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
  \[ E \Rightarrow v \]
  - \( E \): expression in OCaml subset
  - \( v \): value that results from evaluating \( E \)
- To begin with, need formal definitions of:
  - Set \( \text{Exp} \) of expressions
  - Set \( \text{Val} \) 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

\[ E ::= x \mid n \mid \text{true} \mid \text{false} \mid [] \]
\[ \mid E \; op \; E \ (op \in \{+,-,/,\ast,=,<,>,::,\text{etc.}\}) \]
\[ \mid \_op \; E \ (\_op \in \{\text{hd, tl}\}) \]
\[ \mid \text{if } E \text{ then } E \text{ else } E \]
\[ \mid \text{fun } x \rightarrow E \mid E \; E \mid \text{let } x = E \text{ in } E \]
- \( x \) may be any identifier
- \( n \) is any numeral (digit sequence, with optional -).
- \( \text{true} \) and \( \text{false} \) stand for the two boolean constants
- \( [] \) is the empty list

\( \text{Exp} = \) 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
- \( \mathbb{B} = \{ \text{ff}, \text{tt} \} \) be the (math) set of booleans
- nil be a distinguished value (empty list)

Then Val is the smallest set such that

- \( \mathbb{Z}, \mathbb{B} \subseteq \text{Val} \) and nil \( \in \text{Val} \)
- If \( \nu_1, \nu_2 \in \text{Val} \) then \( \langle \nu_1, \nu_2 \rangle \in \text{Val} \)

“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

\[
E ::= x \mid n \mid true \mid false \mid [] \mid \text{if } E \text{ then } E \text{ else } E \\
| \text{fun } x = E \mid E \ E \mid \text{let } x = E \text{ in } E \ldots
\]

```ocaml
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
  | ...
```

CMSC 330
Defining Evaluation (⇒)

- Approach is inductive and uses rules:
  - Idea: if the conditions \( H_1 \ldots H_n \) ("hypotheses") are true, the rule says the condition \( C \) ("conclusion") below the line follows
  - Hypotheses, conclusion are statements about \( \Rightarrow \); 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

<table>
<thead>
<tr>
<th>( )</th>
<th>( )</th>
</tr>
</thead>
</table>
| \( n \Rightarrow n \) | \(
| \( \) | \( n \Rightarrow n \) |
| false \( \Rightarrow \) ff | \( \) | \( true \Rightarrow \) tt |
| \( \) | \( \) |
| \( \) | \( \) |
| \( [] \Rightarrow \) 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

For arithmetic, comparison operations, etc.

\[
\begin{align*}
E_1 &\Rightarrow v_1 & E_2 &\Rightarrow v_2 \\
E_1 \text{ op } E_2 &\Rightarrow v_1 \text{ op } v_2
\end{align*}
\]

For ::

\[
\begin{align*}
E_1 &\Rightarrow v_1 & E_2 &\Rightarrow v_2 \\
E_1 :: E_2 &\Rightarrow \langle v_1, v_2 \rangle
\end{align*}
\]

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: 
      \[(3 + 4) + 5\]

      \[
      \begin{align*}
      3 & \Rightarrow 3 \\
      4 & \Rightarrow 4 \\
      \end{align*}
      \]

      \[
      \begin{align*}
      (3 + 4) & \Rightarrow 7 \\
      5 & \Rightarrow 5 \\
      \end{align*}
      \]

      \[
      (3 + 4) + 5 \Rightarrow 12
      \]

Rules for \(\text{hd}, \text{tl}\)

- 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

- What if \( v_1, v_2 \) aren’t integers?
  - E.g., what if we write \( \text{false} + \text{true} \)?
  - 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

<table>
<thead>
<tr>
<th>( E_1 \Rightarrow v_1 )</th>
<th>( E_2 \Rightarrow v_2 )</th>
<th>( E_1 + E_2 \Rightarrow v_1 + v_2 )</th>
</tr>
</thead>
</table>

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

Rules for If

- Notice that only one branch is evaluated
  - E.g.
    - \( \text{if true then 3 else 4} \Rightarrow 3 \)
    - \( \text{if false then 3 else 4} \Rightarrow 4 \)
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 $\text{eval}(E) = \{ v \mid E \Rightarrow v \}$
  - Determinism of semantics implies at most one element for any $E$

Rules for Identifiers

- The previous rules handle expressions that involve constants (e.g. $1$, $\text{true}$) 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 \( \text{id} \) is an identifier, then \( A(\text{id}) \) 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   |
    | \( y \) | \( \text{ff} \) |

  - then \( A(x) \) is 0, \( A(y) \) is \( \text{ff} \), and \( A(z) \) is undefined

Notation, Operations on Environments

- \( \varepsilon \) is the empty environment (undefined for all ids)
- \( \text{x:v} \) 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

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

- Idea: \( A' \) “overwrites” definitions in \( A \)
- For brevity, can write \( \bullet, A \) 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.

<table>
<thead>
<tr>
<th>( E_1 \Rightarrow v_1 )</th>
<th>( E_2 \Rightarrow v_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 + E_2 \Rightarrow v_1 + v_2 )</td>
<td></td>
</tr>
</tbody>
</table>

- becomes

<table>
<thead>
<tr>
<th>( A; E_1 \Rightarrow v_1 )</th>
<th>( A; E_2 \Rightarrow v_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A; E_1 + E_2 \Rightarrow v_1 + v_2 )</td>
<td></td>
</tr>
</tbody>
</table>

- These modifications can be done systematically
Rule for Identifiers

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

\[
\begin{align*}
A(x) &= v \\
A; x &\Rightarrow v
\end{align*}
\]

Rule for Let Binding

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

\[
\begin{align*}
A; E_1 &\Rightarrow v_1 \\
A, x: v_1; E_2 &\Rightarrow v_2 \\
A; \text{let } x = E_1 \text{ in } E_2 &\Rightarrow v_2
\end{align*}
\]
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 \((A, \lambda x. E) \in Val\)
Rule for Function Definitions

\[
\begin{array}{|c|c|}
\hline
- & \hline
\end{array}
\]

\[A; \text{fun } x \rightarrow E \Rightarrow (A, \lambda x. E)\]

- 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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A; E₁ ⇒ (A'; λx.E)</td>
<td>1\textsuperscript{st} hypothesis: E₁ evaluates to a closure</td>
</tr>
<tr>
<td>A; E₂ ⇒ v₂</td>
<td>2\textsuperscript{nd} hypothesis: E₂ produces a value (call by value!)</td>
</tr>
<tr>
<td>A', x:v₂; E ⇒ v</td>
<td>3\textsuperscript{rd} hypothesis: Body E in modified closure environment produces a value</td>
</tr>
<tr>
<td>A; E₁ E₂ ⇒ v</td>
<td>This last value is the result of the application</td>
</tr>
</tbody>
</table>

Example: (fun x → x + 3) 4

\[
\begin{align*}
\ast, \ x:4; \ x & \Rightarrow 4 \\
\ast, \ x:4; \ 3 & \Rightarrow 3 \\
\ast; \ \text{fun} \ x \rightarrow x + 3 & \Rightarrow (\ast, \ \lambda x. x + 3) \\
\ast; \ 4 & \Rightarrow 4 \\
\ast, \ x:4; \ x + 3 & \Rightarrow 7 \\
\ast; \ (\text{fun} \ x \rightarrow x + 3) \ 4 & \Rightarrow 7
\end{align*}
\]
Example: \((\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y))\) 3 4

\[
\bullet; (\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) \Rightarrow (\bullet, \lambda x.(\text{fun } y \rightarrow x + y))
\]
\[
\bullet; 3 \Rightarrow 3
\]
\[
x:3; (\text{fun } y \rightarrow x + y) \Rightarrow (x:3, \lambda y.(x + y))
\]
\[
\bullet; (\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) 3 \Rightarrow (x:3, \lambda y.(x + y))
\]

Let \(<\text{previous}> = (\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) 3

Example (cont.)

\[
\bullet, x:3, y:4; x \Rightarrow 3 \quad \bullet, x:3, y:4; y \Rightarrow 4
\]
\[
\bullet; <\text{previous}> \Rightarrow (x:3, \lambda y.(x + y))
\]
\[
\bullet; 4 \Rightarrow 4
\]
\[
x:3, y:4; (x + y) \Rightarrow 7
\]
\[
\bullet; ( <\text{previous}> 4 ) \Rightarrow 7
\]
Dynamic Scoping

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

\[
egin{array}{c}
A; E_1 \Rightarrow (A', \lambda x. E) \\
A; E_2 \Rightarrow v_2 \\
A, x : v_2; E \Rightarrow v \\
A; E_1 E_2 \Rightarrow v
\end{array}
\]

- The only difference is to use the current environment \( A \), not the environment \( A' \).
  - Easy to see the origins of the dynamic scoping bug!

Practice Examples

- Give a derivation that proves the following (where \( \bullet \) is the empty environment)
  1. \( \bullet; \text{let } x = 5 \text{ in let } y = 7 \text{ in } x+y \Rightarrow 12 \)
  2. \( \bullet; \text{let } x = \text{let } x = 5 \text{ in } x+2 \text{ in } x+2 \Rightarrow 9 \)
  3. \( \bullet; \text{let } f = \text{fun } x \rightarrow x+5 \text{ in } f \ 7 \Rightarrow 12 \)
  4. \( \bullet; \text{let } y = 5 \text{ in } \text{let } f = \text{fun } x \rightarrow x+y \text{ in } \text{let } y = 6 \text{ in } f \ 7 \Rightarrow 12 \)
- Using the dynamic scoping version of the function application rule, prove
  1. \( \bullet; \text{let } y = 5 \text{ in } \text{let } f = \text{fun } x \rightarrow x+y \text{ in } \text{let } y = 6 \text{ in } f \ 7 \Rightarrow 13 \)