models_I P_1$ or $\sigma \models_I P_2$
$\sigma \models_I P_1 \Rightarrow P_2$	if $\sigma \not\models_I P_1$ or $\sigma \models_I P_2$
$\sigma \models_I \neg P$	if $\sigma \not\models_I P$
$\sigma \models_I \forall i.\, P$	if $\forall k \in Int.\ \sigma \models_{I[i \mapsto k]} P$
$\sigma \models_I \exists i.\, P$	if $\exists k \in Int.\ \sigma \models_{I[i \mapsto k]} P$

$a \to b$

$\neg a \vee b$

Satisfaction

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 σ and interpretation *I*, written $\sigma \vDash_I \{P\}\ c\ \{Q\}$, if:

$$\forall \sigma'.\ \text{if } \sigma \vDash_I P \text{ and } \mathcal{C}[\![c]\!]\sigma = \sigma' \text{ then } \sigma' \vDash_I Q$$

Validity

$\models \text{true}$ $\models (x < 0) \vee (0 < x) \vee (0 = x)$

Definition (Assertion validity)

An assertion P is valid (written $\models P$) if it is valid in any store, under any interpretation: $\forall \sigma, I.\ \sigma \models_I P$

Definition (Partial correctness statement validity)

A partial correctness triple is valid (written $\models \{P\}\ c\ \{Q\}$), if it is valid in any store and interpretation: $\forall \sigma, I.\ \sigma \models_I \{P\}\ c\ \{Q\}$.

Now we know what we mean when we say "assertion P holds" or "partial correctness statement $\{P\}\ c\ \{Q\}$ is valid."

$\forall I.\ \sigma \models_I P$ $\sigma \models P$ $\models_I P$

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: $\forall \sigma, I.\ \sigma \vDash_I \{P\}\ c\ \{Q\}$.

Showing that $\sigma \vDash_I \{P\}\ c\ \{Q\}$ 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: $\forall \sigma, I.\ \sigma \vDash_I \{P\}\ c\ \{Q\}$.

Showing that $\sigma \vDash_I \{P\}\ c\ \{Q\}$ 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.