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

\[
\begin{align*}
E & ::= x | n | \text{true} | \text{false} | [] \\
   & \quad | E \text{ op } E \quad (\text{op } \in \{+, -, *, =, =, <, >, ::, \text{ etc.}\}) \\
   & \quad | l_{\text{op}} E \quad (l_{\text{op}} \in \{\text{hd}, \text{tl}\}) \\
   & \quad | \text{if } E \text{ then } E \text{ else } E \\
   & \quad | \text{fun } x \rightarrow E \mid E \mid \text{let } x = E \text{ in } E
\end{align*}
\]
- \( x \) 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
  • \( Z = \{\ldots, -1, 0, -1, \ldots\} \) be the (math) set of integers
  • \( B = \{\text{ff}, \text{tt}\} \) be the (math) set of booleans
  • \( \text{nil} \) be a distinguished value (empty list)
• Then Val is the smallest set such that
  • \( Z, B \subseteq \text{Val} \) and \( \text{nil} \in \text{Val} \)
  • If \( v_1, v_2 \in \text{Val} \) then \( \langle v_1, v_2 \rangle \in \text{Val} \)
• “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
  • \( +, -, \ast, /, =, <, \leq, \) etc.
• Not all operations are applicable to all values!
  • \( \text{tt} + \text{ff} \) is undefined
  • So is \( 1 + \text{nil} \)
• A key purpose of type checking is to prevent
  these undefined operations from occurring
during execution

Implementing Exp, Val in OCaml

\[
\begin{align*}
E ::= & x \mid n \mid \text{true} \mid \text{false} \mid [] \mid \text{if } E \text{ then } E \text{ else } E \\
& \text{fun } x = E \mid E \ E \mid \text{let } x = E \text{ in } E 
\end{align*}
\]

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

```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
| ...
```

```ocaml
type value =
  Val_Num of int
| Val_Bool of bool
| Val_Nil
| Val_Pair of value * value
| ...
```
### Defining Evaluation (⇒)

| Hypotheses (H₁ ... Hₙ) | Conclusion (C) |

- Approach is inductive and uses rules:
  - Idea: if the conditions $H₁ ... Hₙ$ ("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

<table>
<thead>
<tr>
<th>Value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-$</td>
<td>$n \Rightarrow n$</td>
</tr>
<tr>
<td>$false$</td>
<td>$false \Rightarrow ff$</td>
</tr>
<tr>
<td>$true$</td>
<td>$true \Rightarrow tt$</td>
</tr>
<tr>
<td>$[]$</td>
<td>$[] \Rightarrow nil$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E₁ \Rightarrow v₁$</td>
<td>$E₂ \Rightarrow v₂$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E₁ \ op \ E₂ \Rightarrow v₁ \ op \ v₂$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E₁ \Rightarrow v₁$</td>
<td>$E₂ \Rightarrow v₂$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E₁ :: E₂ \Rightarrow \langle v₁, v₂ \rangle$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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: \((3 + 4) + 5\)

\[
\begin{align*}
3 & \Rightarrow 3 \\
4 & \Rightarrow 4 \\
3 + 4 & \Rightarrow 7 \\
5 & \Rightarrow 5 \\
(3 + 4) + 5 & \Rightarrow 12
\end{align*}
\]

Rules for \(hd, tl\)

- 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 \(false + true\)?
  - 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

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

- 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.
  - if \(true\) then \(3\) else \(4\) \(\Rightarrow 3\)
  - if \(false\) then \(3\) else \(4\) \(\Rightarrow 4\)
Using Rules to Define Evaluation

- **E ⇒ 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 ⊨ v**
- Proofs can be constructed in a goal-directed fashion
- Thus, function **eval (E)** = {v | E ⇒ 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, 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

<table>
<thead>
<tr>
<th>Id</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>ff</td>
</tr>
</tbody>
</table>
  - 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) =
    - A(id) if A'(id) defined
    - A(id) if A'(id) undefined but A(id) defined
    - undefined otherwise
- 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

Rule for Identifiers

- \( A(x) = v \)
- \( A; x \Rightarrow v \)
  - \( x \) is an identifier
  - To determine its value \( v \) “look it up” in \( A \! \)

Existing Rules Have To Be Modified

- E.g.
  \[
  \begin{align*}
  E_1 \Rightarrow v_1 & \quad & E_2 \Rightarrow v_2 \\
  \end{align*}
  \]

  \[
  \begin{align*}
  E_1 + E_2 \Rightarrow v_1 + v_2 \\
  \end{align*}
  \]

  \[
  \begin{align*}
  A; E_1 \Rightarrow v_1 & \quad & A; E_2 \Rightarrow v_2 \\
  \end{align*}
  \]

- \[
  A; E_1 + E_2 \Rightarrow v_1 + v_2 \\
  \]

  These modifications can be done systematically

Rule for Let Binding

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

  \[
  \begin{align*}
  A; E_1 \Rightarrow v_1 & \quad & A, x:v_1; E_2 \Rightarrow v_2 \\
  \end{align*}
  \]

  \[
  A; \text{let } x = E_1 \text{ in } E_2 \Rightarrow v_2 \\
  \]
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 \((A, \lambda x.E) \in \text{Val}\)

Rule for Function Definitions

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

- 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>Hypothesis</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1$ evaluates to a closure</td>
<td>$A; E_1 \Rightarrow (A', \lambda x.E)$</td>
</tr>
<tr>
<td>$E_2$ produces a value (call by value!)</td>
<td>$A; E_2 \Rightarrow v_2$</td>
</tr>
<tr>
<td>Body $E$ in modified closure environment produces a value</td>
<td>$A', x:v_2; E \Rightarrow v$</td>
</tr>
<tr>
<td>This last value is the result of the application</td>
<td>$A; E_1, E_2 \Rightarrow v$</td>
</tr>
</tbody>
</table>

1. First hypothesis: $E_1$ evaluates to a closure
2. Second hypothesis: $E_2$ produces a value (call by value!)
3. Third hypothesis: Body $E$ in modified closure environment produces a value

Example: $(\mathbf{fun} \ x \rightarrow \ x + 3) \ 4$

1. $\cdot; (x:4; x \Rightarrow 4) \Rightarrow (\cdot; (\lambda x.x + 3))$
2. $\cdot; 4 \Rightarrow 4$
3. $\cdot; x:4; x + 3 \Rightarrow 7$

Let $\langle\text{previous}\rangle = (\mathbf{fun} \ x \rightarrow (\mathbf{fun} \ y \rightarrow x + y)) \ 3$

Example (cont.)

1. $\cdot; x:3, y:4; x \Rightarrow 3 \Rightarrow (\cdot; x:3, y:4; y \Rightarrow 4)$
2. $\cdot; \langle\text{previous}\rangle \Rightarrow (x:3, \lambda y.(x + y))$
3. $\cdot; 4 \Rightarrow 4$
4. $x:3, y:4; (x + y) \Rightarrow 7$
5. $\cdot; (\langle\text{previous}\rangle\ 4 \ ) \Rightarrow 7$
Dynamic Scoping

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

\[
\begin{align*}
A; E_1 & \Rightarrow (A', \lambda x. E) \\
A; E_2 & \Rightarrow v_2 \\
A, x : v_2; E & \Rightarrow v
\end{align*}
\]

- 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 \( \bullet \) is the empty environment)
  - \( \bullet; \text{let } x = 5 \text{ in } \text{let } y = 7 \text{ in } x+y \Rightarrow 12 \)
  - \( \bullet; \text{let } x = \text{let } x = 5 \text{ in } x+2 \text{ in } x+2 \Rightarrow 9 \)
  - \( \bullet; \text{let } f = \text{fun } x \rightarrow x+5 \text{ in } f 7 \Rightarrow 12 \)
  - \( \bullet; \text{let } y = 5 \text{ in } f = \text{fun } x \rightarrow x+y \text{ in } \text{let } y = 6 \text{ in } f 7 \Rightarrow 12 \)
- Using the dynamic scoping version of the function application rule, prove
  - \( \bullet; \text{let } y = 5 \text{ in } f = \text{fun } x \rightarrow x+y \text{ in } \text{let } y = 6 \text{ in } f 7 \Rightarrow 13 \)