IMP and Operational Semantics

Lecture 2
ECS 240

Plan

- We'll study a simple imperative language IMP
  - Abstract syntax
  - Operational semantics
  - ~~Denotational semantics~~
  - Axiomatic semantics

  … and relationships between various semantics (with proofs)

- Today: operational semantics (Ch. 2 of Winskel)

Syntax of IMP

- Concrete syntax
  - The rules by which programs can be expressed as strings of characters
  - Deals with issues like keywords, identifiers, statement separators (terminators), comments, indentation, etc.
- Concrete syntax is important in practice
  - For readability, familiarity, parsing speed, effectiveness of error recovery, clarity of error messages
- Well understood principles
  - Use finite automata and context-free grammars
  - Automatic parser generators

Abstract Syntax

- We ignore parsing issues and study programs given as *abstract syntax trees (AST)*

- Abstract syntax tree is the parse tree of the program
  - Ignores issues like comment conventions
  - More convenient for formal and algorithmic manipulation
  - Fairly independent of the concrete syntax

IMP Syntactic Entities

- Int — integer literals
    - $n \in \mathbb{Z}$
- Bool — Boolean values
    - true, false
- Loc — locations (updateable variables)
    - x, y, …
- Aexp — arithmetic expressions
    - e
- Bexp — Boolean expressions
    - b
- Com — commands
    - c

Abstract Syntax (Aexp)

- Arithmetic expressions (Aexp)

$$
\begin{array}{lll}
e ::= & n & \text{for } n \in \mathbb{Z} \\
 & \mid x & \text{for } x \in Loc \\
 & \mid e_1 + e_2 & \text{for } e_1, e_2 \in Aexp \\
 & \mid e_1 - e_2 & \text{for } e_1, e_2 \in Aexp \\
 & \mid e_1 * e_2 & \text{for } e_1, e_2 \in Aexp
\end{array}
$$

- Notes:
  - Variables are not declared
  - All variables have integer type
  - No side-effects (in expressions)

Abstract Syntax (Bexp)

- Boolean expressions (Bexp)

$$
\begin{aligned}
b ::= \;& \text{true} \\
\mid\;& \text{false} \\
\mid\;& e_1 = e_2 && \text{for } e_1, e_2 \in \text{Aexp} \\
\mid\;& e_1 \leq e_2 && \text{for } e_1, e_2 \in \text{Aexp} \\
\mid\;& \neg\, b && \text{for } b \in \text{Bexp} \\
\mid\;& b_1 \wedge b_2 && \text{for } b_1, b_2 \in \text{Bexp} \\
\mid\;& b_1 \vee b_2 && \text{for } b_1, b_2 \in \text{Bexp}
\end{aligned}
$$

Abstract Syntax (Com)

- Commands (Com)

$$
\begin{aligned}
c ::= \; & \text{skip} \\
| \; & x := e && \text{for } x \in \text{Loc and } e \in \text{Aexp} \\
| \; & c_1 \,;\, c_2 && \text{for } c_1, c_2 \in \text{Com} \\
| \; & \text{if } b \text{ then } c_1 \text{ else } c_2 && \text{for } c_1, c_2 \in \text{Com and } b \in \text{Bexp} \\
| \; & \text{while } b \text{ do } c && \text{for } c \in \text{Com and } b \in \text{Bexp}
\end{aligned}
$$

- Notes:
  - The typing rules have been embedded in the syntax definition
  - Other parts are not context-free and need to be checked separately (e.g., all variables are declared)
  - Commands contain all the side-effects in the language
  - Missing: pointers, function calls

Analysis of IMP

- Questions to answer:
  - What is the "meaning" of a given IMP expression or command?
  - How would we go about evaluating IMP expressions and commands?
  - How are the evaluator and the meaning related?

An Operational Semantics

- Specifies the evaluation of expressions and commands
- Abstracts the execution of a concrete interpreter
- Depending on the form of the expression
  - 0, 1, 2, . . . don't evaluate any further.
    - They are normal forms or values.
  - $e_1 + e_2$ is evaluated by first evaluating e_1 to n_1 , then evaluating e_2 to n_2 .
    - The result of the evaluation is the literal representing $n_1 + n_2$.
  - Similar for $e_1 * e_2$

Semantics of IMP

- The meaning of IMP expressions depends on the values of variables
- The value of variables at a given moment is abstracted as a function from Loc to Z (a *state*)
- The set of all states is: $\Sigma = Loc \to Z$
- We use σ to range over Σ

Judgment

- Use <e, σ> $\Downarrow$ n to mean: e evaluates to n in state σ
  - This is a *judgment* (a statement to relate e, σ, and n)
  - We can view $\Downarrow$ as a function with two arguments: e and σ
- This formulation is called *natural operational semantics*
  - Or *big-step operational semantics*
  - The judgment relates the expression and its "meaning"
- Next, we need to specify how $\Downarrow$ is defined

Rules of Inference

- We express the evaluation as *rules of inference* for our judgment
  - called the *derivation rules* for the judgment
  - also called the *evaluation rules* (for operational semantics)

- In general, we have one rule for each language construct

- Example: $e_1 + e_2$

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \qquad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow n_1 + n_2}$$

Evaluation Rules (for Aexp)

$$\frac{}{\langle n, \sigma \rangle \Downarrow n} \qquad \frac{}{\langle x, \sigma \rangle \Downarrow \sigma(x)}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow n_1 + n_2} \qquad \frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 - e_2, \sigma \rangle \Downarrow n_1 - n_2}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 * e_2, \sigma \rangle \Downarrow n_1 * n_2}$$

- This is called *structural operational semantics*
  - rules defined based on the structure of the expression
- These rules do not impose an order of evaluation

Evaluation Rules (for Bexp)

$$\frac{}{\langle \text{true}, \sigma\rangle \Downarrow \text{true}} \qquad \frac{}{\langle \text{false}, \sigma\rangle \Downarrow \text{false}}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 = e_2, \sigma\rangle \Downarrow n_1 = n_2} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2}{\langle e_1 \leq e_2, \sigma\rangle \Downarrow n_1 \leq n_2}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \text{false}}{\langle b_1 \wedge b_2, \sigma\rangle \Downarrow \text{false}} \qquad \frac{\langle b_2, \sigma\rangle \Downarrow \text{false}}{\langle b_1 \wedge b_2, \sigma\rangle \Downarrow \text{false}}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \text{true} \quad \langle b_2, \sigma\rangle \Downarrow \text{true}}{\langle b_1 \wedge b_2, \sigma\rangle \Downarrow \text{true}}$$

How to Read the Rules?

- Forward, as inference rules
  - if we know that the hypothesis judgments hold then we can infer that the conclusion judgment also holds
  - e.g., if we know that $e_1 \Downarrow 5$ and $e_2 \Downarrow 7$, then we can infer that $e_1 + e_2 \Downarrow 12$

How to Read the Rules?

- Backward, as evaluation rules
  - Suppose we want to evaluate $e_1 + e_2$, i.e., find n s.t. $e_1 + e_2 \Downarrow n$ is derivable using the previous rules
  - By inspection of the rules we notice that the last step in the derivation of $e_1 + e_2 \Downarrow n$ **must be** the addition rule
    - the other rules have conclusions that would not match $e_1 + e_2 \Downarrow n$
    - this is called reasoning by <u>inversion</u> on the derivation rules
  - Thus we must find n_1 and n_2 such that $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ are derivable
    - This is done recursively
- Since there is exactly one rule for each kind of expression we say that the rules are <u>syntax-directed</u>
  - At each step at most one rule applies
  - This allows a simple evaluation procedure as above

Evaluation of Commands

- Evaluation of *Aexp/Bexp* produces direct results (a number or a Boolean value), but has no side-effects

- Evaluation of Com has side-effects but no direct result
  - The "result" of a Com is a new state: $\langle c, \sigma \rangle \Downarrow \sigma'$
  - The evaluation of Com may not terminate

Evaluation Rules (for Com)

$$\frac{\langle e, \sigma\rangle \Downarrow n}{\langle x := e, \sigma\rangle \Downarrow \sigma[x := n]}$$

Def: $\sigma[x{:=}\, n](x) = n$
$\sigma[x{:=}\, n](y) = \sigma(y)$

$$\frac{}{\langle \mathrm{skip}, \sigma\rangle \Downarrow \sigma}$$

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \qquad \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle c_1 ; c_2, \sigma\rangle \Downarrow \sigma''}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \mathrm{true} \qquad \langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \mathrm{if}\ b\ \mathrm{then}\ c_1\ \mathrm{else}\ c_2, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \mathrm{false} \qquad \langle c_2, \sigma\rangle \Downarrow \sigma'}{\langle \mathrm{if}\ b\ \mathrm{then}\ c_1\ \mathrm{else}\ c_2, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \mathrm{false}}{\langle \mathrm{while}\ b\ \mathrm{do}\ c, \sigma\rangle \Downarrow \sigma}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \mathrm{true} \qquad \langle c; \mathrm{while}\ b\ \mathrm{do}\ c, \sigma\rangle \Downarrow \sigma'}{\langle \mathrm{while}\ b\ \mathrm{do}\ c, \sigma \rangle \Downarrow \sigma'}$$

Notes on Evaluation of Commands

- The order of evaluation is important
  - c_1 is evaluated before c_2 in $c_1; c_2$
  - c_2 is not evaluated in "if true then c_1 else c_2"
  - c is not evaluated in "while false do c"
  - b is evaluated first in "if b then c_1 else c_2"
  - this is explicit in the evaluation rules
- The evaluation rules are <u>not syntax-directed</u>
  - See the rule for while
  - The evaluation might not terminate
- The evaluation rules suggest an interpreter
- Conditional constructs have multiple evaluation rules
  - but only one can be applied at one time

Disadvantages of Natural-Style Operational Semantics

- Natural-style semantics has two disadvantages
  - It is hard to talk about commands whose evaluation does not terminate
    - There is no σ' such that $<c, \sigma> \Downarrow \sigma'$
    - But that is true also of ill-formed or erroneous commands !
  - It does not give us a way to talk about intermediate states
    - Thus we cannot say that on a parallel machine the execution of two commands is interleaved

- *Small-step semantics* overcomes these problems
  - Execution is modeled as a (possible infinite) sequence of states

Contextual Semantics

- Contextual semantics is a small-step semantics where the atomic execution step is a rewrite of the program
- We will define a relation $\langle c, \sigma\rangle \to \langle c', \sigma'\rangle$
  - c' is obtained from c through an atomic rewrite step
  - Evaluation terminates when the program has been rewritten to a *terminal program*
    - One from which we cannot make further progress
  - For IMP the terminal command is "skip"
  - As long as the command is not "skip" we can make further progress
    - Some commands never reduce to skip (e.g., while true do skip)

What is an Atomic Reduction?

- We need to define
  - What constitutes an atomic reduction step?
    - Granularity is a choice of the semantics designer
    - e.g., choice between an addition of arbitrary integers, or an addition of 32-bit integers
  - How to select the next reduction step, when several are possible?
    - This is the order of evaluation issue

Redexes

- *A redex* is a syntactic expression or command that can be reduced (transformed) in one atomic step
- Defined as a grammar:

  r ::= x
  | $n_1 + n_2$
  | x := n
  | skip; c
  | if true then c_1 else c_2
  | if false then c_1 else c_2
  | while b do c

- For brevity, we mix expression and command redexes
- Note that (1 + 3) + 2 is not a redex, but 1 + 3 is

Local Reduction Rules for IMP

- One for each redex: $\langle r, \sigma\rangle \to \langle e, \sigma'\rangle$
  - means that in state σ, the redex r can be replaced in one step with the expression e

$\langle x, \sigma\rangle \to \langle \sigma(x), \sigma\rangle$

$\langle n_1 + n_2, \sigma\rangle \to \langle n, \sigma\rangle$ where $n = n_1 + n_2$

$\langle n_1 = n_2, \sigma\rangle \to \langle \text{true}, \sigma\rangle$ if $n_1 = n_2$

$\langle x := n, \sigma\rangle \to \langle \text{skip}, \sigma[x := n]\rangle$

$\langle \text{skip}; c, \sigma\rangle \to \langle c, \sigma\rangle$

$\langle \text{if true then } c_1 \text{ else } c_2, \sigma\rangle \to \langle c_1, \sigma\rangle$

$\langle \text{if false then } c_1 \text{ else } c_2, \sigma\rangle \to \langle c_2, \sigma\rangle$

$\langle \text{while } b \text{ do } c, \sigma\rangle \to \langle \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}, \sigma\rangle$

The Global Reduction Rule

- General idea of the contextual semantics
  - Decompose the current expression into the redex to reduce next and the remaining program
    - The remaining program is called a <u>context</u>
  - Reduce the redex "r" to some other expression "e"
  - The resulting expression consists of "e" with the original context
- We use H to range over contexts
- We write H[r] for the expression obtained by placing redex r in context H
- Now we can define a small step

$$\text{If } \langle r, \sigma \rangle \to \langle e, \sigma' \rangle \text{ then } \langle H[r], \sigma \rangle \to \langle H[e], \sigma' \rangle$$

Contexts

- A context is like an expression (or command) with a marker • in the place where the redex goes
  - Context are also called expressions with a hole
  - The marker is sometimes called a hole
  - H[r] is the expression obtained from H by replacing • with the redex r (like the substitution [r/•]H)

- Contexts are defined by a grammar:

$$H ::= \bullet \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c_1 \text{ else } c_2 \mid H;\ c$$

Contexts. Notes (I)

- A context has exactly one • marker
- A redex is never a value
- Contexts specify precisely how to find the next redex
  - Consider $e_1 + e_2$ and its decomposition as $H[r]$
  - If e_1 is n_1 and e_2 is n_2 then $H = \bullet$ and $r = n_1 + n_2$
  - If e_1 is n_1 and e_2 is not n_2 then $H = n_1 + H_2$ and $e_2 = H_2[r]$
  - If e_1 is not n_1 then $H = H_1 + e_2$ and $e_1 = H_1[r]$
  - In the last two cases the decomposition is done recursively
  - Check that in each case the solution is unique

Contextual Semantics. Notes (II).

- E.g. $c = c_1; c_2$
  - either c_1 = skip and then $c = H[\text{skip}; c_2]$ with $H = \bullet$
  - or $c_1 \neq$ skip and then $c_1 = H[r]$; so $c = H'[r]$ with $H' = H; c_2$
- E.g. c = if b then c_1 else c_2
  - either b = true or b = false and then $c = H[r]$ with $H = \bullet$
  - or b is not a value and $b = H[r]$; so $c = H'[r]$ with H' = if H then c_1 else c_2

- Decomposition theorem:

  If c is not "skip" then there exist unique H and r such that c is H[r]

  - "Exist" means progress
  - "Unique" means determinism

Contextual Semantics. Example.

- Consider the small-step evaluation of
  x := 1; x := x + 1 in the initial state [x := 0]

State	Context	Redex
<x := 1; x := x + 1, [x := 0]>	•; x := x + 1	x := 1
<skip; x := x + 1, [x := 1]>	•	skip; x := x + 1
<x := x + 1, [x := 1]>	x := • + 1	x
<x := 1 + 1, [x := 1]>	x := •	1 + 1
<x := 2, [x := 1]>	•	x := 2
<skip, [x := 2]>		

Contextual Semantics. Notes.

- What if we want to express short-circuit evaluation of $\wedge$?
  - Define the following contexts, redexes and local reduction rules

$$H ::= \ldots \mid H \wedge b_2$$

$$r ::= \ldots \mid \text{true} \wedge b \mid \text{false} \wedge b$$

$$\langle \text{true} \wedge b, \sigma \rangle \to \langle b, \sigma \rangle$$

$$\langle \text{false} \wedge b, \sigma \rangle \to \langle \text{false}, \sigma \rangle$$

  - the local reduction kicks in before b_2 is evaluated

Contextual Semantics. Notes.

- One can think of the • as representing the program counter
- The advancement rules for • are non trivial
  - At each step the entire command is decomposed
  - This makes contextual semantics inefficient to implement directly

- The major advantage of contextual semantics is that it allows a mix of local and global reduction rules
  - For IMP we have only local reduction rules: only the redex is reduced
  - Sometimes it is useful to work on the context too