Techniques for Automated Deduction

CS 294-4
Lecture 2
Axiomatic Semantics

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 $P ::= A \mid T \mid \wedge \mid P_1 \wedge P_2 \mid \forall x.P \mid P_1 \Rightarrow P_2 \mid$

  Atoms $A ::= E \mid f(A_1,\ldots,A_n) \mid E_1 \leq E_2 \mid E_1 = E_2 \mid \ldots$
- All boolean expressions are atoms
- We can also have an arbitrary assortment of function symbols
  - ptr(E,T) - expression E denotes a pointer to T
  - E : ptr(T) - same in a different notation
  - 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 Ctx we want Ctx[E_1] = 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 sel(M,E) denotes the contents of memory cell
  - If E denotes an address and V a value then 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 μ in assertions to refer to the current state of memory
- Example:

$$\forall i.\ i \geq 0 \wedge i < 5 \Rightarrow sel(\mu, A + i) > 0$$

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 $\rho,\sigma \;||\!-\!-\; P$ to say that assertion P holds in state ρ,σ
  - Implies that all variables in P are defined in ρ
- We define the $||\!-\!-$ judgment inductively
- And we rely on the semantics of expressions

Semantics of Assertions

$\rho, \sigma \Vdash E$ iff $\rho, \sigma \vdash E \Downarrow 1$

$\rho, \sigma \Vdash T$ always

$\rho, \sigma \Vdash P_1 \wedge P_2$ iff $\rho, \sigma \Vdash P_1$ and $\rho, \sigma \Vdash P_2$

$\rho, \sigma \Vdash \forall x. P$ iff for any $v \in Values$ $\rho[x := v], \sigma \Vdash P$

$\rho, \sigma \Vdash P_1 \Rightarrow P_2$ iff

$\rho, \sigma \Vdash P_1$ implies $\rho, \sigma \Vdash P_2$

Semantics of Memory Expressions

- We need a new kind of values (memory values)

  Values v ::= n | a | σ

$$\rho, \sigma \vdash \mu \Downarrow \sigma$$

$$\frac{\rho, \sigma \vdash E_1 \Downarrow \sigma' \quad \rho, \sigma \vdash E_2 \Downarrow a}{\rho, \sigma \vdash sel(E_1, E_2) \Downarrow \sigma'(a)}$$

$$\frac{\rho, \sigma \vdash E_1 \Downarrow \sigma' \quad \rho, \sigma \vdash E_2 \Downarrow a \quad \rho, \sigma \vdash E_3 \Downarrow v}{\rho, \sigma \vdash upd(E_1, E_2, E_3) \Downarrow \sigma'[*a := v]}$$

Partial Correctness Assertion

{A} c {B} 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:

for all ρ, σ, A, c, B we say that $\Vdash \{A\} c \{B\}$ if

$\forall \rho', \sigma' \quad \rho, \sigma \Vdash A$ and $\rho, \sigma, c \Downarrow \rho', \sigma' \Longrightarrow \rho', \sigma' \Vdash B$

Total Correctness Assertion

[A] c [B] means that

- Whenever we start the execution of c in a state that satisfies A the program <u>does terminate</u> in a state that satisfies B

- Formally:

for all ρ, σ, A, c, B we say that $\Vdash [A] c [B]$ if

$\rho, \sigma \Vdash A \Longrightarrow \exists \rho', \sigma'$ s.t. $\rho, \sigma, c \Downarrow \rho', \sigma'$ and $\rho', \sigma' \Vdash B$

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

$$\frac{}{\vdash \bot}\ \text{true}_i \qquad \frac{\vdash P_1 \quad \vdash P_2}{\vdash P_1 \wedge P_2}\ \text{and}_i$$

$$\frac{\begin{array}{c}[\vdash P_1] \\ \vdots \\ \vdash P_2\end{array}}{\vdash P_1 \Rightarrow P_2}\ \text{impl}_i \qquad \frac{\vdash [u/x]P}{\vdash \forall x.\, P}\ \text{all}_i \quad u \text{ is fresh}$$

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

Hoare Rules

$$\frac{}{\vdash \{B\}\ \text{skip}\ \{B\}}$$

$$\frac{\vdash \{A\}\ c_1\ \{C\} \qquad \vdash \{C\}\ c_2\ \{B\}}{\vdash \{A\}\ c_1; c_2\ \{B\}}$$

$$\frac{\vdash \{A \wedge E\}\ c_1\ \{B\} \qquad \vdash \{A \wedge \neg E\}\ c_2\ \{B\}}{\vdash \{A\}\ \text{if } E \text{ then } c_1 \text{ else } c_2\ \{B\}}$$

Hoare Rules: while

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

$$\frac{\vdash \{A \wedge E\}\ c\ \{A\}}{\vdash \{A\}\ \text{while } E \text{ do } c\ \{A \wedge \neg E\}}$$

Hoare Rules: Assignment

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

$$\frac{}{\vdash \{A[E/x]\}\ x := E\ \{A\}}$$

- 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

$$\frac{}{\left\{ A\left[upd(\mu, E_1, E_2) / \mu \right] \right\} \; *E_1 := E_2 \; \{ A \}}$$

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

$$\frac{}{\vdash sel(upd(M, E_1, E_2), E_1) = E_2}$$

$$\frac{E_1 \neq E_1'}{\vdash sel(upd(M, E_1, E_2), E_1') = sel(M, E_1')}$$

Loop Example

- Want to derive that

  |-- {x <= 0} while x <= 5 do x := x + 1 { x = 6 }

$$\frac{\dfrac{\vdash x \leq 6 \wedge x \leq 5 \Rightarrow x + 1 \leq 6}{\vdash \{ x \leq 6 \wedge x \leq 5 \}\, x := x + 1 \,\{ x \leq 6 \}}}{\vdash \{ x \leq 6 \}\, \texttt{while}\ x \leq 5\ \texttt{do}\ x := x + 1 \,\{ x \leq 6 \wedge \neg x \leq 5 \}}$$

- 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:

```
while (x ≠ y) do
      if(x ≤ y)
        then y := y − x
        else x := x − y
```

- We'll derive that

$$\vdash \{\, x = m \wedge y = n \,\}\, c \,\{\, x = \gcd(m, n) \,\}$$

GCD Example (2)

- Crucial to select
  - Precondition, postcondition and loop invariant
    - Let the precondition Pre be

$$x = m \wedge y = n$$

    - Let the postcondition $Post$ be

$$x = \gcd(m, n)$$

We use the loop invariant

$$I \stackrel{def}{=} \gcd(x, y) = \gcd(m, n)$$

GCD Example (3)

We first use the rule of consequence to obtain the subgoal

$$\vdash \{ I \}\, c \,\{ I \wedge \neg(x \neq y) \} \qquad (1)$$

But we also need to prove

$$\vdash Pre \Rightarrow I \qquad (2)$$
$$\vdash I \wedge \neg(x \neq y) \Rightarrow Post \qquad (3)$$

Subgoal 2 reduces to

$$x = m \wedge y = n \Rightarrow \mathsf{gcd}(x, y) = \mathsf{gcd}(m, n)$$

Subgoal 3 reduces to

$$\mathsf{gcd}(x, y) = \mathsf{gcd}(m, n) \wedge x = y \Rightarrow x = \mathsf{gcd}(m, n)$$

GCD Example (4)

Now we still have to derive subgoal 1:

$$\vdash \{ I \}\, c \,\{ I \wedge \neg(x \neq y) \}$$

We can apply the rule for **`while`** and we get the subgoal

$$\vdash \{ I \wedge x \neq y \}\, d \,\{ I \} \qquad (4)$$

where d is the body of the loop:

$$\begin{array}{l} \texttt{if}(x \leq y) \\ \quad \texttt{then}\ y := y - x \\ \quad \texttt{else}\ x := x - y \end{array}$$

GCD Example (5)

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

$$\vdash \{ I \wedge x \neq y \wedge x \leq y \}\, y := y - x \,\{ I \} \quad (5)$$
$$\vdash \{ I \wedge x \neq y \wedge x > y \}\, x := x - y \,\{ I \} \quad (6)$$

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

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

GCD Example (6)

$$\vdash I \wedge x \neq y \wedge x > y \Rightarrow \gcd(m,n) = \gcd(x - y, y)$$

- 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