Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Three Main Topics of the Course

I
New Programming Paradigm

II
Language Translation

III
Language Semantics

Order of Evaluation

Specification to Implementation

Semantics

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference
Dynamic semantics

- Method of describing meaning of executing a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics

Dynamic Semantics

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes

Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the structure of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations

Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from axioms and inference rules
- Mainly suited to simple imperative programming languages

Axiomatic Semantics

- Used to formally prove a property (post-condition) of the state (the values of the program variables) after the execution of program, assuming another property (pre-condition) of the state before execution
- Written:
  - \{Precondition\} Program \{Postcondition\}
- Source of idea of loop invariant

Denotational Semantics

- Construct a function \( M \) assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs
Natural Semantics

- Aka “Big Step Semantics”
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like
  \[(C, m) \Downarrow m'\]
  or
  \[(E, m) \Downarrow v\]

Simple Imperative Programming Language

- \(I \in \text{Identifiers}\)
- \(N \in \text{Numerals}\)
- \(B ::= \text{true} | \text{false} | B \& B | B \lor B | \text{not } B | E < E | E = E\)
- \(E ::= N | I | E + E | E * E | E - E | - E\)
- \(C ::= \text{skip} | C;C | I ::= E | \text{if } B \text{ then } C \text{ else } C \text{ fi} | \text{while } B \text{ do } C \text{ od}\)

Natural Semantics of Atomic Expressions

- Identifiers: \((I, m) \Downarrow m(I)\)
- Numerals are values: \((N, m) \Downarrow N\)
- Booleans: \((\text{true}, m) \Downarrow \text{true}\)
  \((\text{false}, m) \Downarrow \text{false}\)

Booleans:

\[
\begin{align*}
(B, m) \Downarrow \text{false} & \quad (B, m) \Downarrow \text{true} \\
(B \& B', m) \Downarrow \text{false} & \quad (B \& B', m) \Downarrow \text{true} \\
(B \lor B', m) \Downarrow \text{true} & \quad (B \lor B', m) \Downarrow \text{true} \\
(\text{not } B, m) \Downarrow \text{false} & \quad (\text{not } B, m) \Downarrow \text{true}
\end{align*}
\]

Relations

\[
\begin{align*}
(E, m) \Downarrow U & \quad (E', m) \Downarrow V & U \sim V = b \\
(E \sim E', m) \Downarrow b
\end{align*}
\]

Arithmetic Expressions

\[
\begin{align*}
(E, m) \Downarrow U & \quad (E', m) \Downarrow V & U \text{ op } V = N \\
(E \text{ op } E', m) \Downarrow N
\end{align*}
\]

where \(N\) is the specified value for \(U \text{ op } V\).
Commands

Skip: \((\text{skip}, m) \Downarrow m\)

Assignment: \((E, m) \Downarrow V\)
\[ (I := E, m) \Downarrow m[I ::= V] \]

Sequencing: \((C, m) \Downarrow m'\)
\[ (C', m') \Downarrow m'' \]
\[ (C; C', m) \Downarrow m'' \]

If Then Else Command

\[ (B, m) \Downarrow \text{true} \]
\[ (C, m) \Downarrow m' \]
\[ \text{if } B \text{ then } C \text{ else } C' \text{ fi, } m \Downarrow m' \]

\[ (B, m) \Downarrow \text{false} \]
\[ (C', m) \Downarrow m' \]
\[ \text{if } B \text{ then } C \text{ else } C' \text{ fi, } m \Downarrow m' \]

Example: If Then Else Rule

\[ (2, \{x->7\}) \Downarrow 2 \]
\[ (3, \{x->7\}) \Downarrow 3 \]
\[ 2 + 3, \{x->7\} \Downarrow 5 \]
\[ x, \{x->7\} \Downarrow 7 \]
\[ 5, \{x->7\} \Downarrow 5 \]
\[ y := 2 + 3, \{x->7\} \Downarrow 5 \]
\[ x > 5, \{x->7\} \Downarrow ? \]
\[ \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \]
\[ \{x->7\} \Downarrow ? \]

Example: Arith Relation

\[ ? > ? = ? \]
\[ x, \{x->7\} \Downarrow ? \]
\[ 5, \{x->7\} \Downarrow ? \]
\[ (x > 5, \{x->7\}) \Downarrow ? \]
\[ \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \]
\[ \{x->7\} \Downarrow ? \]
Example: Identifier(s)

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) &\downarrow (5, \{x \rightarrow 7\}) \uparrow 5 \\
(x > 5, \{x \rightarrow 7\}) &\downarrow ?
\end{align*}
\] 

(If \(x > 5\) then \(y := 2 + 3\) else \(y := 3 + 4\), \(\{x \rightarrow 7\}\) \(\downarrow ?\))

Example: Arith Relation

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) &\downarrow (5, \{x \rightarrow 7\}) \uparrow 5 \\
(x > 5, \{x \rightarrow 7\}) &\downarrow \text{true}
\end{align*}
\] 

(If \(x > 5\) then \(y := 2 + 3\) else \(y := 3 + 4\), \(\{x \rightarrow 7\}\) \(\downarrow ?\))

Example: If Then Else Rule

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) &\downarrow (5, \{x \rightarrow 7\}) \uparrow 5 \\
(x > 5, \{x \rightarrow 7\}) &\downarrow \text{true}
\end{align*}
\] 

(If \(x > 5\) then \(y := 2 + 3\) else \(y := 3 + 4\), \(\{x \rightarrow 7\}\) \(\downarrow ?\))

Example: Assignment

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) &\downarrow (5, \{x \rightarrow 7\}) \uparrow 5 \\
(x > 5, \{x \rightarrow 7\}) &\downarrow \text{true}
\end{align*}
\] 

(If \(x > 5\) then \(y := 2 + 3\) else \(y := 3 + 4\), \(\{x \rightarrow 7\}\) \(\downarrow ?\))

Example: Arith Op

\[
\begin{align*}
? + ? &= ? \\
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) &\downarrow (5, \{x \rightarrow 7\}) \uparrow 5 \\
(x > 5, \{x \rightarrow 7\}) &\downarrow \text{true}
\end{align*}
\] 

(If \(x > 5\) then \(y := 2 + 3\) else \(y := 3 + 4\), \(\{x \rightarrow 7\}\) \(\downarrow ?\))

Example: Numerals

\[
\begin{align*}
2 + 3 &= 5 \\
7 > 5 &= \text{true} \\
(x, \{x \rightarrow 7\}) &\downarrow (5, \{x \rightarrow 7\}) \uparrow 5 \\
(x > 5, \{x \rightarrow 7\}) &\downarrow \text{true}
\end{align*}
\] 

(If \(x > 5\) then \(y := 2 + 3\) else \(y := 3 + 4\), \(\{x \rightarrow 7\}\) \(\downarrow ?\))
Example: Arith Op

\[2 + 3 = 5\]
\[(2, {x \rightarrow 7}) \Downarrow 2, (3, {x \rightarrow 7}) \Downarrow 3\]
\[7 > 5 = \text{true}\]
\[(x, {x \rightarrow 7}) \Downarrow 7, (5, {x \rightarrow 7}) \Downarrow 5\]
\[(x > 5, {x \rightarrow 7}) \Downarrow \text{true}\]
\[
\begin{array}{l}
\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,}
\{x \rightarrow 7\} \Downarrow ?
\end{array}
\]

Example: Assignment

\[2 + 3 = 5\]
\[(2, {x \rightarrow 7}) \Downarrow 2, (3, {x \rightarrow 7}) \Downarrow 3\]
\[7 > 5 = \text{true}\]
\[(x, {x \rightarrow 7}) \Downarrow 7, (5, {x \rightarrow 7}) \Downarrow 5\]
\[(x > 5, {x \rightarrow 7}) \Downarrow \text{true}\]
\[
\begin{array}{l}
\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,}
\{x \rightarrow 7\} \Downarrow ?
\end{array}
\]

Example: If Then Else Rule

\[2 + 3 = 5\]
\[(2, {x \rightarrow 7}) \Downarrow 2, (3, {x \rightarrow 7}) \Downarrow 3\]
\[7 > 5 = \text{true}\]
\[(x, {x \rightarrow 7}) \Downarrow 7, (5, {x \rightarrow 7}) \Downarrow 5\]
\[(x > 5, {x \rightarrow 7}) \Downarrow \text{true}\]
\[
\begin{array}{l}
\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,}
\{x \rightarrow 7\} \Downarrow ?
\end{array}
\]

Let in Command

\[(E, m) \Downarrow v \quad (C, m[I \leftarrow v]) \Downarrow m' \]
\[(\text{let } I = E \text{ in } C, m) \Downarrow m''\]

Where
\[m''(y) = m'(y) \text{ for } y \neq I\]
\[m''(I) = m(I) \text{ if } m(I) \text{ is defined,}\]
\[m''(I) \text{ is undefined otherwise}\]
Comment

- Simple Imperative Programming Language introduces variables *implicitly* through assignment
- The let-in command introduces scoped variables *explicitly*
- Clash of constructs apparent in awkward semantics

Interpretation Versus Compilation

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning
- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed

Interpreter

- An **Interpreter** represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language)
- Built incrementally
  - Start with literals
  - Variables
  - Primitive operations
  - Evaluation of expressions
  - Evaluation of commands/declarations

- Takes abstract syntax trees as input
  - In simple cases could be just strings
  - One procedure for each syntactic category (nonterminal)
    - eg one for expressions, another for commands
  - If Natural semantics used, tells how to compute final value from code
  - If Transition semantics used, tells how to compute next "state"
    - To get final value, put in a loop

Natural Semantics Example

- compute_exp (Var(v), m) = look_up v m
- compute_exp (Int(n), _) = Num (n)
- ...
- compute_com(IfExp(b,c1,c2),m) =
  - if compute_exp (b,m) = Bool(true)
    then compute_com (c1,m)
  - else compute_com (c2,m)

- compute_com(While(b,c), m) =
  - if compute_exp (b,m) = Bool(false)
    then m
  - else compute_com
    (While(b,c), compute_com(c,m))

- May fail to terminate - exceed stack limits
- Returns no useful information then
Transition Semantics

- Form of operational semantics
- Describes how each program construct transforms machine state by transitions
- Rules look like 
  
  \( (C, m) \rightarrow (C', m') \) or \( (C, m) \rightarrow m' \)
- \( C, C' \) is code remaining to be executed
- \( m, m' \) represent the state/store/memory/environment
- Partial mapping from identifiers to values
- Sometimes \( m \) (or \( C \)) not needed
- Indicates exactly one step of computation

Expressions and Values

- \( C, C' \) used for commands; \( E, E' \) for expressions; \( U,V \) for values
- Special class of expressions designated as values
  - Eg 2, 3 are values, but 2+3 is only an expression
- Memory only holds values
- Other possibilities exist

Evaluation Semantics

- Transitions successfully stops when \( E/C \) is a value/memory
- Evaluation fails if no transition possible, but not at value/memory
- Value/memory is the final meaning of original expression/command (in the given state)
- Coarse semantics: final value / memory
- More fine grained: whole transition sequence

Simple Imperative Programming Language

- \( I \in \text{Identifiers} \)
- \( N \in \text{Numerals} \)
- \( B ::= true \mid false \mid B \& B \mid B \lor B \mid \text{not } B \mid E < E \mid E = E \)
- \( E ::= N \mid I \mid E + E \mid E * E \mid E - E \mid \text{- } E \)
- \( C ::= \text{skip} \mid C;C \mid I ::= E \mid \text{if } B \text{ then } C \text{ else } C \text{ fi} \mid \text{while } B \text{ do } C \text{ od} \)

Transitions for Expressions

- Numerals are values
- Boolean values = \{true, false\}
- Identifiers: \( (I,m) \rightarrow (m(I), m) \)

Boolean Operations:

- Operators: (short-circuit)
  - \( (\text{false} \& B, m) \rightarrow (\text{false},m) \)
  - \( (\text{true} \& B, m) \rightarrow (B,m) \)
  - \( (B \& B', m) \rightarrow (B'' \& B', m) \)
  - \( (\text{true} \lor B, m) \rightarrow (\text{true},m) \)
  - \( (B \lor B', m) \rightarrow (B'' \lor B', m) \)
  - \( (\text{false} \lor B, m) \rightarrow (\text{false},m) \)
  - \( (B \lor B', m) \rightarrow (B'' \lor B', m) \)
  - \( (\text{not } \text{true}, m) \rightarrow (\text{false},m) \)
  - \( (B, m) \rightarrow (B', m) \)
  - \( (\text{not } B, m) \rightarrow (\text{not } B', m) \)
Relations

\[(E, m) \rightarrow (E', m)\]
\[(E \sim E', m) \rightarrow (E' \sim E', m)\]
\[(E, m) \rightarrow (E', m)\]
\[(V \sim E, m) \rightarrow (V \sim E', m)\]
\[(U \sim V, m) \rightarrow (true, m) \text{ or } (false, m)\]

depending on whether \(U \sim V\) holds or not

Arithmetic Expressions

\[(E, m) \rightarrow (E', m)\]
\[(E \; op \; E', m) \rightarrow (E' \; op \; E', m)\]
\[(E, m) \rightarrow (E', m)\]
\[(V \; op \; E, m) \rightarrow (V \; op \; E', m)\]
\[(U \; op \; V, m) \rightarrow (N, m)\]
where \(N\) is the specified value for \(U \; op \; V\)

Commands - in English

- skip means done evaluating
- When evaluating an assignment, evaluate the expression first
- If the expression being assigned is already a value, update the memory with the new value for the identifier
- When evaluating a sequence, work on the first command in the sequence first
- If the first command evaluates to a new memory (ie completes), evaluate remainder with new memory

If Then Else Command - in English

- If the boolean guard in an if\_then\_else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.
While Command

(while \( B \) do \( C \) od, \( m \)) -->
(if \( B \) then \( C \); while \( B \) do \( C \) od else skip fi, \( m \))

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.