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
- Three main approaches to formal semantics
 - Denotational
 - Operational
 - Axiomatic

Styles of Semantics

- Denotational semantics: translate programs into math!
 - Usually: convert programs into functions mapping inputs to outputs
 - Analogous to compilation
- Operational semantics: define how programs execute
 - Often on an abstract machine (mathematical model of computer)
 - Analogous to interpretation
- 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

This Course: Operational Semantics

- We will show how an operational semantics may be defined for Micro-Ocaml
 - And develop an interpreter for it, along the way
- Approach: 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

Definitional Interpreter

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

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

Micro-OCaml Expression Grammar

$$e := x \mid n \mid e + e \mid let x = e in e$$

- **Le**, **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

$$e := x \mid n \mid e + e \mid let x = e in e$$

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 is describing its abstract syntax tree (AST), i.e., e's structure

```
e := x \mid n \mid e + e \mid \text{let } x = e \text{ in } e
corresponds to (in definite right)
```

Parsing (deferred)

- The parsing problem is how to convert program text into an AST, i.e., a value of the type below
 - We defer worrying about this problem until later
 - > Hint: Relates to using something like regular expressions to read in text and construct values like the following from it

Values

An expression's final result is a value. What can values be?

$$\mathbf{v} := \mathbf{n}$$

- Just numerals for now
 - In terms of an interpreter's representation:
 type value = int
 - 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 \Rightarrow 8
 - let f=1+2 in let z=1 in $f+z \Rightarrow 4$

Rules as English Text

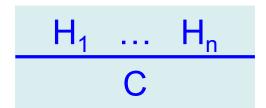
Suppose e is a numeral n

No rule for x

- Then e evaluates to itself, i.e., $n \Rightarrow n$
- Suppose e is an addition expression e1 + e2
 - If e1 evaluates to n1, i.e., e1 \Rightarrow n1
 - If *e2* evaluates to *n2*, i.e., *e2* ⇒ *n2*
 - Then e evaluates to n3, where n3 is the sum of n1 and n2
 - l.e., $e1 + e2 \Rightarrow n3$
- 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., $e2\{v1/x\} \Rightarrow v2$
 - 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 $e2 \Rightarrow v2$

Rules of Inference

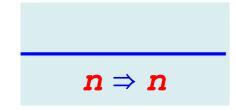
- We can use a more compact notation for the rules we just presented: rules of inference
 - Has the following format



- Says: if the conditions H₁ ... 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 will 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



- Suppose e is an addition expression e1 + e2
 - If e1 evaluates to n1, i.e., e1 ⇒ n1
 - If **e**2 evaluates to **n**2, i.e., **e**2 ⇒ **n**2
 - Then e evaluates to n3, where n3 is the sum of n1 and n2
 - l.e., e1 + e2 ⇒ n3

$$e1 \Rightarrow n1$$
 $e2 \Rightarrow n2$ $n3$ 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., $e2\{v1/x\} \Rightarrow v2$
 - Then e evaluates to v2, i.e., let x = e1 in $e2 \Rightarrow v2$

```
e1 \Rightarrow v1 e2\{v1/x\} \Rightarrow v2
let x = e1 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
 - \triangleright Goal: Show that let x = 4 in $x+3 \Rightarrow 7$

Derivations

$$4 \Rightarrow 4 \qquad 3 \Rightarrow 3 \qquad 7 \text{ is } 4+3$$

$$4 \Rightarrow 4 \qquad 4+3 \Rightarrow 7$$

$$1 \text{ let } \mathbf{x} = 4 \text{ in } \mathbf{x}+3 \Rightarrow 7$$

CMSC 330 Spring 2017

Quiz 1

What is derivation of the following judgment?

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

(a)

$$2 \Rightarrow 2$$
 $3 + 8 \Rightarrow 11$
 $2 + (3 + 8) \Rightarrow 13$

```
(c)

8 \Rightarrow 8

3 \Rightarrow 3

11 \text{ is } 3+8

-----

2 \Rightarrow 2   3 + 8 \Rightarrow 11   13 \text{ is } 2+11

-----

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

Quiz 1

What is derivation of the following judgment?

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

(a)

$$2 \Rightarrow 2$$
 $3 + 8 \Rightarrow 11$
 $2 + (3 + 8) \Rightarrow 13$

```
(b)

3 \Rightarrow 3 \quad 8 \Rightarrow 8

------

3 + 8 \Rightarrow 11 \qquad 2 \Rightarrow 2

------

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

```
(c)

8 \Rightarrow 8

3 \Rightarrow 3

11 \text{ is } 3+8

-----

2 \Rightarrow 2   3 + 8 \Rightarrow 11   13 \text{ is } 2+11

-----

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

Definitional Interpreter

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"
                                                 n \Rightarrow n
    Num n \rightarrow n
  | Plus (e1,e2) ->
                                   e1 \Rightarrow n1 e2 \Rightarrow n2 n3 is n1+n2
      let n1 = eval e1 in
      let n2 = eval e2 in
                                              e1 + e2 \Rightarrow n3
      let n3 = n1+n2 in
      n3
                                       e1 \Rightarrow v1 e2\{v1/x\} \Rightarrow v2
  | Let (x,e1,e2) ->
                                       let x = e1 in e2 \Rightarrow v2
      let v1 = eval e1 in
      let e2' = subst v1 \times e2 in
      let v2 = eval e2' in v2
```

CMSC 330 Spring 2017

Derivations = Interpreter Call Trees

$$4 \Rightarrow 4 \qquad 3 \Rightarrow 3 \qquad 7 \text{ is } 4+3$$

$$4 \Rightarrow 4 \qquad 4+3 \Rightarrow 7$$

$$1 \text{ let } \mathbf{x} = 4 \text{ in } \mathbf{x}+3 \Rightarrow 7$$

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

```
eval Num 4 \Rightarrow 4 eval Num 3 \Rightarrow 3 7 is 4+3

eval (subst 4 "x"

eval Num 4 \Rightarrow 4 Plus(Ident("x"), Num 3)) \Rightarrow 7

eval Let("x", Num 4, Plus(Ident("x"), Num 3)) \Rightarrow 7
```

CMSC 330 Spring 2017

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

Environment-style Semantics

- The previous semantics uses 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 (intuition: the variable has been declared)
 - ... or undefined (intuition: variable has not been declared)
- An environment can also be thought of as a table

ld	Val
x	0
У	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

25

Semantics with Environments

The environment semantics changes the judgment

$$e \Rightarrow v$$

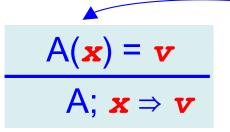
to be

A;
$$e \Rightarrow v$$

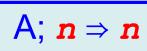
where A is an environment

- 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



Look up variable x in environment A



```
A; e1 \Rightarrow v1 A, x:v1; e2 \Rightarrow v2
A; let x = e1 in e2 \Rightarrow v2
```

Extend environment A with mapping from x to v1

A;
$$e1 \Rightarrow n1$$
 A; $e2 \Rightarrow n2$ $n3$ is $n1+n2$
A; $e1 + e2 \Rightarrow n3$

Quiz 2

What is a derivation of the following judgment?

•; let x=3 in $x+2 \Rightarrow 5$

```
(a)

x \Rightarrow 3  2 \Rightarrow 2  5 \text{ is } 3+2

3 \Rightarrow 3  x+2 \Rightarrow 5

-----

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

•; 3 \Rightarrow 3 \quad x:3; \quad x+2 \Rightarrow 5

•; let x=3 in x+2 \Rightarrow 5
```

CMSC 330 Spring 2017

Quiz 2

What is a derivation of the following judgment?

•; let x=3 in $x+2 \Rightarrow 5$

```
(a)

x \Rightarrow 3  2 \Rightarrow 2  5 \text{ is } 3+2

3 \Rightarrow 3  x+2 \Rightarrow 5

-----

let x=3 in x+2 \Rightarrow 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
```

CMSC 330 Spring 2017

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
```

CMSC 330 Spring 2017 30

Definitional Interpreter: Evaluation

```
let rec eval env e =
 match e with
    Ident x -> lookup env x
   Num n \rightarrow 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
```

CMSC 330 Spring 2017

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:

Rules for Eq0 and Booleans

A; $e \Rightarrow 0$ A; true \Rightarrow true

A; eq0 $e \Rightarrow$ true

A; $e \Rightarrow v \quad v \neq 0$ A; false \Rightarrow false

A; eq0 $e \Rightarrow$ false

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

Rules for Conditionals

A;
$$e1 \Rightarrow \text{true} \quad A$$
; $e2 \Rightarrow v$

A; if $e1$ then $e2$ else $e3 \Rightarrow v$

A; $e1 \Rightarrow \text{false} \quad A$; $e3 \Rightarrow v$

A; if $e1$ then $e2$ else $e3 \Rightarrow v$

- Notice that only one branch is evaluated
 - A; if eq0 0 then 3 else $4 \Rightarrow 3$
 - A; if eq0 1 then 3 else $4 \Rightarrow 4$

Quiz 3

What is the derivation of the following judgment?

•; if eq0 3-2 then 5 else $10 \Rightarrow 10$

```
(a)
•; 3 ⇒ 3 •; 2 ⇒ 2 3-2 is 1

•; eq0 3-2 ⇒ false •; 10 ⇒ 10

•; if eq0 3-2 then 5 else 10 ⇒ 10
```

CMSC 330 Spring 2017

Quiz 3

What is the derivation of the following judgment?

•; if eq0 3-2 then 5 else $10 \Rightarrow 10$

```
(a)
•; 3 ⇒ 3 •; 2 ⇒ 2 3-2 is 1

•; eq0 3-2 ⇒ false •; 10 ⇒ 10

•; if eq0 3-2 then 5 else 10 ⇒ 10
```

```
(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
  \mid Let (x,e1,e2) \rightarrow
     let v1 = eval env e1 in
     let env' = extend env x v1 in
     let v2 = eval env' e2 in v2
                                        Basically both rules for
  | Eq0 e1 ->
                                        eq0 in this one snippet
     let Int n = eval env e1 in
     if n=0 then Bool true else Bool false
  | If (e1,e2,e3) ->
                                        Both if rules here
     let Bool b = eval env e1 in
     if b then eval env e2
     else eval env e3
```

Quick Look: Type Checking

- Inference rules can also be used to specify a program's static semantics
 - I.e., the rules for type checking
- We won't cover this in depth in this course, but here is a flavor.
- Types t ::= bool | int
- Judgment ⊢ e: t says e has type t
 - We define inference rules for this judgment, just as with the operational semantics

Some Type Checking Rules

Boolean constants have type bool

```
⊢ true:bool ⊢ false:bool
```

- Equality checking has type bool too
 - Assuming its target expression has type int

```
⊢e:int
⊢eq0 e:bool
```

Conditionals

```
⊢e1:bool ⊢e2:t ⊢e3:t

⊢if e1 then e2 else e3:t
```

Handling Binding

- What about the types of variables?
 - Taking inspiration from the environment-style operational semantics, what could you do?
- Change judgment to be G ⊢ e: t which says
 e has type t under type environment G
 - G is a map from variables x to types t
 - > Analogous to map A, maps vars to types, not values
- What would be the rules for let, and variables?

Type Checking with Binding

Variable lookup

$$G(x) = t$$

$$G \vdash x : t$$

analogous to

$$A(x) = v$$

$$A; x \Rightarrow v$$

Let binding

$$G \vdash e1: t1$$
 $G,x:t1 \vdash e2: t2$
 $G \vdash let x = e1 in e2: t2$

analogous to

A;
$$e1 \Rightarrow v1$$
 A, $x:v1$; $e2 \Rightarrow v2$
A; let $x = e1$ in $e2 \Rightarrow v2$

Scaling up

- Operational semantics (and similarly styled typing rules) can handle full languages
 - With records, recursive variant types, objects, firstclass 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