Operational Semantics

Syntax vs. semantics

• Syntax = grammatical structure
• Semantics = underlying meaning

• Sentences in a language can be syntactically well-formed but semantically meaningless
  ■ if (“foo” > 37) { oogbooga(3); “baz” * “qux”; }

Syntax vs. semantics (cont’d)

• General principle: enforce correctness at the earliest stage possible
  ■ Keywords identified in lexer
  ■ Balanced ()’s enforced in parser
  ■ Types enforced afterward

• Why?
  ■ Earlier in pipeline ⇒ simpler to think about
  ■ Reporting errors is easier
    - Less transformation from original program
    - Errors may be easier to localize
  ■ Faster algorithms for detecting violations
    - Higher chance could employ them interactively in IDE

Detour: Natural deduction

• We are going to use natural deduction rules to describe semantics
  ■ So we need to understand how those work first

• Natural deduction rules provide a syntax for writing down proofs
  ■ Each rule is an axiom, or an allowable deduction
  ■ Rules are composed together
    - The result is called a derivation
  ■ The things rules prove are called judgments
Structure of a rule

- H1 ... Hn are hypotheses, C is the conclusion
- “If H1 and H2 and ... and Hn hold, then C holds”

Example: Logic

\[
\begin{array}{c}
A \\
A \land B \\
^\land I \\
A \\
\land E_L \\
B \\
\land E_R \\
A \lor B \\
A \lor B \\
\lor L \\
B \\
\lor R \\
A \lor B \\
C \\
C \\
\lor E \\
\rightarrow I \\
A \\
A \Rightarrow B \\
B \\
\Rightarrow E (modus ponens)
\end{array}
\]

Example: Logic (cont’d)

\[
\begin{array}{c}
A \\
\ldots \\
C \\
\neg A \\
\neg I (reductio ad absurdum) \\
\neg A \\
\neg E (noncontradiction)
\end{array}
\]

- Note these are axioms from classical logic

Example derivations

\[
\begin{array}{c}
A \land (B \lor C) \\
A \\
(A \land (B \lor C)) \Rightarrow A
\end{array}
\]

\[
\begin{array}{c}
A \lor (A \land B) \\
A \\
A \lor (A \land B) \Rightarrow A
\end{array}
\]
**IMP: A language of commands**

- $a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$
- $b ::= bv \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \land b_1 \mid b_0 \lor b_1$
- $c ::= \text{skip} \mid X := a \mid c_0 ; c_1 \mid \text{if } b \text{ then } c_0 \text{ else } c_1 \mid \text{while } b \text{ do } c$

  - $n \in \mathbb{N}$ = integers, $X \in \text{Var}$ = variables, $bv \in \text{Bool} = \{\text{true, false}\}$
  - Could have written:
    - $n ::= 0 \mid 1 \mid 2 \mid 3 \mid \ldots$
    - $X ::= x \mid y \mid z \mid xx \mid \ldots$
  - Notice grammar is for ASTs
    - Not concerned about issues like ambiguity, associativity, precedence
  - Syntax stratified into commands ($c$) and expressions ($a,b$)
    - Expressions have no side effects
  - No function calls (and no higher order functions)

**Program state**

- IMP contains imperative updates, so we need to model the program *state*
  - Here the state is simply the integer value of each variable
  - (Notice can’t assign a boolean to a variable, by syntax!)
- State:
  - $\sigma : \text{Var} \to \mathbb{N}$
  - A state $\sigma$ is a mapping from variables to their values

**Judgments**

- Operational semantics has three kinds of judgments
  - $\langle a, \sigma \rangle \to n$
    - In state $\sigma$, arithmetic expression $a$ evaluates to $n$
  - $\langle b, \sigma \rangle \to bv$
    - In state $\sigma$, boolean expression $b$ evaluates to true or false
  - $\langle c, \sigma \rangle \to \sigma'$
    - Running command $c$ in state $\sigma$ produces state $\sigma'$
  - Can immediately see only commands have side effects
    - Only form whose evaluation produces a new state
    - Commands also do not return values
    - Note this is math, so we express state changes by creating the new state $\sigma'$. We can’t just “mutate” $\sigma$.

**Arithmetic evaluation**

- $\langle n, \sigma \rangle \to n$
- $\langle X, \sigma \rangle \to \sigma(X)$
- $\langle a_0, \sigma \rangle \to n_0$
- $\langle a_1, \sigma \rangle \to n_1$
- $\langle a_0 + a_1, \sigma \rangle \to n_0 + n_1$
- $\langle a_0 - a_1, \sigma \rangle \to n_0 - n_1$
- $\langle a_0 \times a_1, \sigma \rangle \to n_0 \times n_1$
- $\langle a_0, \sigma \rangle \to n_0$
- $\langle a_1, \sigma \rangle \to n_1$
- $\langle a_0 \times a_1, \sigma \rangle \to n_0 \times n_1$
Arithmetic evaluation (cont’d)

- Notes:
  - Rule for variables only defined if $X$ is in $\text{dom}(\sigma)$. Otherwise the program goes wrong - it produces no final value
  - Hypotheses of last three rules stacked to save space
  - Notice difference between syntactic operators, on the left side of arrows, and mathematical operators, on the right side of arrows
  - One rule for each kind of expression
    - These are syntax-directed rules
  - In the rules, we use terminals and non-terminals in the grammar to stand for anything producible from them
    - E.g., $n$ stands for any integer; $\sigma$ for any state; etc.
  - Order of evaluation irrelevant, because there are no side effects

Sample derivation

- $1+2+3$
- $(2\times x)-4$ in $\sigma = [x\rightarrow 3]$

Correspondence to OCaml

```ocaml
(* a ::= n | X | a0+a1 | a0-a1 | a0*a1 *)

let rec aeval sigma = function
  | AInt n -> n
  | AVar n -> List.assoc n sigma
  | APlus (a1, a2) -> (aeval sigma a1) + (aeval sigma a2)
  | AMinus (a1, a2) -> (aeval sigma a1) - (aeval sigma a2)
  | ATimes (a1, a2) -> (aeval sigma a1) * (aeval sigma a2)
```

Represent $\sigma$ as an association list

Boolean evaluation

- $\langle \text{true}, \sigma \rangle \rightarrow \text{true}$
- $\langle \text{false}, \sigma \rangle \rightarrow \text{false}$
- $\langle \neg b, \sigma \rangle \rightarrow \neg \text{bv}$

- $\langle a0, \sigma \rangle \rightarrow n0$
- $\langle a1, \sigma \rangle \rightarrow n1$
- $\langle a0=1, \sigma \rangle \rightarrow n0=1$
- $\langle a0\leq a1, \sigma \rangle \rightarrow n0\leq n1$
- $\langle b0, \sigma \rangle \rightarrow \text{bv0}$
- $\langle b1, \sigma \rangle \rightarrow \text{bv1}$
- $\langle b0\land b1, \sigma \rangle \rightarrow \text{bv0}\land\text{bv1}$
- $\langle b0\lor b1, \sigma \rangle \rightarrow \text{bv0}\lor\text{bv1}$
Sample derivations

- ¬false ∧ true
- 2 ≤ X ∨ X ≤ 4 in σ = [X→3]

Correspondence to OCaml

```ocaml
(* b ::= bv | a0=a1 | a0≤a1 | ~b | b0∧b1 | b0∨b1 *)
type bexpr =
  | BV of bool
  | BEq of aexpr * aexpr
  | BLeq of aexpr * aexpr
  | BNot of bexpr
  | BAnd of bexpr * bexpr
  | BOr of bexpr * bexpr

let rec beval sigma = function
  | BV b -> b
  | BEq (a1, a2) -> (aeval sigma a1) = (aeval sigma a2)
  | BLeq (a1, a2) -> (aeval sigma a1) <= (aeval sigma a2)
  | BNot b -> not (beval sigma b)
  | BAnd (b1, b2) -> (beval sigma b1) && (beval sigma b2)
  | BOr (b1, b2) -> (beval sigma b1) || (beval sigma b2)
```

Command evaluation

- <skip, σ> → σ
- <a, σ> → n
- <X:=a, σ> → σ[X→n]

- Here σ[X→a] is the state that is the same as σ, except X now maps to a
  - (σ[X→a])(X) = a
  - (σ[X→a])(Y) = σ(Y)  X ≠ Y
- Notice order of evaluation explicit in sequence rule

Command evaluation (cont’d)

- <b, σ> → true
- <b, σ> → false
- <if b then c0 else c1, σ> → σ0
- <if b then c0 else c1, σ> → σ1

- Two rules for conditional
  - Just like in logic we needed two rules for ∧-E and ∨-I
  - Notice we specify only one command is executed
Command evaluation (cont’d)

\[ \langle b, \sigma \rangle \rightarrow \text{false} \]
\[ \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma \]

\[ \langle b, \sigma \rangle \rightarrow \text{true} \]
\[ \langle c; \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma' \]
\[ \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \sigma' \]

Sample derivations

- \( n := 3; f := 1; \text{ while } n \geq 1 \text{ do } f := f \ast n; n := n - 1 \)

Correspondence to OCaml

```ocaml
let rec ceval sigma = function
| CSkip -> sigma
| CAssn (x, a) -> (x:(aeval sigma a))::sigma
| CSeq (c0, c1) ->
  let sigma0 = ceval sigma c0 in ceval sigma0 c1
| CIf (b, c0, c1) ->
  if (beval sigma b) then (ceval sigma c0)
    else (ceval sigma c1)
| CWhile (b, c) ->
  if (beval sigma b)
    then ceval sigma (CSeq (c, CWhile(b,c)))
    else sigma
```

Big-step semantics

- Semantics given are “big step” or “natural semantics”
  - E.g., \( \langle c, \sigma \rangle \rightarrow \sigma' \)
  - Commands fully evaluated to produce the final output state, in one, big step
- Limitation: Can’t give semantics to non-terminating programs
  - We would need to work with infinite derivations, which is typically not valid
    - Requires a coinductive, rather than inductive, interpretation
Small-step semantics

- Instead, can expose intermediate steps of computation
  - $a \rightarrow_\sigma a'$
    - Evaluating $a$ one step in state $\sigma$ produces $a'$
  - $b \rightarrow_\sigma b'$
    - Evaluating $b$ one step in state $\sigma$ produces $b'$
  - $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$
    - Running command $c$ in state $\sigma$ for one step yields a new command $c'$ and new state $\sigma'$
- Note putting $\sigma$ on the arrow is just a convenience
  - Good notation for stringing evaluations together
    - $a_0 \rightarrow_\sigma a_1 \rightarrow_\sigma a_2 \rightarrow_\sigma ...$
  - Put 1 on arrow for commands just to let us distinguish different kinds of arrows

Small-step rules for arithmetic

\[
\begin{align*}
X & \rightarrow_\sigma \sigma(X) \\
\text{a0} & \rightarrow_\sigma \text{a0}' \\
\text{a1} & \rightarrow_\sigma \text{a1}' \\
p = m + n & \rightarrow_\sigma n + m \rightarrow_\sigma p
\end{align*}
\]

- Similarly for - and ×
- Notice no rule for evaluating integer $n$
  - An integer is in normal form, meaning no further evaluation is possible
- We’ve fixed the order of evaluation
  - Could also have made it non-deterministic

Context rules

- We have some rules that do the “real” work
  - The rest are context rules that define order of evaluation
- Cool trick (due to Hieb and Felleisen):
  - Define a context as a term with a “hole” in it
    - $C ::= □ | C + a | n + C | C - a | n - C | C \times a | n \times C$
  - Notice the terms generated by this grammar always have exactly one □, and it always appears at the next position that can be evaluated
  - Define $C[a]$ to be $C$ where □ is replaced by $a$
    - Ex: $(□ + 3) \times [4] = (4 + 3) \times 5$
  - Now add one, single context rule:

\[
\begin{align*}
a & \rightarrow_\sigma a' \\
C[a] & \rightarrow_\sigma C[a']
\end{align*}
\]

Small-step rules for booleans

- Very similar to arithmetic expressions
  - Too boring to write them all down...
# Small-step rules for commands

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Transition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle X:=n, \sigma \rangle)</td>
<td>(\rightarrow_1)</td>
<td>(\langle \text{skip}, \sigma[x\rightarrow n] \rangle)</td>
</tr>
<tr>
<td>(\langle \text{skip}; c_1, \sigma \rangle)</td>
<td>(\rightarrow_1)</td>
<td>(\langle c_1, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle \text{if true then } c_0 \text{ else } c_1, \sigma \rangle)</td>
<td>(\rightarrow_1)</td>
<td>(\langle c_0, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle \text{if false then } c_0 \text{ else } c_1, \sigma \rangle)</td>
<td>(\rightarrow_1)</td>
<td>(\langle c_1, \sigma \rangle)</td>
</tr>
<tr>
<td>(\langle \text{while } b \text{ do } c, \sigma \rangle)</td>
<td>(\rightarrow_1)</td>
<td>(\langle \text{if } b \text{ then (}c; \text{while } b \text{ do } c) \text{ else skip}, \sigma \rangle)</td>
</tr>
</tbody>
</table>

- Define/extend contexts (for exps, and for comms):
  - \(D ::= \square \mid D; c\)
  - \(C ::= \ldots \mid X:=C \mid \text{if } C \text{ then } c_0 \text{ else } c_1 \mid \text{while } C \text{ do } c\)
- and the context rule: \(\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle\)
  \(\langle D[c], \sigma \rangle \rightarrow_1 \langle D[c'], \sigma' \rangle\)