CMSC 330: Organization of Programming Languages

Operational Semantics

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

`“1+2”`

Concrete Syntax

Parse

`Plus(Int 1, Int 2)`

Abstract Syntax

- What does **`Plus(Int 1, Int 2)`** mean?

Operational semantics

- Define how programs execute
  - Often on an abstract machine (mathematical model of computer)
  - Analogous to interpretation

- We will define an operational semantics for Micro-Ocaml
  - And develop an interpreter for it, along the way

- Approach: use rules to define a judgment

$$e \Rightarrow v$$

Micro-OCaml Expression Grammar

e ::= *x* | *n* | *e* + *e* | `let` *x* `=` *e1* `in` *e2*

Corresponding AST:

```
type id = string
type exp =
    | Ident of id                (* x *)
    | Num of int                 (* n *)
    | Plus of exp * exp          (* e+e *)
    | Let of id * exp * exp (* let x=e1 in e2 *)
```

Defining the Semantics

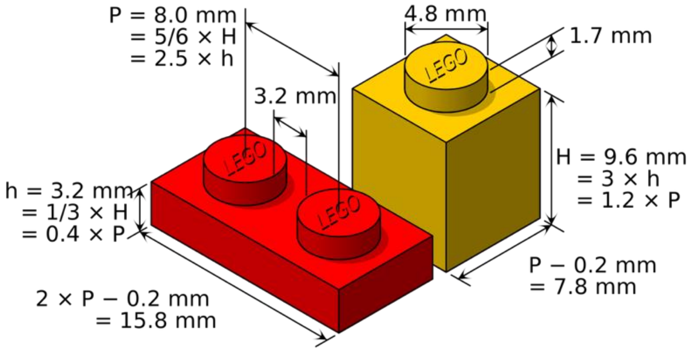
- Use rules to define judgment $e \Rightarrow v$
- Inference Rules

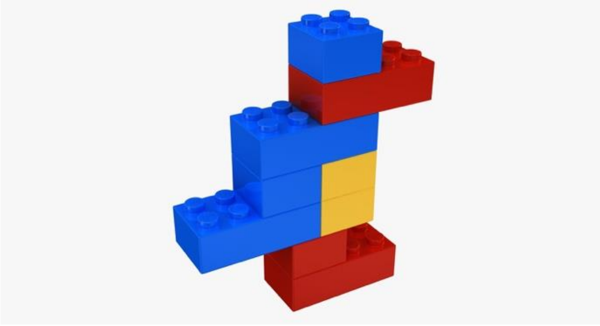
$$\frac{H_{1,}\ H_2\ \ \ldots\ \ H_n}{C}$$

$$\frac{\begin{array}{c}\forall x\left(Man(x) \to Mortal(x)\right)\\ Man(Socrates)\end{array}}{\therefore\quad Mortal(Socrates)}$$

$H_1 \wedge H_2 \wedge \ldots H_n \Rightarrow C$

Rules are Lego Blocks





Rules of Inference: Num and Sum

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

axiom

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

```
match e with
| Num n -> n
| Plus (e1,e2) ->
     let n1 = eval e1 in
     let n2 = eval e2 in
     let n3 = n1 + n2 in
     n3
```

Rules of Inference: Let

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

```
match e with
| Let (x,e1,e2) ->
     let v1 = eval e1 in
     let e2′ = subst v1 x e2 in
     let v2 = eval e2′
  in 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` ⇒ `7`

Derivations

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

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

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

Goal: show that

`let x = 4 in x+3` ⇒ `7`

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

Quiz 1

What is derivation of the following judgment?

`2 + (3 + 8) ⇒ 13`

```
(a)
2 ⇒ 2      3 + 8 ⇒ 11
---------------------
2 + (3 + 8) ⇒ 13
```

```
(b)
             8 ⇒ 8
             3 ⇒ 3
             11 is 3+8
             ---------
2 ⇒ 2      3 + 8 ⇒ 11     13 is 2+11
------------------------------------
2 + (3 + 8) ⇒ 13
```

```
(c)
3 ⇒ 3     8 ⇒ 8
-------------
3 + 8 ⇒ 11               2 ⇒ 2
-------------------------------
2 + (3 + 8) ⇒ 13
```

Quiz 1

What is derivation of the following judgment?

`2 + (3 + 8) ⇒ 13`

```
(a)
2 ⇒ 2      3 + 8 ⇒ 11
---------------------
2 + (3 + 8)  ⇒  13
```

```
(b)
            8 ⇒ 8
            3 ⇒ 3
            11 is 3+8
            ---------
2 ⇒ 2       3 + 8 ⇒ 11      13 is 2+11
-------------------------
2 + (3 + 8)  ⇒  13
```

```
(c)
3 ⇒ 3     8 ⇒ 8
------------
3 + 8 ⇒ 11             2 ⇒ 2
-------------------------
2 + (3 + 8)  ⇒  13
```

Definitional Interpreter

- 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
```

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

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

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

Derivations = Interpreter Call Trees

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

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

$$\dfrac{\mathtt{eval\ Num\ 4} \Rightarrow 4 \qquad \dfrac{\mathtt{eval\ Num\ 4} \Rightarrow 4 \quad \mathtt{eval\ Num\ 3} \Rightarrow 3 \quad \mathtt{7} \text{ is } \mathtt{4}+\mathtt{3}}{\begin{array}{l}\mathtt{eval\ (subst\ 4\ “x”} \\ \quad \mathtt{Plus(Ident(“x”),Num\ 3))} \Rightarrow 7\end{array}}}{\mathtt{eval\ Let(“x”,Num\ 4,Plus(Ident(“x”),Num\ 3))} \Rightarrow 7}$$

Semantics Defines Program Meaning

- $e \Rightarrow v$ 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 there *exists no* v *for which* $e \Rightarrow v$
- Proofs can be constructed bottom-up
  - In a goal-directed fashion
- Thus, function eval $e = \{v \mid e \Rightarrow v\}$
  - Determinism of semantics implies at most one element for any e
- So: Expression e *means* v

Environment-style Semantics

- So far, semantics used substitution to handle variables
  - As we evaluate, we replace all occurrences of a variable `x` with values 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 we see them

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 ***v*** (intuition: the value of the variable stored on the stack)
    - undefined (intuition: the variable has not been declared)
- An environment can visualized 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

- $\bullet$ is the empty environment
- $A, x{:}v$ is the environment that extends A with a mapping from x to v
  - Sometimes just write $x{:}v$ instead of $\bullet, x{:}v$ for brevity
- Lookup $A(x)$ is defined as follows

$$\bullet(x) \quad = \quad \text{undefined}$$

$$(A, y{:}v)(x) = \begin{cases} v & \text{if } x = y \\ A(x) & \text{if } x <> y \text{ and } A(x) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

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 “undefined”
  | (y,v)::env’ ->
    if x = y then v
    else lookup env’ x
```

An environment is just a list of mappings, which are just pairs of variable to value

- called an association list

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

Environment-style Rules

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

Look up variable x in environment A

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

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

Extend environment A with mapping from x to $v1$

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

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
```

Quiz 2

What is a derivation of the following judgment?

`•; let x=3 in x+2 ⇒ 5`

```
(a)
           x ⇒ 3    2 ⇒ 2   5 is 3+2
           --------------------
3 ⇒ 3        x+2 ⇒ 5
------------------------
let x=3 in x+2 ⇒ 5
```

```
(c)
x:2; x⇒3   x:2; 2⇒2   5 is 3+2
------------------------------
•; let x=3 in x+2 ⇒ 5
```

```
(b)         x:3; x ⇒ 3   x:3; 2 ⇒ 2    5 is 3+2
            ----------------------------------
•;3 ⇒ 3     x:3; x+2 ⇒ 5
--------------------------
•; let x=3 in x+2 ⇒ 5
```

Quiz 2

What is a derivation of the following judgment?

•; let x=3 in x+2 ⇒ 5

```
(a)
             x ⇒ 3    2 ⇒ 2   5 is 3+2
             -------------------
3 ⇒ 3      x+2  ⇒  5
----------------------
let x=3 in x+2 ⇒ 5
```

```
(c)
x:2; x⇒3   x:2; 2⇒2   5 is 3+2 ----
------------------------
•; let x=3 in x+2 ⇒ 5
```

```
(b)          x:3; x ⇒ 3   x:3; 2 ⇒ 2    5 is 3+2
            --------------------------------
•;3 ⇒ 3     x:3; x+2  ⇒  5
------------------------
•; let x=3 in x+2 ⇒ 5
```

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

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

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

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

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

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

Quiz 3

What is the derivation of the following judgment?

`•; if eq0 3-2 then 5 else 10 ⇒ 10`

```
(a)
•; 3 ⇒ 3    •; 2 ⇒ 2   3-2 is 1
-------------------------
•; eq0 3-2  ⇒  false        •; 10 ⇒ 10
-----------------------------------
•; if eq0 3-2 then 5 else 10  ⇒  10
```

```
(b)
3 ⇒ 3   2 ⇒ 2
3-2 is 1
----------------
eq0 3-2 ⇒ false        10 ⇒ 10
------------------------------
if eq0 3-2 then 5 else 10 ⇒ 10
```

```
(c)
•; 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
```

Quiz 3

What is the derivation of the following judgment?

`•; if eq0 3-2 then 5 else 10 ⇒ 10`

(a)

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

(b)

```
3 ⇒ 3   2 ⇒ 2
3-2 is 1
----------------
eq0  3-2 ⇒  false           10 ⇒ 10
-------------------------------
if eq0  3-2  then  5  else  10 ⇒ 10
```

(c)

```
•; 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
```

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
```

Adding Closures to Micro-OCaml

$$\begin{aligned} e ::=\ & x \mid v \mid e + e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\ & \mid \texttt{eq0}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\ & \mid e\ e \mid \texttt{fun}\ x\ \texttt{->}\ e \end{aligned}$$

Environment

$$v ::= n \mid \texttt{true} \mid \texttt{false} \mid (A, \lambda x.e)$$

Code
(id and exp)

- In terms of interpreter definitions:

```
type exp =
   | Val of value
   | If of exp * exp * exp
    … (* as before *)
   | Call of exp * exp
   | Fun of id * exp
```

```
type value =
    Int of int
  | Bool of bool
  | Closure of env * id * exp
```

Rule for Closures: Lexical/Static Scoping

$$\frac{}{A;\ \texttt{fun}\ x\ \texttt{->}\ e\ \Rightarrow\ (A, \lambda x.e)}$$

$$\frac{A;\ e1 \Rightarrow (A', \lambda x.e) \qquad A;\ e2 \Rightarrow v1 \qquad A', x{:}v1;\ e\ \Rightarrow\ v}{A;\ e1\ \ e2\ \Rightarrow\ v}$$

- Notice
  - Creating a closure captures the current environment A
  - A call to a function
    - evaluates the body of the closure’s code ***e*** with function closure’s environment A’ extended with parameter ***x*** bound to argument ***v1***

Rule for Closures: Dynamic Scoping

$$\frac{}{A;\ \texttt{fun}\ x\ \texttt{->}\ e \Rightarrow (\bullet, \lambda x.e)}$$

$$\frac{A;\ e1 \Rightarrow (\bullet, \lambda x.e) \qquad A;\ e2 \Rightarrow v1 \qquad A, x:v1;\ e \Rightarrow v}{A;\ e1\ e2 \Rightarrow v}$$

- Notice
  - Creating a closure ignores the current environment A
  - A call to a function
    - evaluates the body of the closure's code ***e*** with the current environment A extended with parameter ***x*** bound to argument ***v1***

Scaling up

- Operational semantics can handle full languages
  - With records, recursive variant types, objects, first-class functions, and more

- Provides a concise notation for explaining what a language does. Clearly shows:
  - Evaluation order
  - Call-by-value vs. call-by-name
  - Static scoping vs. dynamic scoping
  - ... We may look at more of these later

Scaling up: Lego City