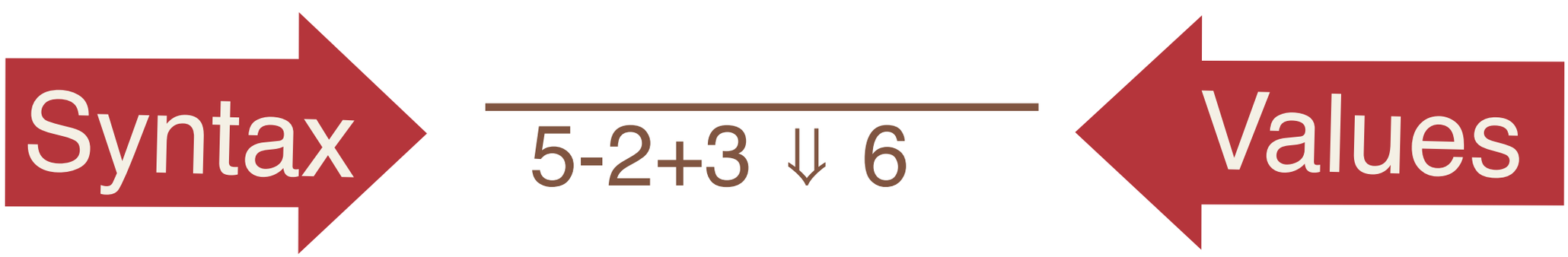
CS 565

Programming Languages (graduate)
Spring 2024

Week 7
Small-Step Operational Semantics

Big-Step Semantics

- Binary relation on pairs of syntax and values
- Read '⇓' as 'evaluates to'
- Specifies what values program can map to



- Good for whole program reasoning
  - Compiler Correctness; program equivalence;
- Bad for talking about intermediate states
  - Concurrent programs; errors

Concurrent Imp

Consider Imp with a fork operator

```
c ::= c1;c2
    | if b then ct else ce fi
    | skip
    | x := a
    | while b do c end
    | par c1 with c2 end
```

Imp Program


```
C := skip
   | x := A
   | C ; C
   | if B then C
          else C end
   | while B do C end
   | par C with C end
```

```
par
  X := 2;
  Y := 4
with
  Z := 5;
  W := 6
end
```

Small-Step

- Binary relation on pairs of expressions
- Read '$e_1 \longrightarrow e_2$' as 'reduces to'
- Specifies single transition of abstract machine
- Exposes intermediate states

Small-Step

Consider toy language:

$$E ::= C\ \mathbb{N} \mid E +_E E$$

Big-Step Semantics

$$\frac{e_n \Downarrow n \qquad e_m \Downarrow m}{e_n +_E e_m \Downarrow n + m}$$

$$\frac{}{C\ n \Downarrow n}$$

Small-Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

$$e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow \ldots \longrightarrow e_n$$

Small-Step

Consider toy language:

$$E ::= C\ \mathbb{N} \mid E +_E E$$

$$(C\ 1)+((C\ 2)+(C\ 3))+((C\ 4)+(C\ 6)))$$
$$\longrightarrow$$
$$(C\ 1)+(C\ 5)+((C\ 4)+(C\ 6)))$$
$$\longrightarrow$$
$$(C\ 1)+((C\ 5)+(C\ 10))$$
$$\longrightarrow$$
$$(C\ 1)+(C\ 15)$$
$$\longrightarrow$$
$$C\ 16$$

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

$$e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow \ldots \longrightarrow e_n$$

Small-Step

Consider toy language:

$$E ::= C\ \mathbb{N}\ |\ E +_E E$$

Two "flavors" of rule:

(Boring) Congruence rules

Interesting rules

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

$$e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow \ldots \longrightarrow e_n$$

Step Size

Big Step Semantics

$$\frac{e_n \Downarrow n \qquad e_m \Downarrow m}{e_n +_E e_m \Downarrow n + m}$$

$$\frac{}{C\ n \Downarrow n}$$

Big-Step reduction relation is from syntax, to values.

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

Small-Step reduction relation is from syntax, to syntax.

Small-Step Termination

- How to tell when we're 'done' evaluating?
- Define a class of syntactic values:

$$\frac{}{\textbf{value}\ C\ n}$$

Now we can talk about making progress

Theorem [STRONG PROGRESS]:

For any term t, either t is a value or there exists a term t' such that $t \longrightarrow t'$.

Small-Step Termination

- How to tell when we're 'done' evaluating?
- Another style of defining values:

$$v := C\ n$$

$$\frac{e_m \longrightarrow e_m'}{v +_E e_m \longrightarrow v +_E e_m'}$$

Example

How many steps does this program need to reach a value?

(C 10)+((C 12) + (C 23))

★ 0

★ 1

★ 2

★ 3

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

Concept Check

How many steps does this program need to reach a value?

C 10

★ 0

★ 1

★ 2

★ 3

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

Normal Form

A term e that isn't reducible is in normal form.

$$\neg \exists\, e'.\, e \longrightarrow e'$$

How is this different from a value?

Syntactic versus semantic.

Do not need to coincide!

Example

How might we change the reduction rules so that there are normal forms that aren't values?

$$\frac{}{\textbf{value}\ C\ n}$$

Small Step Semantics

$$\frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'}$$

$$\frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

MultiStep Relation

We generically lift single-step to full execution as the *transitive, reflexive* closure:

$$\frac{}{e \longrightarrow^* e}\ \text{REFL} \qquad \frac{e_1 \longrightarrow^* e_2 \qquad e_2 \longrightarrow e_3}{e_1 \longrightarrow^* e_3}\ \text{TRANS}$$

So: (C 1)+((C 2) + (C 3))+((C 4)+(C 6))) $\longrightarrow^*$ 16:

1+((2+3)+(4+6)) $\longrightarrow$ 1+(5+(4+6)) $\longrightarrow$ 1+(5+10)
$\longrightarrow$ 6+10 $\longrightarrow$ 16

Evaluation Order

Small step semantics give control over the order in which terms are reduced

$$\frac{e_m \longrightarrow e_m'}{e_n + e_m \longrightarrow e_n + e_m'} \qquad \frac{e_n \longrightarrow e_n'}{e_n +_E e_m \longrightarrow e_n' +_E e_m}$$

$$\frac{e_m \longrightarrow e_m'}{C\ n +_E e_m \longrightarrow C\ n +_E e_m'} \qquad \frac{}{C\ n +_E C\ m \longrightarrow C\ (n + m)}$$

Evaluation orders can be **deterministic** or **nondeterministic**

Small-Step Semantics for Imp

Inference Rules for $\longrightarrow$

$$\frac{\sigma, a_1 \Downarrow v}{\sigma, x := a_1 \longrightarrow [x \mapsto v]\sigma,\ \mathsf{skip}} \quad \textsc{CSAssn}$$

$$\frac{\sigma_1, c_1 \longrightarrow \sigma_2, c_3}{\sigma_1, c_1 ; c_2 \longrightarrow \sigma_2, c_3 ; c_2} \quad \textsc{CSSeqStep}$$

$$\frac{}{\sigma, \mathsf{skip}; c_2 \longrightarrow \sigma, c_2} \quad \textsc{CSSeqSkip}$$

Small-Step Semantics for Imp

Inference Rules for $\longrightarrow$

Reduction Rules

$$\frac{}{\sigma, \textbf{if true then } c_t \textbf{ else } c_f \textbf{ end} \longrightarrow \sigma,\ c_t}\ \textsc{CSIfT}$$

$$\frac{}{\sigma, \textbf{if false then } c_t \textbf{ else } c_f \textbf{ end} \longrightarrow \sigma,\ c_f}\ \textsc{CSIfF}$$

Congruence Rules

$$\frac{\sigma,\ b_1 \longrightarrow_B b_2}{\sigma, \textbf{if } b_1 \textbf{ then } c_t \textbf{ else } c_f \textbf{ end} \longrightarrow \sigma, \textbf{ if } b_2 \textbf{ then } c_t \textbf{ else } c_f \textbf{ end}}\ \textsc{CSIfStep}$$

Small-Step Semantics for Imp

Inference Rules for ⟶

CSWHILE

σ, **while** b **do** c ⟶

σ, **if** b **then** c; **while** b **do** c **end**

else skip **end**

Small-Step Semantics

Inference Rules for $\longrightarrow$

CSPARL

$$\frac{\sigma_1, c_1 \longrightarrow \sigma_2, c_3}{\sigma_1, \textbf{par } c_1 \textbf{ with } c_2 \longrightarrow \sigma_2, \textbf{par } c_3 \textbf{ with } c_2}$$

CSPARR

$$\frac{\sigma_1, c_2 \longrightarrow \sigma_2, c_3}{\sigma_1, \textbf{par } c_1 \textbf{ with } c_2 \longrightarrow \sigma_2, \textbf{par } c_1 \textbf{ with } c_3}$$

CSPARFINISH

$$\frac{}{\sigma, \textbf{par skip with skip} \longrightarrow \sigma, \text{skip}}$$

Imp Program


```
C := skip
   | x := A
   | C ; C
   | if B then C
          else C end
   | while B do C end
   | par C with C end
```

σ,

```
par
  X := 2;
  Y := 4
with
  X := 5;
  Y := 6
end
```

[X↦2][Y↦4]σ, skip

[X↦5][Y↦4]σ, skip

Imp Program


```
while true do
  skip
end
```

⟶

```
if true then
  while true do
    skip
  end
else skip
```

⟶

```
while true do
  skip
end
```

Summary

To recap smallstep operational semantics:

These rules form an abstract reference ‘implementation’ for a language, entirely at the level of the language itself

You can validate a particular implementation of a language against this specification (or prove that it meets it)

The rules are declarative: they don’t necessarily proscribe evaluation order, or even how to implement each step

This approach allows us to reason about properties of the language, e.g. type safety, irrespective of an implementation