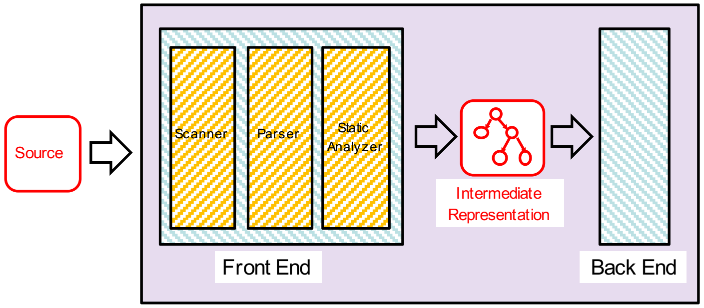
CMSC 330: Organization of Programming Languages

Operational Semantics

Recall Architecture of Compilers, Interpreters



Front end: syntax, (possibly) type checking, other checks
Back end: semantics (i.e. execution)

Specifying Syntax, Semantics

- We have seen how the syntax of a programming language may be specified precisely
  - Regular expressions
  - Context-free grammars
- What about formal methods for defining the semantics of a programming language?
  - I.e., what does a program mean / do?

Formal Semantics of a Prog. Lang.

- Mathematical description of all possible computations performed by programs written in that language
- Three main approaches to formal semantics
  - Denotational
  - Operational
  - Axiomatic

Formal Semantics (cont.)

- 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 using a subset of OCaml
- Approach: use rules to define a relation

  E ⇒ v

  - E: expression in OCaml subset
  - v: value that results from evaluating E
- To begin with, need formal definitions of:
  - Set Exp of expressions
  - Set Val of values

Defining Exp

- Recall: operational semantics defines what happens in backend
  - Front end parses code into abstract syntax trees (ASTs)
  - So inputs to backend are ASTs
- How to define ASTs?
  - Standard approach
    - Using grammars!
  - Idea
    - Grammar defines abstract syntax (no parentheses, grouping constructs, etc.; grouping is implicit)

OCaml Abstract Syntax

E ::= x | n | `true` | `false` | `[]`
  | E *op* E (*op* ∈ {+,-,/,*,=,<,>,::, etc.})
  | *l_op* E (*l_op* ∈ {hd, tl})
  | `if` E `then` E `else` E
  | `fun` x → E | E E | `let` x = E `in` E

- x may be any identifier
- n is any numeral (digit sequence, with optional -).
- `true` and `false` stand for the two boolean constants
- `[]` is the empty list

Exp = set of (type-correct) ASTs defined by grammar

- Note that the grammar is ambiguous
  - OK because not using grammar for parsing
  - But for defining the set of all syntactically legal terms

Values

- What can results be?
  - Integers
  - Booleans
  - Lists
  - Functions
- We will deal with first three initially

Formal Definition of Val

- Let
  - $\mathbb{Z}$ = {…, -1, 0, -1, …} be the (math) set of integers
  - $\mathbb{B}$ = {ff, tt} be the (math) set of booleans
  - nil be a distinguished value (empty list)
- Then Val is the smallest set such that
  - $\mathbb{Z}, \mathbb{B} \subseteq$ Val and nil $\in$ Val
  - If $v_1, v_2 \in$ Val then $\langle v_1, v_2 \rangle \in$ Val
- "Smallest set"?
  - Every integer and boolean is a value, as is nil
  - Any pair of values is also a value

Operations on Val

- Basic operations will be assumed
  - +, -, *, /, =, <, ≤, etc.
- Not all operations are applicable to all values!
  - tt + ff is undefined
  - So is 1 + nil
- A key purpose of type checking is to prevent these undefined operations from occurring during execution

Implementing Exp, Val in OCaml

E ::= x | n | true | false | [] | if E then E else E
| fun x = E | E E | let x = E in E …

```
type ast =
    Id of string
  | Num of int
  | Bool of bool
  | Nil
  | If of ast * ast * ast
  | Fun of string * ast
  | App of ast * ast
  | Let of string * ast * ast
  | …
```

Val

```
type value =
    Val_Num of int
  | Val_Bool of bool
  | Val_Nil
  | Val_Pair of value *
               value
  | …
```

Defining Evaluation (⇒)

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

- Approach is inductive and uses rules:
  - Idea: if the conditions H_1 … H_n ("hypotheses") are true, the rule says the condition C ("conclusion") below the line follows
  - Hypotheses, conclusion are statements about ⇒; hypotheses involve subexpressions of conclusions
  - If n=0 (no hypotheses) then the conclusion is automatically true and is called an axiom
    - A "-" may be written in place of the hypothesis list in this case
    - Terminology: statements one is trying to prove are called judgments
- This method is often called "Structural Operational Semantics (SOS)" or "Natural Semantics"

SOS Rules: Basic Values

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

$$\frac{-}{\texttt{false} \Rightarrow \text{ff}} \qquad \frac{-}{\texttt{true} \Rightarrow \text{tt}}$$

$$\frac{-}{\texttt{[]} \Rightarrow \text{nil}}$$

- Each basic entity evaluates to its corresponding value
- Note: axioms!

SOS Rules: Built-in Functions

- How about built-in functions (+, -, etc.)?
  - In OCaml, type-checking done in front end
  - Thus, ASTs coming to back end are type-correct
  - So we assume Exp contains type-correct ASTs
- We will use relevant operations on value side

SOS Rules: Built-in Functions

- For arithmetic, comparison operations, etc.

$$\frac{E_1 \Rightarrow v_1 \qquad E_2 \Rightarrow v_2}{E_1 \text{ op } E_2 \Rightarrow v_1 \text{ op } v_2}$$

- For : :

$$\frac{E_1 \Rightarrow v_1 \qquad E_2 \Rightarrow v_2}{E_1 :: E_2 \Rightarrow \langle v_1, v_2 \rangle}$$

- Rules are recursive
- :: is implemented using pairing
  - Type system guarantees result is list

Trees of Semantic Rules

- When we apply rules to an expression, we actually get a tree
  - Corresponds to the recursive evaluation procedure
    - For example: (3 + 4) + 5

$$\dfrac{\dfrac{3 \Rightarrow 3 \qquad 4 \Rightarrow 4}{(3+4) \Rightarrow 7} \qquad 5 \Rightarrow 5}{(3+4)+5 \Rightarrow 12}$$

Rules for `hd`, `tl`

$$\dfrac{E \Rightarrow \langle v_1, v_2 \rangle}{\texttt{hd}\ E \Rightarrow v_1} \qquad \dfrac{E \Rightarrow \langle v_1, v_2 \rangle}{\texttt{tl}\ E \Rightarrow v_2}$$

- Note that the rules only apply if E evaluates to a pair of values
- Nothing in this rule requires the pair to correspond to a list
- The OCaml type system ensures this

Error Cases

$$\dfrac{E_1 \Rightarrow v_1 \qquad E_2 \Rightarrow v_2}{E_1 + E_2 \Rightarrow v_1 + v_2}$$

- What if v_1, v_2 aren't integers?
  - E.g., what if we write `false + true`?
  - It can be parsed in OCaml, but type checker will disallow it from being passed to back end
- In a language with dynamic strong typing (e.g. Ruby), rules include explicit type checks

$$\dfrac{E_1 \Rightarrow v_1 \quad v_1 \in \mathbb{Z} \quad E_2 \Rightarrow v_2 \quad v_2 \in \mathbb{Z}}{E_1 + E_2 \Rightarrow v_1 + v_2}$$

- Convention: if no rules are applicable to an expression, its result is an error

Rules for If

$$\dfrac{E_1 \Rightarrow tt \qquad E_2 \Rightarrow v_2}{\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v_2}$$

$$\dfrac{E_1 \Rightarrow ff \qquad E_3 \Rightarrow v_3}{\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v_3}$$

- Notice that only one branch is evaluated
- E.g.
  - `if true then 3 else 4` ⇒ 3
  - `if false then 3 else 4` ⇒ 4

Using Rules to Define Evaluation

- $E \Rightarrow v$ holds if and only if a proof can be built
  - Proofs start with axioms, involve applications of rules whose hypotheses have been proved
  - No proof means $E \not\Rightarrow v$
- Proofs can be constructed in a goal-directed fashion
- Thus, function eval (E) = $\{v \mid E \Rightarrow v\}$
  - Determinsm of semantics implies at most one element for any E

Rules for Identifiers

- The previous rules handle expressions that involve constants (e.g. `1`, `true`) and operations
- What about variables?
  - These are allowed as expressions
  - How do we evaluate them?
- In a program, variables must be declared
  - The values that are part of the declaration are used to evaluate later occurrences of the variables
  - We will use environments to "hold" these declarations in our semantics

Environments

- Mathematically, an environment is a partial function from identifiers to values
  - If A is an environment, and id is an identifier, then A(id) 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	ff

  - then A(x) is 0, A(y) is ff, 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

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

- Idea: A' "overwrites" definitions in A
- For brevity, can write •, A as just A

Semantics with Environments

- To give a semantics for identifiers, we will extend judgments from

  $$E \Rightarrow v$$

  to

  $$A; E \Rightarrow v$$

  where A is an environment
  - Idea: A is used to give values to the identifiers in E
  - A can be thought of as containing all the declarations made up to E
- Existing rules can be modified by inserting A everywhere in the judgments

Existing Rules Have To Be Modified

- E.g.

$$\frac{E_1 \Rightarrow v_1 \qquad E_2 \Rightarrow v_2}{E_1 + E_2 \Rightarrow v_1 + v_2}$$

- becomes

$$\frac{A;\ E_1 \Rightarrow v_1 \qquad A;\ E_2 \Rightarrow v_2}{A;\ E_1 + E_2 \Rightarrow v_1 + v_2}$$

- These modifications can be done systematically

Rule for Identifiers

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

- x is an identifier
- To determine its value v "look it up" in A!

Rule for Let Binding

- We evaluate the the first expression, and then evaluate the second expression in an environment extended to include a binding for x

$$\frac{\begin{array}{c} A;\ E_1 \Rightarrow v_1 \\ A,\ x{:}v_1;\ E_2 \Rightarrow v_2 \end{array}}{A;\ \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v_2}$$

Function Values

- So far our semantics handles
  - Constants
  - Built-in operations
  - Identifiers
- What about function definitions?
  - Recall form: `fun` x → E
  - To evaluate these expressions we need to add closures to our set of values

Closures

- … are what OCaml function expressions evaluate to
- A closure consists of
  - Parameter (id)
  - Body (expression)
  - Environment (used to evaluate free variables in body)
- Formal extension to Val
  - if x is an id, E is an expression, and A is an environment
  - … then (A, λx.E) ∈ Val

Rule for Function Definitions

$$\frac{\text{–}}{A;\ \texttt{fun}\ x \rightarrow E \Rightarrow (A, \lambda x.E)}$$

- The expression evaluates to a closure
  - The id and body in the closure come from the expression
  - The environment is the one in effect when the expression is evaluated
- This will be used to implement static scope

Evaluating Function Application

- How do we evaluate a function application expression of the form E_1 E_2?
  - Static scope
  - Call by value
- Strategy
  - Evaluate E_1, producing v_1
  - If v_1 is indeed a function (i.e. closure) then
    - Evaluate E_2, producing v_2
    - Set the parameter of closure v_1 equal to v_2
    - Evaluate the body under this binding of the parameter
    - Remember that non-parameter ids in the body must be interpreted using the closure!

Rule for Function Application

$$\frac{A;\ E_1 \Rightarrow (A',\ \lambda x.E) \qquad A;\ E_2 \Rightarrow v_2 \qquad A',\ x{:}v_2;\ E \Rightarrow v}{A;\ E_1\ E_2 \Rightarrow v}$$

- 1st hypothesis: E_1 evaluates to a closure
- 2nd hypothesis: E_2 produces a value (call by value!)
- 3rd hypothesis: Body E in modified closure environment produces a value
- This last value is the result of the application

Example: (fun x → x + 3) 4

$$\frac{\bullet;\ \texttt{fun}\ x \to x+3 \Rightarrow (\bullet,\ \lambda x.x+3) \qquad \bullet;\ 4 \Rightarrow 4 \qquad \dfrac{\bullet,\ x{:}4;\ x \Rightarrow 4 \qquad \bullet,\ x{:}4;\ 3 \Rightarrow 3}{\bullet,\ x{:}4;\ x+3 \Rightarrow 7}}{\bullet;\ (\texttt{fun}\ x \to x+3)\ \ 4 \Rightarrow 7}$$

Example: (fun x → (fun y → x + y)) 3 4

$$\frac{\bullet;\ (\text{fun } x \to (\text{fun } y \to x+y)) \Rightarrow (\bullet,\ \lambda x.(\text{fun } y \to x+y)) \qquad \bullet;\ 3 \Rightarrow 3 \qquad x{:}3;\ (\text{fun } y \to x+y) \Rightarrow (x{:}3,\ \lambda y.(x+y))}{\bullet;\ (\text{fun } x \to (\text{fun } y \to x+y))\ \ 3 \Rightarrow (x{:}3,\ \lambda y.(x+y))}$$

Let <previous> = (fun x → (fun y → x + y)) 3

Example (cont.)

$$\frac{\bullet;\ \text{<previous>} \Rightarrow (x{:}3,\ \lambda y.(x+y)) \qquad \bullet;\ 4 \Rightarrow 4 \qquad \dfrac{\bullet, x{:}3,\ y{:}4;\ x \Rightarrow 3 \qquad \bullet, x{:}3,\ y{:}4;\ y \Rightarrow 4}{x{:}3,\ y{:}4;\ (x+y) \Rightarrow 7}}{\bullet;\ (\ \text{<previous>}\ 4\ ) \Rightarrow 7}$$

Dynamic Scoping

- The previous rule for functions implements static scoping, since it implements closures
- We could easily implement dynamic scoping

$$\frac{A;\ E_1 \Rightarrow (A',\ \lambda x.E) \quad A;\ E_2 \Rightarrow v_2 \quad \mathbf{A},\ x{:}v_2;\ E \Rightarrow v}{A;\ E_1\ E_2 \Rightarrow v}$$

- In short: use the current environment A, not A'
  - Easy to see the origins of the dynamic scoping bug!
- Question: How might you use both?

Practice Examples

- Give a derivation that proves the following (where • is the empty environment)
  - •; let x = 5 in let y = 7 in x+y ⇒ 12
  - •; let x = let x = 5 in x+2 in x+2 ⇒ 9
  - •; let f = fun x → x+5 in f 7 ⇒ 12
  - •; let y = 5 in let f = fun x → x+y in let y = 6 in f 7 ⇒ 12
- Using the dynamic scoping version of the function application rule, prove
  - •; let y = 5 in let f = fun x → x+y in let y = 6 in f 7 ⇒ 13