Formal Operational Semantics of Programming Languages

Michael Hicks

University of Maryland

Formal Semantics of a Prog. Lang.

- Mathematical description of the meaning of programs written in that language
  - What a program computes, and what it does
  - Basis for reasoning about what programs do, either manually or automatically
- We will focus on operational semantics
  - Our running example: *Micro-Ocaml*
- Operational semantics analogous to interpretation
  - We'll write an interpreter Micro-Ocaml in OCaml
    - OK if you do not know OCaml; I'll explain as we go

Micro-OCaml Expression Grammar

e ::= ***x*** | ***n*** | ***e*** **+** ***e*** | **let** ***x*** **=** ***e*** **in** ***e***

- ***e***, ***x***, ***n*** are *meta-variables* that stand for categories of syntax
  - ***x*** is any identifier (like **z**, **y**, **foo**)
  - ***n*** is any numeral (like **1**, **0**, **10**, **-25**)
  - ***e*** is any expression (here defined, recursively!)

- *Concrete syntax* of actual expressions in **black**
  - Such as **let**, **+**, **z**, **foo**, **in**, …

- ::= and | are *meta-syntax* used to define the syntax of a language (part of "Backus-Naur form," or BNF)

Micro-OCaml Expression Grammar

`e` ::= `x` | `n` | `e + e` | `let x = e in e`

- Examples
  - `1` is a numeral `n` which is an expression `e`
  - `1+z` is an expression `e` because
    - `1` is an expression `e`,
    - `z` is an identifier `x`, which is an expression `e`, and
    - `e + e` is an expression `e`
  - `let z = 1 in 1+z` is an expression `e` because
    - `z` is an identifier `x`,
    - `1` is an expression `e` ,
    - `1+z` is an expression `e`, and
    - `let x = e in e` is an expression `e`

Abstract Syntax = Structure

- Here, the grammar for ***e*** describes an abstract syntax tree (AST), i.e., expression ***e***'s structure

$$e ::= x \mid n \mid e + e \mid \texttt{let } x = e \texttt{ in } e$$

  - An inductive definition for elements of a set ***e***

- Note: the parsing problem is how to convert text into an AST, corresponding to some data format
  - We defer worrying about this problem

Operational Semantics

- Use rules to define a judgment

$$e \Rightarrow v$$

- Says "*e evaluates to v*"
  - **e**: expression in Micro-OCaml
  - **v**: value that results from evaluating **e**
- Values just numerals for now

$$v ::= n$$

  - In a full language, values **v** will also include booleans (**true**, **false**), strings, functions, …

Defining the Semantics

- Use rules to define judgment **`e`** ⇒ **`v`**
- These rules will allow us to show things like
  - **`1+3`** ⇒ **`4`**
    - **`1+3`** is an expression **`e`**, and **`4`** is a value **`v`**
    - This judgment claims that **`1+3`** evaluates to **`4`**
    - We use rules to prove it to be true
  - **`let foo=1+2 in foo+5`** ⇒ **`8`**
  - **`let f=1+2 in let z=1 in f+z`** ⇒ **`4`**

Rules as English Text

- Suppose ***e*** is a numeral ***n***
  - Then ***e*** evaluates to itself, i.e., ***n*** ⇒ ***n***
  - Examples: `2` ⇒ `2`, `3` ⇒ `3`, *not* `2` ⇒ `7`

- Suppose ***e*** is an addition expression ***e1*** + ***e2***
  - If ***e1*** evaluates to ***n1***, i.e., ***e1*** ⇒ ***n1***
  - If ***e2*** evaluates to ***n2***, i.e., ***e2*** ⇒ ***n2***
  - Then ***e*** evaluates to ***n3***, where ***n3*** is the sum of ***n1*** and ***n2***
  - I.e., ***e1*** + ***e2*** ⇒ ***n3***
  - Examples: `3+2` ⇒ `5`, `1+1` ⇒ `2`, `1+2+3` ⇒ `6`, *not* `2+3` ⇒ `7`

Rules as English Text

No rule for ***x***

- Suppose ***e*** is a let expression **`let`** ***x*** **`=`** ***e1*** **`in`** ***e2***
  - If ***e1*** evaluates to ***v***, i.e., ***e1*** ⇒ ***v1***
  - If ***e2{v1/x}*** evaluates to ***v2***, i.e., ***e2{v1/x}*** ⇒ ***v2***
    - Here, ***e2{v1/x}*** means "the expression after substituting occurrences of ***x*** in ***e2*** with ***v1***"
  - Then ***e*** evaluates to ***v2***, i.e., **`let`** ***x*** **`=`** ***e1*** **`in`** ***e2*** ⇒ ***v2***

- Examples
  - **`let x=3 in x`** ⇒ **`3`**
  - **`let z=3 in 2+z`** ⇒ **`5`**
  - **`let y=3+2 in let x=y in x+y`** ⇒ **`10`**
  - *not* **`let x=3 in 5`** ⇒ **`7`**
  - *not* **`let x=3 in y`** ⇒ **`y`**

Rules of Inference

- We can use a more compact notation for the rules we just presented: rules of inference
  - Has the following format

$$\frac{H_1 \quad \dots \quad H_n}{C}$$

  - Says: if the conditions $H_1 \quad \dots \quad H_n$ (hypotheses) are true, then the condition C (conclusion) is true
  - If n=0 (no hypotheses) then the conclusion automatically holds; this is called an axiom
- We use inference rules to speak about evaluation

Rules of Inference: Num and Sum

- Suppose ***e*** is a numeral ***n***
  - Then ***e*** evaluates to itself, i.e., ***n*** ⇒ ***n***

$$\frac{}{n \Rightarrow n}$$

- Suppose ***e*** is an addition expression ***e1*** + ***e2***
  - If ***e1*** evaluates to ***n1***, i.e., ***e1*** ⇒ ***n1***
  - If ***e2*** evaluates to ***n2***, i.e., ***e2*** ⇒ ***n2***
  - Then ***e*** evaluates to ***n3***, where ***n3*** is the sum of ***n1*** and ***n2***
  - I.e., ***e1*** + ***e2*** ⇒ ***n3***

$$\frac{e1 \Rightarrow n1 \qquad e2 \Rightarrow n2 \qquad n3 \text{ is } n1+n2}{e1 + e2 \Rightarrow n3}$$

Rules of Inference: Let

- Suppose ***e*** is a let expression **`let`** ***x*** **`=`** ***e1*** **`in`** ***e2***
  - If ***e1*** evaluates to ***v***, i.e., ***e1*** ⇒ ***v1***
  - If ***e2{v1/x}*** evaluates to ***v2***, i.e., ***e2{v1/x}*** ⇒ ***v2***
  - Then ***e*** evaluates to ***v2***, i.e., **`let`** ***x*** **`=`** ***e1*** **`in`** ***e2*** ⇒ ***v2***

$$\frac{e1 \Rightarrow v1 \qquad e2\{v1/x\} \Rightarrow v2}{\texttt{let } x = e1 \texttt{ in } e2 \Rightarrow v2}$$

Derivations

- When we apply rules to an expression in succession, we produce a derivation
  - It's a kind of tree, rooted at the conclusion

- Produce a derivation by goal-directed search
  - Pick a rule that could prove the goal
  - Then repeatedly apply rules on the corresponding hypotheses
    - Goal: Show that `let x = 4 in x+3` $\Rightarrow$ `7`

Derivations

$$\frac{}{n \Rightarrow n}$$

$$\frac{e1 \Rightarrow n1 \quad e2 \Rightarrow n2 \quad n3 \text{ is } n1+n2}{e1 + e2 \Rightarrow n3}$$

$$\frac{e1 \Rightarrow v1 \quad e2\{v1/x\} \Rightarrow v2}{\texttt{let } x \texttt{ = } e1 \texttt{ in } e2 \Rightarrow v2}$$

Goal: show that

`let x = 4 in x+3` ⇒ `7`

$$\frac{\dfrac{}{4 \Rightarrow 4} \quad \dfrac{}{3 \Rightarrow 3} \quad 7 \text{ is } 4+3}{\dfrac{4 \Rightarrow 4 \qquad 4+3 \Rightarrow 7}{\texttt{let x = 4 in x+3} \Rightarrow 7}}$$

Quiz 1

What is a derivation of the following judgment?

$$2 + (3 + 8) \Rightarrow 13$$

(a)

$$\frac{2 \Rightarrow 2 \qquad 3 + 8 \Rightarrow 11}{2 + (3 + 8) \Rightarrow 13}$$

(b)

$$\frac{\dfrac{3 \Rightarrow 3 \qquad 8 \Rightarrow 8}{3 + 8 \Rightarrow 11} \qquad 2 \Rightarrow 2}{2 + (3 + 8) \Rightarrow 13}$$

(c)

$$\frac{2 \Rightarrow 2 \qquad \dfrac{\dfrac{8 \Rightarrow 8 \qquad 3 \Rightarrow 3}{11 \text{ is } 3+8}}{3 + 8 \Rightarrow 11} \qquad 13 \text{ is } 2+11}{2 + (3 + 8) \Rightarrow 13}$$

Quiz 1

What is a derivation of the following judgment?

$$2 + (3 + 8) \Rightarrow 13$$

(a)

$$\frac{2 \Rightarrow 2 \qquad 3 + 8 \Rightarrow 11}{2 + (3 + 8) \Rightarrow 13}$$

(b)

$$\frac{\dfrac{3 \Rightarrow 3 \qquad 8 \Rightarrow 8}{3 + 8 \Rightarrow 11} \qquad 2 \Rightarrow 2}{2 + (3 + 8) \Rightarrow 13}$$

(c)

$$\frac{2 \Rightarrow 2 \qquad \dfrac{\dfrac{8 \Rightarrow 8 \qquad 3 \Rightarrow 3}{11 \text{ is } 3+8}}{3 + 8 \Rightarrow 11} \qquad 13 \text{ is } 2+11}{2 + (3 + 8) \Rightarrow 13}$$

Semantics Defines Program Meaning

- **`e`** ⇒ **`v`** holds if and only if a *proof* can be built
  - Proofs are derivations: axioms at the top, then rules whose hypotheses have been proved to the bottom
  - No proof means **`e`** ⇏ **`v`**
- Proofs can be constructed bottom-up
  - In a goal-directed fashion
- Thus, function eval **`e`** = {**`v`** | **`e`** ⇒ **`v`** }
  - Determinism of semantics implies at most one element for any **`e`**
- So: Expression **`e`** *means* **`v`**

OpSem as Definitional Interpreter

- The rules for judgment **`e`** ⇒ **`v`** can be easily turned into code
  - The language's expressions **`e`** and values **`v`** have corresponding OCaml datatype representations **`exp`** and **`value`**
  - The semantics is represented as a function

```
eval: exp -> value
```

- This way of presenting the semantics is referred to as a definitional interpreter
  - The interpreter defines the language's meaning

Abstract Syntax for Interpreter

- Here, the grammar for ***e*** and ***v*** corresponds to the following OCaml type definitions

e ::= ***x*** | ***n*** | ***e*** + ***e*** | **let** ***x*** = ***e*** **in** ***e***

v ::= ***n***

```
type id = string
type num = int
type exp =
    | Ident of id
    | Num of num
    | Plus of exp * exp
    | Let of id * exp * exp

type value = int
```

Definitional Interpreter Code

Trace of evaluation of **`eval`** function corresponds to a derivation by the rules

- The style of rules lends itself directly to the implementation of an interpreter as a recursive function

```
let rec eval (e:exp):value =
  match e with
    Ident x -> (* no rule *)
     failwith "no value"
  | Num n -> n
  | Plus (e1,e2) ->
     let n1 = eval e1 in
     let n2 = eval e2 in
     let n3 = n1+n2 in
     n3
  | Let (x,e1,e2) ->
     let v1 = eval e1 in
     let e2' = subst v1 x e2 in
     let v2 = eval e2' in v2
```

$$\frac{}{n \Rightarrow n}$$

$$\frac{e1 \Rightarrow n1 \quad e2 \Rightarrow n2 \quad n3 \text{ is } n1+n2}{e1 + e2 \Rightarrow n3}$$

$$\frac{e1 \Rightarrow v1 \quad e2\{v1/x\} \Rightarrow v2}{\texttt{let } x \texttt{ = } e1 \texttt{ in } e2 \Rightarrow v2}$$

Derivations = Interpreter Call Trees

$$\dfrac{\dfrac{}{\texttt{4} \Rightarrow \texttt{4}} \quad \dfrac{}{\texttt{3} \Rightarrow \texttt{3}} \quad \dfrac{}{\texttt{7 is 4+3}}}{\dfrac{\dfrac{}{\texttt{4} \Rightarrow \texttt{4}} \quad \dfrac{}{\texttt{4+3} \Rightarrow \texttt{7}}}{\texttt{let x = 4 in x+3} \Rightarrow \texttt{7}}}$$

Has the same shape as the recursive call tree of the interpreter:

$$\dfrac{\dfrac{}{\texttt{eval Num 4} \Rightarrow 4} \quad \dfrac{}{\texttt{eval Num 3} \Rightarrow 3} \quad \dfrac{}{\texttt{7 is 4+3}}}{\dfrac{\dfrac{}{\texttt{eval Num 4} \Rightarrow 4} \quad \dfrac{}{\texttt{eval (subst 4 "x" Plus(Ident("x"),Num 3))} \Rightarrow 7}}{\texttt{eval Let("x",Num 4,Plus(Ident("x"),Num 3))} \Rightarrow 7}}$$

Environment-style Semantics

- The previous semantics uses substitution to handle variables
  - As we evaluate, we replace all occurrences of a variable ***x*** with value it is bound to
- An alternative semantics, closer to a real implementation, is to use an environment
  - As we evaluate, we maintain an explicit map from variables to values, and look up variables as needed

Environments

- Mathematically, an environment is a partial function from identifiers to values
  - If A is an environment, and ***x*** is an identifier, then A(***x***) can either be …
  - … a value (intuition: the variable has been declared)
  - … or undefined (intuition: variable has not been declared)
- An environment can also be thought of as a table
  - If A is

Id	Val
`x`	0
`y`	2

  - then A(`x`) is `0`, A(`y`) is `2`, and A(`z`) is undefined

Notation, Operations on Environments

- • is the empty environment (undefined for all ids)
- ***x***:***v*** is the environment that maps ***x*** to ***v*** and is undefined for all other ids
- If A and A' are environments then A, A' is the environment defined as follows

$$(A, A')(x) = \begin{cases} A'(x) & \text{if } A'(x) \text{ defined} \\ A(x) & \text{if } A'(x) \text{ undefined but } A(x) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- So: A' *shadows* definitions in A
- For brevity, can write •, A as just A

Semantics with Environments

- The environment semantics changes the judgment

$$e \Rightarrow v$$

to be

$$A;\ e \Rightarrow v$$

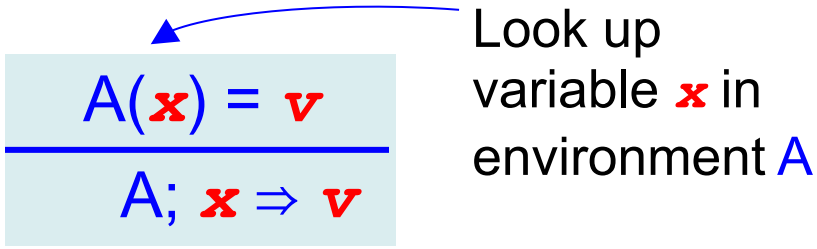
where A is an environment

  - Idea: A is used to give values to the identifiers in `e`
  - A can be thought of as containing declarations made up to `e`

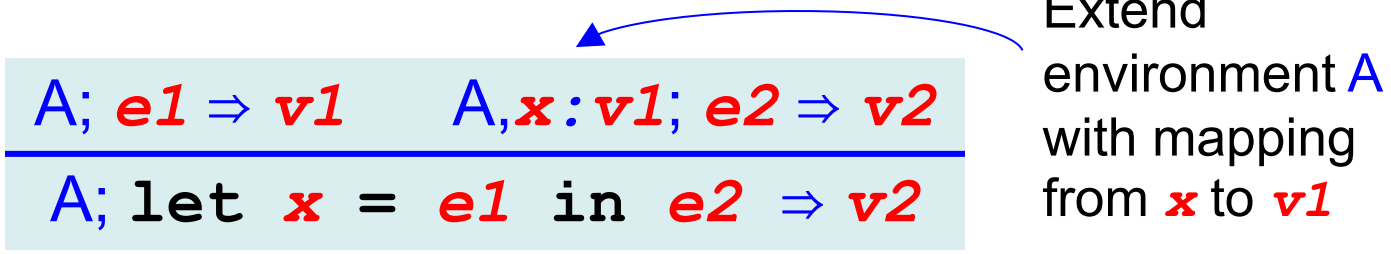
- Previous rules can be modified by

  - Inserting A everywhere in the judgments
  - Adding a rule to look up variables `x` in A
  - Modifying the rule for `let` to add `x` to A

Environment-style Rules



$$\frac{}{A;\ n \Rightarrow n}$$



$$\frac{A;\ e1 \Rightarrow n1 \qquad A;\ e2 \Rightarrow n2 \qquad n3 \text{ is } n1+n2}{A;\ e1 + e2 \Rightarrow n3}$$

Quiz 2

What is a derivation of the following judgment?

$$\bullet ; \texttt{let x=3 in x+2} \Rightarrow 5$$

(a)

$$\dfrac{\dfrac{x \Rightarrow 3 \quad 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{3 \Rightarrow 3 \quad x+2 \Rightarrow 5}}{\texttt{let x=3 in x+2} \Rightarrow 5}$$

(b)

$$\dfrac{\bullet ; 3 \Rightarrow 3 \quad \dfrac{x:3; x \Rightarrow 3 \quad x:3; 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{x:3; x+2 \Rightarrow 5}}{\bullet ; \texttt{let x=3 in x+2} \Rightarrow 5}$$

(c)

$$\dfrac{x:2; x \Rightarrow 3 \quad x:2; 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{\bullet ; \texttt{let x=3 in x+2} \Rightarrow 5}$$

Quiz 2

What is a derivation of the following judgment?

$$\bullet ; \texttt{let x=3 in x+2} \Rightarrow 5$$

(a)

$$\frac{x \Rightarrow 3 \quad 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{\dfrac{3 \Rightarrow 3 \quad x+2 \Rightarrow 5}{\texttt{let x=3 in x+2} \Rightarrow 5}}$$

(c)

$$\frac{\dfrac{x:2; x \Rightarrow 3 \quad x:2; 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{\bullet ; \texttt{let x=3 in x+2} \Rightarrow 5}}{}$$

(b)

$$\frac{\dfrac{x:3; x \Rightarrow 3 \quad x:3; 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{\bullet ; 3 \Rightarrow 3 \quad x:3; x+2 \Rightarrow 5}}{\bullet ; \texttt{let x=3 in x+2} \Rightarrow 5}$$

Definitional Interpreter: Environments

```
type env = (id * value) list

let extend env x v = (x,v)::env

let rec lookup env x =
  match env with
     [] -> failwith "no var"
  | (y,v)::env' ->
     if x = y then v
     else lookup env' x
```

Definitional Interpreter: Evaluation

```
let rec eval env e =
  match e with
     Ident x -> lookup env x
   | Num n -> n
   | Plus (e1,e2) ->
      let n1 = eval env e1 in
      let n2 = eval env e2 in
      let n3 = n1+n2 in
      n3
   | Let (x,e1,e2) ->
      let v1 = eval env e1 in
      let env' = extend env x v1 in
      let v2 = eval env' e2 in v2
```

Adding Conditionals to Micro-OCaml

```
e ::= x | v | e + e | let x = e in e
    | eq0 e | if e then e else e

v ::= n | true | false
```

- In terms of interpreter definitions:

```
type exp =
  | Val of value
  | … (* as before *)
  | Eq0 of exp
  | If of exp * exp * exp
```

```
type value =
    Int of int
  | Bool of bool
```

Rules for Eq0 and Booleans

$$\frac{}{A;\ \texttt{true} \Rightarrow \texttt{true}} \qquad \frac{A;\ e \Rightarrow 0}{A;\ \texttt{eq0}\ e \Rightarrow \texttt{true}}$$

$$\frac{}{A;\ \texttt{false} \Rightarrow \texttt{false}} \qquad \frac{A;\ e \Rightarrow v \quad v \neq 0}{A;\ \texttt{eq0}\ e \Rightarrow \texttt{false}}$$

- Booleans evaluate to themselves
  - ; **false** ⇒ **false**
- **eq0** tests for **0**
  - ; **eq0 0** ⇒ **true**
  - ; **eq0 3+4** ⇒ **false**

Rules for Conditionals

$$\frac{A;\ e1 \Rightarrow \texttt{true} \qquad A;\ e2 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 \Rightarrow v}$$

$$\frac{A;\ e1 \Rightarrow \texttt{false} \qquad A;\ e3 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 \Rightarrow v}$$

- Notice that only one branch is evaluated
  - ; `if eq0 0 then 3 else 4` ⇒ `3`
  - ; `if eq0 1 then 3 else 4` ⇒ `4`

Quiz 3

What is the derivation of the following judgment?

$$\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10$$

(a)

$$\frac{\dfrac{}{\bullet;\ 3 \Rightarrow 3} \quad \dfrac{}{\bullet;\ 2 \Rightarrow 2} \quad 3\text{-}2 \text{ is } 1}{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad \dfrac{}{\bullet;\ 10 \Rightarrow 10}$$

$$\frac{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false} \qquad \bullet;\ 10 \Rightarrow 10}{\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

(b)

$$\frac{\dfrac{}{3 \Rightarrow 3} \quad \dfrac{}{2 \Rightarrow 2} \quad 3\text{-}2 \text{ is } 1}{\texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad \dfrac{}{10 \Rightarrow 10}$$

$$\frac{\texttt{eq0 3-2} \Rightarrow \texttt{false} \qquad 10 \Rightarrow 10}{\texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

(c)

$$\frac{\dfrac{\dfrac{}{\bullet;\ 3 \Rightarrow 3} \quad \dfrac{}{\bullet;\ 2 \Rightarrow 2} \quad 3\text{-}2 \text{ is } 1}{\bullet;\ \texttt{3-2} \Rightarrow 1} \quad 1 \neq 0}{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad \dfrac{}{\bullet;\ 10 \Rightarrow 10}$$

$$\frac{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false} \qquad \bullet;\ 10 \Rightarrow 10}{\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

Quiz 3

What is the derivation of the following judgment?

$$\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10$$

(a)

$$\frac{\dfrac{\bullet;\ 3 \Rightarrow 3 \quad \bullet;\ 2 \Rightarrow 2 \quad 3\text{-}2 \text{ is } 1}{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad \bullet;\ 10 \Rightarrow 10}{\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

(b)

$$\frac{\dfrac{3 \Rightarrow 3 \quad 2 \Rightarrow 2 \quad 3\text{-}2 \text{ is } 1}{\texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad 10 \Rightarrow 10}{\texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

(c)

$$\frac{\dfrac{\dfrac{\bullet;\ 3 \Rightarrow 3 \quad \bullet;\ 2 \Rightarrow 2 \quad 3\text{-}2 \text{ is } 1}{\bullet;\ \texttt{3-2} \Rightarrow 1} \quad 1 \neq 0}{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad \bullet;\ 10 \Rightarrow 10}{\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

Updating the Interpreter

```
let rec eval env e =
  match e with
    Ident x -> lookup env x
  | Val v -> v
  | Plus (e1,e2) ->
     let Int n1 = eval env e1 in
     let Int n2 = eval env e2 in
     let n3 = n1+n2 in
     Int n3
  | Let (x,e1,e2) ->
     let v1 = eval env e1 in
     let env' = extend env x v1 in
     let v2 = eval env' e2 in v2
  | Eq0 e1 ->
     let Int n = eval env e1 in
     if n=0 then Bool true else Bool false
  | If (e1,e2,e3) ->
     let Bool b = eval env e1 in
     if b then eval env e2
     else eval env e3
```

Basically both rules for **eq0** in this one snippet

Both **if** rules here

Micro-OCaml: Syntax and Semantics

e ::= **x** | **v** | **e + e** | **let x = e in e**
| **eq0 e** | **if e then e else e**

v ::= **n** | **true** | **false**

A; **e** ⇒ **v**

"**e** *evaluates to* **v** *in environment* A"

- • is the empty environment (undefined for all ids)
- **x:v** is the environment that maps **x** to **v**
- A,A' is the concatenation of two environments

Operational Semantics Rules

$$\frac{A(x) = v}{A;\ x \Rightarrow v} \qquad \frac{}{A;\ n \Rightarrow n}$$

$$\frac{A;\ e1 \Rightarrow v1 \qquad A, x:v1;\ e2 \Rightarrow v2}{A;\ \texttt{let}\ x = e1\ \texttt{in}\ e2 \Rightarrow v2}$$

$$\frac{A;\ e1 \Rightarrow n1 \qquad A;\ e2 \Rightarrow n2 \qquad n3 \text{ is } n1+n2}{A;\ e1 + e2 \Rightarrow n3}$$

Operational Semantics Rules

$$\frac{}{A;\ \texttt{true} \Rightarrow \texttt{true}}$$

$$\frac{A;\ e \Rightarrow 0}{A;\ \texttt{eq0}\ e \Rightarrow \texttt{true}}$$

$$\frac{}{A;\ \texttt{false} \Rightarrow \texttt{false}}$$

$$\frac{A;\ e \Rightarrow v \quad v \neq 0}{A;\ \texttt{eq0}\ e \Rightarrow \texttt{false}}$$

$$\frac{A;\ e1 \Rightarrow \texttt{true} \quad A;\ e2 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 \Rightarrow v}$$

$$\frac{A;\ e1 \Rightarrow \texttt{false} \quad A;\ e3 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 \Rightarrow v}$$

Alternative: "Small step" semantics

- Judgment A; ***e*** ⇒ ***v*** relates expression ***e*** to its final value ***v*** under environment A
  - Evaluation is one "big step" to the final result
- Alternative judgment A; ***e*** → A'; ***e'*** relates ***e*** to slightly smaller ***e'***
  - Evaluation is a series of "small steps" to the final result, with intermediate environments along the way
  - I.e.: If A; ***e*** ⇒ ***v*** then
    - there exists A1; ***e1***, A2; ***e2***, …, Ak; ***ek*** and A'; ***v*** such that
    - A; ***e*** → A1; ***e1*** and A1; ***e1*** → A2; ***e2*** and so on
    - until Ak; ***ek*** → A'; ***v***.

Example Rules: If

- Computation rules

$$\frac{}{A;\ \texttt{if true then } e2 \texttt{ else } e3 \rightarrow A;\ e2}$$

$$\frac{}{A;\ \texttt{if false then } e2 \texttt{ else } e3 \rightarrow A;\ e3}$$

- Congruence rule

$$\frac{A;\ e1 \rightarrow A';\ e1'}{A;\ \texttt{if } e1 \texttt{ then } e2 \texttt{ else } e3 \rightarrow A';\ \texttt{if } e1' \texttt{ then } e2 \texttt{ else } e3}$$

Example Evaluation

One “big step”

```
•; 3 ⇒ 3
•; 2 ⇒ 2
3-2 is 1
----------
•; 3-2 ⇒ 1    1 ≠ 0
------------------
•; eq0 3-2 ⇒ false     •; 10 ⇒ 10
---------------------------------
•; if eq0 3-2 then 5 else 10 ⇒ 10
```

Three “small steps”

```
•; if eq0 3-2 then 5 else 10 → •; if eq0 1 then 5 else 10
•; if eq0 1 then 5 else 10 → •; if false then 5 else 10
•; if false then 5 else 10 → •; 10
```

Why Small Step?

- Big step semantics makes it easy to reason about complete executions
- Small step semantics makes it easier to reason about their individual steps, to
  - identify exactly where something goes wrong
  - compute the cost of an (sub)execution
  - reason about non-terminating executions

Other Styles of Semantics

- Denotational semantics: translate programs into math!
  - Usually: convert programs into functions mapping inputs to outputs
  - Analogous to compilation
- Axiomatic semantics
  - Describe programs as predicate transformers, i.e. for converting initial assumptions into guaranteed properties after execution
    - Preconditions: assumed properties of initial states
    - Postcondition: guaranteed properties of final states
  - Logical rules describe how to systematically build up these transformers from programs

Operational Semantics "Scales up"

- Operational semantics can be used to define behavior of full languages
  - With features such as (recursive) functions, records, arrays, pointers, objects, inheritance, modules, threads, input/output, …
- Example: C semantics in K framework
  - https://github.com/kframework/c-semantics
- Example: C semantics in CompCert compiler
  - http://gallium.inria.fr/~xleroy/publi/mechanized-semantics-aplas-cpp-2012.pdf
- Other styles of formal semantics don't scale well

Core Languages

- Researchers rarely define the semantics for a full language
  - Rather, they focus on a “core language” that contains the most relevant language features
  - Makes it easier to understand the key ideas

- Once proven in a core language, idea can be implemented in a full language
  - Or formalized in a larger language

Further Reading

- Practical Foundations of Programming Languages (Harper), Chaps 4-5, 7
  - https://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf
- Types and Programming Languages (Pierce), draft, Chap 6
  - http://ropas.snu.ac.kr/~kwang/520/pierce_book.pdf
- PLT Redex
  - https://redex.racket-lang.org/
  - Tutorials: http://docs.racket-lang.org/redex/