Review - Operational Semantics

- We have an imperative language with pointers and function calls
- We have defined the semantics of the language
- Operational semantics
  - Relatively simple
  - Not compositional
  - Adequate guide for an implementation
More Semantics

• There is also denotational semantics
  - Each program has a meaning in the form of a mathematical object
  - Compositional
  - More complex formalism
    • e.g. what are appropriate meanings?
• Neither is good for arguing program correctness

Axiomatic Semantics

• Usually consists of
  - A language for making assertions about programs
  - Rules for establishing when assertions hold
• Typical assertions
  - This program terminates with \( x = 0 \)
  - If this program terminates, variables \( x \) and \( y \) have the same value
  - Throughout the execution, all pointers dereferenced are non-null
• Axiomatic semantics is equivalent in expressiveness with other forms of semantics
  - Sound and complete
Languages for Assertions

- A specification language
  - Must be easy to use and expressive (conflicting needs)
  - Must have
    - Syntax: how to construct assertions
    - Semantics: what assertions mean
- Typical examples
  - Extensions of first-order logic
  - Temporal logic (used in protocol specification, hardware specification)

State-Based Assertions

- Assertions that characterize the state of the execution
  - Recall: state = state of locals + state of memory
- Our assertions will need to be able to refer to
  - Variables
  - Contents of memory
- What are not state-based assertions
  - Variable x is live
  - Lock L will be released
  - There is no correlation between the values of x and y
**An Assertion Language**

- We’ll use a fragment of first-order logic first
  
  Formulas \( P ::= A \mid T \mid \bot \mid P \land P \mid \forall x.P \mid P_1 \Rightarrow P_2 \mid \)  
  
  Atoms \( A ::= E \mid f(A_1, \ldots, A_n) \mid E_1 \leq E_2 \mid E_1 = E_2 \mid \)  

- All boolean expressions are atoms
- We can also have an arbitrary assortment of function symbols
  - \( \text{ptr}(E,T) \) - expression \( E \) denotes a pointer to \( T \)
  - \( E : \text{ptr}(T) \) - same in a different notation
  - \( \text{reachable}(E_1,E_2) \) - list cell \( E_2 \) is reachable from \( E_1 \)

**Handling Memory State**

- We want our assertion language to have a compositional semantics
  - If \( E_1 = E_2 \) then for any context \( \text{Ctx} \) we want \( \text{Ctx}[E_1] = \text{Ctx}[E_2] \)
  - Thus we have cannot have side effects in assertions

- We model the state of memory as a mapping from addresses to values
  - If \( E \) denotes an address and \( M \) a memory state then \( \text{sel}(M,E) \) denotes the contents of memory cell
  - If \( E \) denotes an address and \( V \) a value then \( \text{upd}(M,E,V) \) denotes a new memory state obtained from \( M \) by writing \( V \) at address \( E \)
**More on Memory**

- We allow variables to range over memory states
  - So we can quantify over all possible memory states
- And we use the special pseudo-variable \( \mu \) in assertions to refer to the current state of memory

- Example:

  \[
  \forall i. i \geq 0 \land i < 5 \Rightarrow \text{sel}(\mu, A + i) > 0
  \]

  says that entries 0..4 in array \( A \) are positive

---

**Semantics of Assertions**

- An assertion can hold or not in a given state
  - Equivalently, an assertion denotes a set of states
- We write \( \rho, \sigma \models P \) to say that assertion \( P \) holds in state \( \rho, \sigma \)
  - Implies that all variables in \( P \) are defined in \( \rho \)
- We define the \( \models \) judgment inductively
- And we rely on the semantics of expressions
Semantics of Assertions

- We need a new kind of values (memory values)
  
  Values \( v ::= n | a | \sigma \)

  \[
  \begin{align*}
  & s, \sigma \vdash E \quad \text{iff} \\
  & s, \sigma \vdash T \quad \text{always} \\
  & s, \sigma \vdash P_1 \land P_2 \quad \text{iff} \\
  & s, \sigma \vdash P_1 \implies P_2 \\
  & s, \sigma \vdash \forall x. P \quad \text{iff for any } v \in \text{Values} \\
  & s, \sigma \vdash P \quad \text{implies } s[\sigma] = v, \sigma \vdash P \\
  & s, \sigma \vdash P_1 \implies P_2 \quad \text{iff} \\
  & s, \sigma \vdash P_1 \implies P_2
  \end{align*}
  \]

Semantics of Memory Expressions

- We need a new kind of values (memory values)
  
  Values \( v ::= n | a | \sigma \)

  \[
  \begin{align*}
  s, \sigma \vdash \mu \Downarrow \sigma \\
  & s, \sigma \vdash E_1 \Downarrow^{s'} \sigma, \sigma \vdash E_2 \Downarrow^{s'} a \\
  & s, \sigma \vdash \text{seq}(E_1, E_2) \Downarrow^{s'} (a) \\
  & s, \sigma \vdash \text{upd}(E_1, E_2, E_3) \Downarrow^{s'} [\sigma[a := v]]
  \end{align*}
  \]
Partial Correctness Assertion

\[ \{A\} c \{B\} \text{ means that} \]

- Whenever we start the execution of \( c \) in a state that satisfies \( A \), the program either does not terminate or it terminates in a state that satisfies \( B \)

• Formally:

\[
\forall s, s', A, c, B. \quad s, s' \vdash A \quad \text{and} \quad s, s', c, s' \vdash s', s' \implies s', s' \vdash B
\]

Total Correctness Assertion

\[ [A] c [B] \text{ means that} \]

- Whenever we start the execution of \( c \) in a state that satisfies \( A \) the program does terminate in a state that satisfies \( B \)

• Formally:

\[
\forall s, s', A, c, B. \quad s, s' \vdash A \quad \text{and} \quad s, s', c, s' \vdash s', s' \implies \exists s', s'. \text{ st. } s' \vdash B
\]
**Why Aren't We Done Yet?**

- Now we can assert things about programs
- But the only way to check them is to
  - Start the program in a state that satisfies the precondition
  - Evaluate the program and get the final state
  - Verify the postcondition
- This is called testing
- Not enough
  - We cannot start the program in all states that satisfy the precondition
  - If the program is non-deterministic we cannot find all the final states
  - We cannot verify the postcondition in general

**Derivation Rules**

- We write $\vdash \{A\} \triangleleft \{B\}$ when we can derive (prove) the partial correctness assertion
  - We wish that $\vdash \{A\} \triangleleft \{B\}$ iff $\vdash \{A\} \triangleleft \{B\}$

- We write $\vdash A$ when we can derive (prove) the assertion $A$
  - We wish that $(\forall \rho \sigma. \rho, \sigma \vdash A)$ iff $\vdash A$
Derivation Rules for Assertions

- \( \vdash \top \quad \text{true} \)
- \( \vdash P \quad \text{and} \)
- \( \vdash P_1 \land P_2 \quad \text{and} \)
- \( \vdash P \Rightarrow P_2 \quad \text{impl} \)
- \( \vdash [\alpha x] P \quad \text{all} \alpha \quad \text{is fresh} \)

- We also need rules for literals
- Those are part of various theories that extend first-order logic

Hoare Rules

- \( \vdash \{B\} \text{skip} \{B\} \)
- \( \vdash \{A\} c_1 \{C\} \quad \vdash \{C\} c_2 \{B\} \)
- \( \vdash \{A\} c_1 ; c_2 \{B\} \)
- \( \vdash \{A \land E\} c_1 \{B\} \quad \vdash \{A \land \neg E\} c_2 \{B\} \)
- \( \vdash \{A\} \text{if } E \text{ then } c_1 \text{ else } c_2 \{B\} \)
Hoare Rules: while

- The rule for while is not syntax-directed
  - It requires a loop invariant

\[
\begin{align*}
\vdash & \{ A \land E \} \quad \vdash \{ A \} \\
\vdash & \{ A \} \quad \text{while} \quad E \quad \text{do} \quad \vdash \{ A \land \neg E \}
\end{align*}
\]

Hoare Rules: Assignment

- Example: \{ A \} \ x := x + 2 \ {x \geq 5}. What is A?
- General rule:

\[
\vdash \{ A \} \quad x := E \quad \vdash \{ A \}
\]

- Surprising how simple the rule is!
- Try \{ A \} \ *x = 5 \ { *x + y = 10 \}
- A is "*y = 5 or x = y"
- How come the rule does not work?
**Hoare Rules: Side-Effects**

- To correctly model store to memory we must use memory expressions

\[
\{ \exists A \left[ \text{upd}(\mu, E_1, E_2) \right] \} \quad \text{let} \ E_1 := E_2 \{ A \}
\]

- We also have some axioms for the theory of memory expressions

\[
\vdash \text{sel}(\text{upd}(M, E_1, E_2), E'_1) = E_2
\]

\[
\text{let} \ E_1 := \text{sel}(\text{upd}(M, E_1, E_2), E'_1) = \text{sel}(M, E'_1)
\]

**Loop Example**

- Want to derive that

\[
\{ x \leq 0 \} \text{while } x \leq 5 \text{ do } x := x + 1 \{ x = 6 \}
\]

\[
\vdash x \leq 6 \wedge x \leq 5 \Rightarrow x + 1 \leq 6
\]

\[
\vdash \{ x \leq 6 \wedge x \leq 5 \} x := x + 1 \{ x \leq 6 \}
\]

\[
\vdash \{ x \leq 6 \} \text{while } x \leq 5 \text{ do } x := x + 1 \{ x \leq 6 \wedge \neg x \leq 5 \}
\]

- Then, by rule of consequence we get the conclusion
- Note, it was crucial to "invent" the loop invariant
GCD Example

- Let $c$ be the program:

  
  \[
  \text{while } (x \neq y) \text{ do} \\
  \quad \text{if } (x \leq y) \\
  \quad \quad \text{then } y := y - x \\
  \quad \quad \text{else } x := x - y
  \]

- We'll derive that

  \[
  \vdash \{ x = m \land y = n \} c \{ x = \text{gcd}(m, n) \}
  \]

GCD Example (2)

- Crucial to select
  - Precondition, postcondition and loop invariant

  - Let the precondition $Pre$ be

    \[
    x = m \land y = n
    \]

  - Let the postcondition $Post$ be

    \[
    x = \text{gcd}(m, n)
    \]

  We use the loop invariant

  \[
  I \overset{\text{def}}{=} \text{gcd}(x, y) = \text{gcd}(m, n)
  \]
**GCD Example (3)**

We first use the rule of consequence to obtain the subgoal
\[ \vdash \{ I \} c \{ I \land \neg(x \neq y) \} \quad (1) \]

But we also need to prove
\[ \vdash Pre \Rightarrow I \quad (2) \]
\[ \vdash I \land \neg(x \neq y) \Rightarrow Post \quad (3) \]

Subgoal 2 reduces to
\[ x = m \land y = n \Rightarrow \gcd(x, y) = \gcd(m, n) \]

Subgoal 3 reduces to
\[ \gcd(x, y) = \gcd(m, n) \land x = y \Rightarrow x = \gcd(m, n) \]

---

**GCD Example (4)**

Now we still have to derive subgoal 1:
\[ \vdash \{ I \} c \{ I \land \neg(x \neq y) \} \]

We can apply the rule for while and we get the subgoal
\[ \vdash \{ I \land x \neq y \} d \{ I \} \quad (4) \]

where \( d \) is the body of the loop:

\[
\begin{align*}
\text{if}(x \leq y) & \\
\text{then } y & := y - x \\
\text{else } x & := x - y
\end{align*}
\]
GCD Example (5)

We can derive subgoal 4 using the rule for conditionals and we get two subgoals:

\[ \vdash \{ I \land x \neq y \land x \leq y \} y := y - x \{ I \} \quad (5) \]
\[ \vdash \{ I \land x \neq y \land x > y \} x := x - y \{ I \} \quad (6) \]

Each of the subgoals 5 and 6 can be derived using the rule of consequence followed by assignment:

\[ \vdash I \land x \neq y \land x \leq y \Rightarrow \gcd(m, n) = \gcd(x, y - x) \quad (7) \]
\[ \vdash I \land x \neq y \land x > y \Rightarrow \gcd(m, n) = \gcd(x - y, y) \quad (8) \]

GCD Example (6)

\[ \vdash I \land x \neq y \land x > y \Rightarrow \gcd(m, n) = \gcd(x - y, y) \]

- The above can be proved by realizing that \( \gcd(x, y) = \gcd(x-y, y) \)

- Q.e.d.
- This completes the proof
- We used a lot of arithmetic
- We had to invent the loop invariants