Small-Step Semantics

Types of Operational Semantics

- **Big-step** – evaluate an entire expression in one big step

  $e \Downarrow v$: expression e evaluates to value v

- **Small-step** – evaluation is modeled as a series of small steps

  $e \to e' \to e'' \to e''' \to \ldots \to v$

  $e \to^* v$: e evaluate to v after a series of steps

Big Step vs. Small Step

$40+4\times100 \Downarrow 440$ vs. $40+4\times100 \to 40+400$

$40+400 \to 440$

$40+4\times100 \overset{2}{\to} 440$

$40+4\times100 \overset{*}{\to} 440$

"Simple Imperative Programming Language" (IMP)

arithmetic exps: $E ::= \text{Integer} \mid \text{Var} \mid E \oplus E$

boolean exps: $B ::= \text{true} \mid \text{false} \mid E \sim E$

statements: $S ::= \text{skip}$
$\mid \text{Var} := E$
$\mid S_1 ; S_2$
$\mid \text{if } B \text{ then } S_1 \text{ else } S_2$
$\mid \text{while } B \text{ do } S$

Evaluation relations:

	(big step)	(small step)
arith exps :	$\langle E, \sigma \rangle \Downarrow_e v$	$\langle E, \sigma \rangle \rightarrow_e E$
boolean exps :	$\langle B, \sigma \rangle \Downarrow_b b$	$\langle B, \sigma \rangle \rightarrow_b B$
statements :	$\langle S, \sigma \rangle \Downarrow_s \sigma'$	$\langle S, \sigma \rangle \rightarrow_s \langle S', \sigma' \rangle$

Arithmetic Expression

LITERAL $\dfrac{}{\langle i, \sigma \rangle \to i}$ where $i \in \mathbb{Z}$

VARIABLE $\dfrac{}{\langle x, \sigma \rangle \to i}$ where $\sigma(x) = i$

ARITH $\dfrac{}{\langle i_1 \oplus i_2, \sigma \rangle \to i_3}$ where $i_3 = i_1 \oplus i_2$, $i_1, i_2, i_3 \in \mathbb{Z}$

$$\frac{\langle e_1, \sigma \rangle \to e_1'}{\langle e_1 \oplus e_2, \sigma \rangle \to \langle e_1' \oplus e_2, \sigma \rangle} \qquad \frac{\langle e_2, \sigma \rangle \to e_2'}{\langle i \oplus e_2, \sigma \rangle \to \langle i \oplus e_2', \sigma \rangle}$$

skip

$\langle \text{skip}, \sigma \rangle \rightarrow \langle \phi, \sigma \rangle$

the empty program !

i.e., $\langle \text{skip}, \sigma \rangle$ is a final configuration

Assignment

① $\langle x := v, \sigma \rangle \rightarrow \langle \text{skip}, \sigma(x := v) \rangle$

② $$\frac{\langle e, \sigma \rangle \rightarrow_e e'}{\langle x := e, \sigma \rangle \rightarrow \langle x := e', \sigma \rangle}$$

Sequencing $(S_1 ; S_2)$

$$\frac{}{\langle \text{skip} ; S, \sigma \rangle \rightarrow \langle S, \sigma \rangle}$$

$$\frac{\langle S_1, \sigma \rangle \rightarrow \langle S_1', \sigma' \rangle}{\langle S_1 ; S_2, \sigma \rangle \rightarrow \langle S_1' ; S_2, \sigma' \rangle}$$

S_2 is untouched!

if-statement

$$\frac{\langle b, \sigma \rangle \to_b b'}{\langle \text{if } b \text{ then } s_1 \text{ else } s_2, \sigma \rangle \to \langle \text{if } b' \text{ then } s_1 \text{ else } s_2, \sigma \rangle}$$

$$\langle \text{if true then } s_1 \text{ else } s_2, \sigma \rangle \to \langle s_1, \sigma \rangle$$

$$\langle \text{if false then } s_1 \text{ else } s_2, \sigma \rangle \to \langle s_2, \sigma \rangle$$

while-statement

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

e.g., evaluate $\langle n := 4 ; \text{while } n < 10 \text{ do } n := n + 8, \sigma(n := 0) \rangle$

$\rightarrow \langle \text{skip} ; \text{while } n < 10 \text{ do } n := n + 8, \sigma(n := 4) \rangle$

$\rightarrow \langle \text{while } n < 10 \text{ do } n := n + 8, \sigma(n := 4) \rangle$

$\rightarrow \langle \text{if } n < 10 \text{ then } n := n + 8 ; W \text{ else skip}, \sigma(n := 4) \rangle$

$\rightarrow \langle \text{if } 4 < 10 \text{ then } n := n + 8 ; W \text{ else skip}, \sigma(n := 4) \rangle$

$\rightarrow \langle \text{if true then } n := n + 8 ; W \text{ else skip}, \sigma(n := 4) \rangle$

$\rightarrow \langle n := n + 8 ; W, \sigma(n := 4) \rangle$

$\rightarrow \langle n := 4 + 8 ; W, \sigma(n := 4) \rangle$

$\rightarrow \langle n := 12 ; W, \sigma(n := 4) \rangle$

$\rightarrow \langle \text{skip} ; W, \sigma(n := 12) \rangle$

$\rightarrow \langle \text{while } n<10 \text{ do } n:=n+8 , \sigma(n:=12) \rangle$

$\rightarrow \langle \text{if } n<10 \text{ then } n:=n+8 ; W \text{ else skip} , \sigma(n:=12) \rangle$

$\rightarrow \langle \text{if } 12<10 \text{ then } n:=n+8 ; W \text{ else skip} , \sigma(n:=12) \rangle$

$\rightarrow \langle \text{if false then } n:=n+8 ; W \text{ else skip} , \sigma(n:=12) \rangle$

$\rightarrow \langle \text{skip}, \sigma(n:=12) \rangle$

$\langle n:=4 ; \text{while } n<10 \text{ do } n:=n+8 , \sigma(n:=0) \rangle$

$\rightarrow^* \langle \text{skip}, \sigma(n:=12) \rangle$

More Arrows!

$\rightarrow^0 \equiv$ "identity"

$\rightarrow^1 \equiv$ single step / $\rightarrow$

$\rightarrow^n \equiv \rightarrow \cdot \rightarrow^{n-1}$

$\rightarrow^* \equiv \bigcup_{i=0}^{\infty} \rightarrow^i$

$\rightarrow^+ \equiv \bigcup_{i=1}^{\infty} \rightarrow^i$

$a \leftarrow b \equiv b \rightarrow a$

$\leftrightarrow \equiv \rightarrow \cup \leftarrow$

$\leftrightarrow^* \equiv (\rightarrow \cup \leftarrow)^*$

$\leftrightarrow^*$ is likely more general than you think!

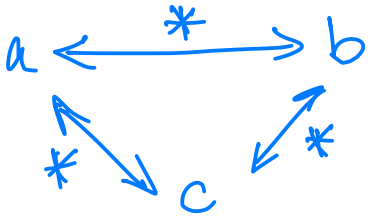
$$a \leftrightarrow^* b \not\equiv a \leftarrow^* b \cup a \rightarrow^* b$$

e.g., if $a \leftarrow c \rightarrow d \rightarrow e \leftarrow f \leftarrow g \rightarrow b$,

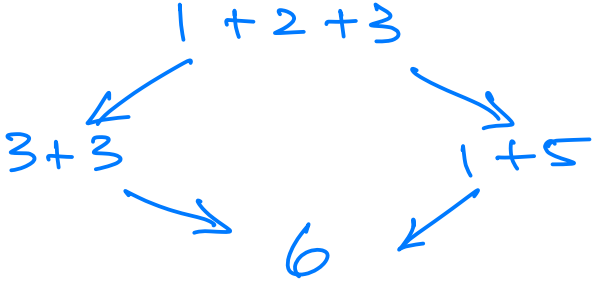
$a \leftrightarrow^* b$

Revisiting Church-Rosser

if $a \leftrightarrow^* b$, then a and b normalize to the same value!



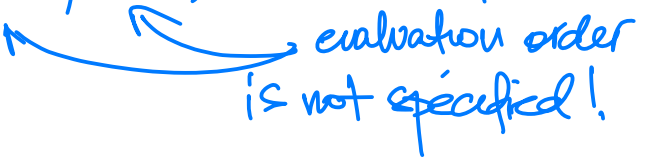
equivalently, if $a \to b_1$ and $a \to b_2$, b_1 and b_2 normalize to the same value.



"confluence"
aka.
"diamond property"

What has it ?

- Alonzo Church + J. Barkley Rosser proved that the λ-calculus has these properties in 1936!
- Most programming languages have it (sometimes)
  - does C? consider `foo(i++, i--)`



Why does it matter?

- very important for theorem provers
- one benefit: you can check if $a \equiv b$ by evaluating them.

. . . thus far, we've focused on operational semantics

i.e., what they evaluate to (big step)
+
how they are evaluated (small step)

} on some abstract machine

It would be useful if we had a way to more generally specify what a program is supposed to compute, so that we may verify its correctness

→ Axiomatic Semantics