Lectures 20, 21: Axiomatic Semantics

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Static Analysis

Based on slides by George Necula

Remember Operational Semantics

- We have a functional language with references
- We have defined the semantics of the language
- Operational semantics
  - Relatively simple (related to state machines)
  - Not compositional (due to loops and recursive calls)
  - Adequate guide for implementing an interpreter

More kinds of semantics

- There is also denotational semantics
  - Each program has a meaning in the form of a mathematical object
  - Compositional
  - More complex formalism (depending on the mathematics used)
  - Closer to a compiler from source to math
- Neither is good for showing program correctness
  - Operational semantics requires running the code
  - Denotational semantics requires complex calculations

Axiomatic semantics

- An axiomatic semantics consists of
  - A language for making assertions about programs
  - Rules for establishing when assertions hold
- Typical assertions
  - This program terminates
  - If this program terminates, the variables x and y have the same value throughout the execution of the program
  - The array accesses are within the array bounds
- Some typical languages of assertions
  - First-order logic
  - Other logics (temporal, linear)
  - Special-purpose specification languages (Z, Larch, JML)

History

- Program verification is almost as old as programming (e.g., "Checking a Large Routine", Turing 1949)
- In the late '60s, Floyd had rules for flow-charts and Hoare had rules for structured languages
- Since then, there have been axiomatic semantics for substantial languages, and many applications
  - Program verifiers (70s and 80s)
  - PREfix: Symbolic execution for bug hunting (WinXP)
  - Software validation tools
  - Malware detection
  - Automatic test generation

Hoare said

"Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs."

–C.A.R Hoare,
"An Axiomatic Basis for Computer Programming", 1969

Dijkstra said

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Hoare also said

"It has been found a serious problem to define these languages [ALGOL, FORTRAN, COBOL] with sufficient rigor to ensure compatibility among all implementations. ...one way to achieve this would be to insist that all implementations of the language shall satisfy the axioms and rules of inference which underlie proofs of properties of programs expressed in the language. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language."

Other Applications of Axiomatic Semantics

- The project of defining and proving everything formally has not succeeded (at least not yet)
- Proving has not replaced testing and debugging (and praying)
- Applications of axiomatic semantics
  - Proving the correctness of algorithms (or finding bugs)
  - Proving the correctness of hardware descriptions (or finding bugs)
  - "extended static checking" (e.g., checking array bounds)
  - Documentation of programs and interfaces

Safety and liveness

- Partial vs. total correectness assertions
  - Safety vs. liveness properties
    - Safety: nothing “bad” happens
    - Liveness: something “good” happens eventually
  - Usually focus on safety (partial correctness)

Assertions

- The assertions we make about programs are of the form

$$\{A\}\ c\ \{B\}$$

  - If A holds in a state σ and $\langle \sigma, c \rangle \downarrow \sigma'$
  - Then B holds in σ'

- A is the *precondition* and B is the *postcondition*
- For example

$$\{x \leq y\}\ z := x;\ z := z + 1\ \{x < y\}$$

- This is called a *Hoare triple* or *Hoare assertions*

Assertions (cont'd)

- $\{A\}\ c\ \{B\}$ is a *partial correctness assertion*
  - Does not imply termination
- $[A]\ c\ [B]$ is a *total correctness assertion*
  - If A holds at state σ
  - Then there exists σ' such that $\langle\sigma, c\rangle \downarrow \sigma'$
  - And B holds in state σ'

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
- These 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

The assertion language

- We use a fragment of first-order predicate logic

$$
\begin{array}{rccl}
\text{Formulas} & A & ::= & O \mid \top \mid \bot \\
 & & \mid & A \wedge A \mid A \vee A \mid A => A \mid \forall x.A \mid \exists x.A \\
\text{Atoms} & O & ::= & f(O, \ldots, O) | e \leq e | e = e | \ldots \\
\text{Expressions} & e & ::= & n \mid \mathsf{true} \mid \mathsf{false} \mid \ldots
\end{array}
$$

- We can also have an arbitrary assortment of function symbols
  - $\mathsf{ptr}(e, T)$ – expression e denotes a pointer to a T
  - $e : \mathsf{ptr}(T)$ – same in a different notation
  - $\mathsf{reachable}(e_1, e_2)$ – list cell e_2 is reachable from e_1
  - these can be built-in or defined

Semantics of assertions

- We introduced a language of assertions, we need to assign meanings to assertions
  - *We ignore references to memory for now*
- Notation $\sigma \models A$ means that an assertion holds in a given state
  - This is well defined when σ is defined on all variables well-defined occurring in A
- The $\models$ judgment is defined inductively on the structure of assertions

Semantics of assertions (cont'd)

- Formal definition

$$
\begin{array}{lcll}
\sigma & \models & \mathsf{true} & \text{always} \\
\sigma & \models & e_1 = e_2 & \text{iff } \langle \sigma, e_1 \rangle \downarrow n_1 \text{ and } \langle \sigma, e_2 \rangle \downarrow n_2 \text{ and } n_1 = n_2 \\
\sigma & \models & e_1 \leq e_2 & \text{iff } \langle \sigma, e_1 \rangle \downarrow n_1 \text{ and } \langle \sigma, e_2 \rangle \downarrow n_2 \text{ and } n_1 \leq n_2 \\
\sigma & \models & A_1 \wedge A_2 & \text{iff } \sigma \models A_1 \text{ and } \sigma \models A_2 \\
\sigma & \models & A_1 \vee A_2 & \text{iff } \sigma \models A_1 \text{ or } \sigma \models A_2 \\
\sigma & \models & A_1 => A_2 & \text{iff } \sigma \models A_1 \text{ implies } \sigma \models A_2 \\
\sigma & \models & \forall x . A & \text{iff } \forall n \in \mathbb{Z} \,.\, \sigma[x := n] \models A \\
\sigma & \models & \exists x . A & \text{iff } \exists n \in \mathbb{Z} \,.\, \sigma[x := n] \models A
\end{array}
$$

Semantics of assertions (cont'd)

- Now we can define formally the meaning of a partial correctness assertion

$$\models \{A\}\ c\ \{B\} : \forall\sigma\,.\,\exists\sigma'\,.\,(\sigma \models A \wedge \langle\sigma, c\rangle \downarrow \sigma') \Rightarrow \sigma' \models B$$

- and the meaning of a total correctness assertion

$$\models [A]\ c\ [B]$$

  - $\forall\sigma\,.\,\exists\sigma'\,.\,(\sigma \models A \wedge \langle\sigma, c\rangle \downarrow \sigma') \Rightarrow \sigma' \models B$
  - $\forall\sigma\,.\,\sigma \models A \Rightarrow \exists\sigma'\,.\,\langle\sigma, c\rangle \downarrow \sigma'$

Is this enough?

- Now we have the formal mechanism to decide when $\{A\}\ c\ \{B\}$
  - Start the program in all states that satisfies A
  - Run the program
  - Check that each final state satisfies B
- This is exhaustive testing
- Not enough
  - Cannot try the program in all states satisfying the precondition
  - Cannot find all final states for non-deterministic programs
  - It is impossible to effectively verify the truth of a $\forall x.A$ postcondition (by using the definition of validity)

Derivations as validity witnesses

- We define a symbolic technique for deriving valid assertions from others that are known to be valid
  - We start with validity of first-order formulas
- We write $\vdash A$ when we can derive (prove) assertion A
  - We want $(\forall \sigma \,.\, \sigma \models A)$ iff $\sigma \vdash A$
- We write $\vdash \{A\}\ c\ \{B\}$ when we can derive (prove) the partial correctness assertion
  - We want $\models \{A\}\ c\ \{B\}$ iff $\vdash \{A\}\ c\ \{B\}$

Derivation rules for assertions

- The derivation rules for $\vdash A$ are the usual from first-order logic
- Axioms in natural deduction style (inference rules):

$$\frac{\vdash A \qquad \vdash B}{\vdash A \wedge B} \qquad \frac{\vdash A[\alpha/x] \qquad \alpha \text{fresh}}{\vdash \forall x.A}$$

$$\frac{\vdash \forall x.A}{\vdash A[e/x]} \qquad \frac{\vdash A \text{ implies } \vdash B}{\vdash A \Rightarrow B} \qquad \frac{\vdash A \Rightarrow B \qquad \vdash A}{\vdash B}$$

$$\frac{\vdash A[e/x]}{\vdash \exists x.A} \qquad \frac{\begin{array}{c}\vdash \exists x.A \\ \vdash A[\alpha/x] \text{ implies } \vdash B\end{array}}{\vdash B}$$

Derivation rules for triples

- Similarly, we define $\vdash \{A\}\ c\ \{B\}$ when we can derive the triple using derivation rules
- There is a derivation rule for each instruction in the language
- Plus the rule of *consequence*

$$\frac{\vdash A' \Rightarrow A \quad \vdash \{A\}\ c\ \{B\} \quad \vdash B \Rightarrow B'}{\vdash \{A'\}\ c\ \{B'\}}$$

Derivation rules for Hoare logic

- One rule for each syntactic construct

$$\frac{}{\{A\}\ \text{skip}\ \{A\}}$$

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

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

Hoare rules: loop

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

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

- Try and see what is wrong if you make changes (e.g. drop $\neg b$, relax invariant, ...)

Hoare rules: assignment

- Example: $\{A\}\ x := x + 2\ \{x \geq 5\}$. What is A?
  - A has to imply $\{x \geq 3\}$
- General rule

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

- It was simple after all...
- Example
  - Assume that x does not appear in e
  - Show that $\{\mathrm{true}\}\ x := e\ \{x = e\}$

The assignment axiom

- Hoare said:

> *"Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic."*

- One catch: aliasing
- For languages with aliasing we might need extra machinery
  - If x and y are aliased then $\{\text{true}\}\ x := 5\ \{x + y = 10\}$ is true

Multiple Hoare rules

- For some language constructs there are more than one possible rules
  - Assignment

$$\frac{}{\{A\}\ x := e\ \{\exists x_0 \,.\, A[x_0/x] \wedge x = e[x_0/x]\}}$$

  - Loop

$$\frac{\vdash A \wedge b \Rightarrow C \quad \{C\}\ c\ \{A\} \quad \vdash A \wedge \neg b \Rightarrow B}{\vdash \{A\}\ \text{while } b\text{do } c\ \{B\}}$$

  - Loop (again)

$$\frac{\{A\}\ c\ \{(b \wedge A) \vee (\neg b \wedge B)\}}{\vdash \{b \wedge A \vee \neg b \wedge B\}\ \text{while } b\text{do } c\ \{B\}}$$

Example: conditional

- Verify that $\vdash \{\text{true}\}\ \text{if } y \le 0 \text{ then } x := 1 \text{ else } x := y\ \{x > 0\}$

$$\frac{\begin{array}{c} D_1 :: \vdash \{\text{true} \wedge y \le 0\}\ x := 1\ \{x > 0\} \\ D_2 :: \vdash \{\text{true} \wedge \neg(y \le 0)\}\ x := y\ \{x > 0\} \end{array}}{\vdash \{\text{true}\}\ \text{if } y \le 0 \text{ then } x := 1 \text{ else } x := y\ \{x > 0\}}$$

- We prove D_1 using consequence and assignment

$$D_1 :: \frac{\vdash \{1 > 0\}\ x := 1\ \{x > 0\} \qquad \vdash \text{true} \wedge y \le 0 \Rightarrow 1 > 0}{\vdash \{\text{true} \wedge y \le 0\}\ x := 1\ \{x > 0\}}$$

- We prove D_2 using consequence and assignment, too

$$D_2 :: \frac{\vdash \{y > 0\}\ x := y\ \{x > 0\} \qquad \vdash \text{true} \wedge \neg(y \le 0) \Rightarrow y > 0}{\vdash \{\text{true} \wedge \neg(y \le 0)\}\ x := y\ \{x > 0\}}$$

Example: loop

- Verify that $\vdash \{x \le 0\}\ \text{while}\ x \le 5\text{do}\ x := x+1\ \{x = 6\}$
- First, we must "guess" the invariant: $x \le 6$
  - Use the loop rule to verify the while

$$\dfrac{\dfrac{\begin{array}{c}\vdash (x \le 6) \wedge (x \le 5) \Rightarrow (x+1 \le 6) \\ \vdash \{x+1 \le 6\}\ x := x+1\ \{x \le 6\}\end{array}}{\vdash \{x \le 6 \wedge x \le 5\}\ x := x+1\ \{x \le 6\}}}{\vdash \{x \le 6\}\ \text{while}\ x \le 5\text{do}\ x := x+1\ \{(x \le 6) \wedge \neg(x \le 5)\}}$$

  - Then use consequence to get the wanted property

$$\dfrac{\begin{array}{c}\vdash x \le 0 \Rightarrow x \le 6 \\ \vdash \{x \le 6\}\ \text{while}\ \ldots\ \{(x \le 6) \wedge \neg(x \le 5)\} \\ \vdash (x \le 6) \wedge \neg(x \le 5) \Rightarrow x = 6\end{array}}{\vdash \{x \le 0\}\ \text{while}\ x \le 5\text{do}\ x := x+1\ \{x = 6\}}$$

Example: loop forever

- Verify that $\vdash \{A\}\ \text{while truedo}\ c\ \{B\}$ holds for any A, B and c
- Again, we "guess" the invariant true and apply the consequence and loop rules

$$\dfrac{\begin{array}{c}\dfrac{\vdash \{\text{true} \wedge \text{true}\}\ c\ \{\text{true}\}}{\vdash \{\text{true}\}\ \text{while truedo}\ c\ \{\text{true} \wedge \neg\text{true}\}} \\ \vdash A \Rightarrow \text{true} \qquad\qquad \text{true} \wedge \neg\text{true} => \text{false}\end{array}}{\vdash \{A\}\ \text{while truedo}\ c\ \{B\}}$$

- We need an additional lemma

$$\forall c.\ \vdash \{\text{true}\}\ c\ \{\text{true}\}$$

- How do we prove it?

Example: GCD

- Let c be the program

```
while ($x \neq y$) do
  if ($x \leq y$)
    then $y := y - x$
    else $x := x - y$
```

- Verify that

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

Example: GCD (cont'd)

- The precondition Pre is $x = m \wedge y = n$
- The postcondition $Post$ is $x = \mathrm{gcd}(m, n)$
- It is crucial to select the proper loop invariant I

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

- Before applying the rule for loop, we apply the rule of consequence to reduce the problem to

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

- To do that we also need

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

- The first is

$$x = m \wedge y = n => \mathrm{gcd}(x, y) = \mathrm{gcd}(m, n)$$

- The second is

$$\mathrm{gcd}(x, y) = \mathrm{gcd}(m, n) \wedge x = y => x = \mathrm{gcd}(m, n)$$

Example: GCD (cont'd)

- We still need to verify

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

- Now we can apply the rule for loop to get

$$\vdash \{I \land x \neq y\}\ d\ \{I\}$$

- ...where d is the body of the loop

```
if ($x \leq y$)
  then $y := y - x$
  else $x := x - y$
```

Example: GCD (cont'd)

- We use the rule for conditionals to reduce the last goal to the two subgoals

$$\vdash \{I \land x \neq y \land x \leq y\}\ y := y - x\ \{I\}$$
$$\vdash \{I \land x \neq y \land \neg(x \leq y)\}\ x := x - y\ \{I\}$$

- We use consequence and assignment to reduce them to

$$\vdash I \land x \neq y \land x \leq y \Rightarrow I[(y-x)/y]$$
$$\vdash I \land x \neq y \land \neg(x \leq y) \Rightarrow I[(x-y)/x]$$

or

$$\vdash I \land x \neq y \land x \leq y \Rightarrow \text{gcd}(m,n) = \text{gcd}(x, y-x)$$
$$\vdash I \land x \neq y \land \neg(x \leq y) \Rightarrow \text{gcd}(m,n) = \text{gcd}(x-y, y)$$

Example: GCD (cont'd)

- But we can prove that $\mathrm{gcd}(x, y) = \mathrm{gcd}(x - y, y)$
- Q.E.D. –verification is done
- Things to notice
  - We used a lot of arithmetic to prove implications
  - We have to invent, or “guess” the loop invariants
- What about total correctness?

Hoare rule for function call

- If the function is not recursive, we can inline it

$$\frac{f(x_1, \ldots, x_n) = c_f \quad \vdash \{A\}\ c_f\ \{B[f/x]\}}{\vdash \{A[e_1/x_1, \ldots, e_n/x_n]\}\ x := f(e_1, \ldots, e_n)\ \{B\}}$$

- In general
  - Each function has a precondition Pre_f and a postcondition $Post_f$

$$\frac{}{\vdash \{Pre_f[e_1/x_1, \ldots, e_n/x_n]\}\ x := f(e_1, \ldots, e_n)\ \{Post_f[x/f]\}}$$

  - We verify the function body for the function pre- and postcondition

$$\vdash \{Pre_f\}\ c_f\ \{Post_f\}$$

Using Hoare rules

- Hoare rules are mostly syntax directed
- There are three issues
  - When do we apply the rule of consequence?
  - How do we prove the implications used in consequence?
  - What invariant do we use for each while loop?
- The implications of consequence are theorem proving
  - This turns out to be doable!
  - The hardest problem is the loop invariants
    Should we ask the programmer for invariants?

So far

- We have a language for asserting properties of programs
- We know when such an assertion is true
- We have a symbolic method for deriving assertions

$$\begin{tikzcd}
\begin{array}{c} A \\ \{A\}\ c\ \{B\} \end{array} \arrow[rr, "\text{meaning}"] & & \begin{array}{c} \sigma \models A \\ \models \{A\}\ c\ \{B\} \end{array} \arrow[dl, bend right, "\text{completeness}"'] \\
& \begin{array}{c} \vdash A \\ \vdash \{A\}\ c\ \{B\} \end{array} \arrow[ul, "\text{derivation}"] \arrow[ur, bend right, "\text{soundness}"'] &
\end{tikzcd}$$

Soundness of axiomatic semantics

- Formal statement
  - If $\vdash \{A\}\ c\ \{B\}$ then $\models \{A\}\ c\ \{B\}$
- Equivalently
  - For all σ, if
    - $\sigma \models A$
    - and $\langle \sigma, c \rangle \Downarrow \sigma'$
    - and $\{A\}\ c\ \{B\}$
  - Then $\sigma' \models B$

Completeness of axiomatic semantics

- If $\models \{A\}\ c\ \{B\}$ holds, can we always derive $\vdash \{A\}\ c\ \{B\}$?
- If not, then there are valid properties that we cannot verify with Hoare rules
- The good news: For our language so far, Hoare triples are complete
- The bad news: only if the underlying logic is complete
    - I.e., $\models A$ implies $\vdash A$
    - This is called relative completeness