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
  - Operational ⇐ this course
    - Often on an abstract machine (mathematical model of computer)
    - Analogous to interpretation
  - Denotational
  - Axiomatic
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

- Rules for judgment $e \Rightarrow v$ can be easily turned into idiomatic OCaml code for an interpreter
  - The language’s expressions $e$ and values $v$ have corresponding OCaml datattype representations $\text{exp}$ and $\text{value}$
  - The semantics is represented as a function
    
    $$\text{eval}: \text{exp} \rightarrow \text{value}$$

- This way of presenting the semantics is referred to as a definitional interpreter
  - The interpreter defines the language’s meaning
Abstract Syntax Tree spec. via "Grammar"

- We use a grammar for \( e \) to \textbf{directly} describe an expression’s abstract syntax tree (AST), i.e., \( e \)'s structure

\[
e ::= x | n | e + e | \text{let } x = e \text{ in } e
\]

corresponds to (in definitional interpreter)

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

We are \textit{not} concerned about the process of \textbf{parsing}, i.e., from text to an AST. We can thus ignore issues of ambiguity, etc. and focus on the \textbf{structure} of the AST given by the grammar.
Micro-OCaml Expression Grammar

\[ e ::= x | n | e + e | \text{let } x = e \text{ in } e \]

- \(e, x, n\) are **meta-variables** that stand for categories of syntax (like non-terminals in a CFG)
  - \(x\) is any identifier (like \(z, y, \text{foo}\))
  - \(n\) is any numeral (like \(1, 0, 10, -25\))
  - \(e\) is any expression (here defined, recursively!)

**Concrete syntax** of actual expressions in **black**

- Such as \(\text{let}, +, z, \text{foo}, \text{in}, \ldots\) (like terminals in a CFG)

\(::=\) and \(|\) are **meta-syntax** used to define the syntax of a language

(part of “Backus-Naur form,” or BNF)
Micro-OCaml Expression Grammar

\[ e ::= x \mid n \mid e + e \mid \text{let } x = e \text{ in } e \]

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 \( e \)

• \( \text{let } z = 1 \text{ in } 1 + z \) is an expression \( e \) because
  ➢ \( z \) is an identifier \( x \),
  ➢ \( 1 \) is an expression \( e \),
  ➢ \( 1 + z \) is an expression \( e \), and
  ➢ \( \text{let } x = e \text{ in } e \) is an expression \( e \)
Values

- A value \( v \) is an expression’s final result

\[ v ::= n \]

- Just numerals for now
  - In terms of an interpreter’s representation:
    \[
    \text{type value} = \text{int}
    \]
  - In a full language, values \( v \) will also include booleans (\text{true}, \text{false}), strings, functions, …
Defining the Semantics

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

- Judgments are just statements. We use rules to prove that the statement is true.
  - $1+3 \Rightarrow 4$
    - $1+3$ 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 $n$
  - Then $e$ evaluates to itself, i.e., $n \Rightarrow n$

- Suppose $e$ is an addition expression $e_1 + e_2$
  - If $e_1$ evaluates to $n_1$, i.e., $e_1 \Rightarrow n_1$
  - And if $e_2$ evaluates to $n_2$, i.e., $e_2 \Rightarrow n_2$
  - Then $e$ evaluates to $n_3$, where $n_3$ is the sum of $n_1$ and $n_2$
  - I.e., $e_1 + e_2 \Rightarrow n_3$

- Suppose $e$ is a let expression $\text{let } x = e_1 \text{ in } e_2$
  - If $e_1$ evaluates to $v_1$, i.e., $e_1 \Rightarrow v_1$
  - And if $e_2\{v_1/x\}$ evaluates to $v_2$, i.e., $e_2\{v_1/x\} \Rightarrow v_2$
    - Here, $e_2\{v_1/x\}$ means “the expression after substituting occurrences of $x$ in $e_2$ with $v_1$”
  - Then $e$ evaluates to $v_2$, i.e., $\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2$
Rules are Lego Blocks

P = 8.0 mm
= 5/6 × H
= 2.5 × h

h = 3.2 mm
= 1/3 × H
= 0.4 × P

2 × P = 0.2 mm
= 15.8 mm

H = 9.6 mm
= 3 × h
= 1.2 × P

P = 0.2 mm
= 7.8 mm
We can use a more compact notation for the rules we just presented: **rules of inference**

- Has the following format:

  ![Inference Rule Diagram](image)

- Says: if the conditions $H_1 \ldots H_n$ ("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 are using inference rules where $C$ is our judgment about evaluation, i.e., that $\mathbf{e} \Rightarrow \mathbf{v}$.
Rules of Inference: Num and Sum

- Suppose $e$ is a numeral $n$
  - Then $e$ evaluates to itself, i.e., $n \Rightarrow n$

- Suppose $e$ is an addition expression $e_1 + e_2$
  - If $e_1$ evaluates to $n_1$, i.e., $e_1 \Rightarrow n_1$
  - If $e_2$ evaluates to $n_2$, i.e., $e_2 \Rightarrow n_2$
  - Then $e$ evaluates to $n_3$, where $n_3$ is the sum of $n_1$ and $n_2$, i.e., $e_1 + e_2 \Rightarrow n_3$

$e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad n_3$ is $n_1 + n_2$

$e_1 + e_2 \Rightarrow n_3$
Rules of Inference: Let

- Suppose $e$ is a let expression $\text{let } x = e_1 \text{ in } e_2$
  - If $e_1$ evaluates to $v$, i.e., $e_1 \Rightarrow v_1$
  - If $e_2\{v_1/x\}$ evaluates to $v_2$, i.e., $e_2\{v_1/x\} \Rightarrow v_2$
  - Then $e$ evaluates to $v_2$, i.e., $\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2$

\[
\begin{align*}
  e_1 & \Rightarrow v_1 \\
  e_2\{v_1/x\} & \Rightarrow v_2 \\
\text{let } x = e_1 \text{ in } e_2 & \Rightarrow v_2
\end{align*}
\]
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 \( \text{let } x = 4 \text{ in } x+3 \Rightarrow 7 \)
Derivations

\[ \begin{align*}
\text{Goal: show that} & \quad \text{let } x = 4 \text{ in } x+3 \Rightarrow 7 \\
& \quad 4 \Rightarrow 4 \quad 3 \Rightarrow 3 \quad 7 \text{ is } 4+3 \\
& \quad 4 \Rightarrow 4 \\
& \quad 4+3 \Rightarrow 7 \\
& \quad \text{let } x = 4 \text{ in } x+3 \Rightarrow 7
\end{align*} \]
Quiz 1

What is derivation of the following judgment?

\[ 2 + (3 + 8) \Rightarrow 13 \]

(a)
\[
\begin{align*}
2 & \Rightarrow 2 \\
3 + 8 & \Rightarrow 11 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]

(b)
\[
\begin{align*}
8 & \Rightarrow 8 \\
3 & \Rightarrow 3 \\
11 & \text{is} 3+8 \\
\hline
2 \Rightarrow 2 \\
3 + 8 & \Rightarrow 11 \\
13 & \text{is} 2+11 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]

(c)
\[
\begin{align*}
3 & \Rightarrow 3 \\
8 & \Rightarrow 8 \\
\hline
3 + 8 & \Rightarrow 11 \\
2 & \Rightarrow 2 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]

What is derivation of the following judgment?
What is derivation of the following judgment?

\[ 2 + (3 + 8) \Rightarrow 13 \]

(a) \[ \begin{align*}
2 & \Rightarrow 2 \quad 3 + 8 \Rightarrow 11 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*} \]

(b) \[ \begin{align*}
8 & \Rightarrow 8 \\
3 & \Rightarrow 3 \\
11 & \text{is 3+8} \\
\hline
2 & \Rightarrow 2 \\
3 + 8 & \Rightarrow 11 \\
13 & \text{is 2+11} \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*} \]

(c) \[ \begin{align*}
3 & \Rightarrow 3 \\
8 & \Rightarrow 8 \\
\hline
3 + 8 & \Rightarrow 11 \\
\hline
2 & \Rightarrow 2 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*} \]
Definitional Interpreter

- The style of rules lends itself directly to the implementation of an interpreter as a recursive function.

```ocaml
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
```

- \( n \Rightarrow n \)
- \( e1 \Rightarrow n1 \quad e2 \Rightarrow n2 \quad n3 \text{ is } n1+n2 \)
- \( e1 + e2 \Rightarrow n3 \)
- \( e1 \Rightarrow v1 \quad e2\{v1/x\} \Rightarrow v2 \)
- \( \text{let } x = e1 \text{ in } e2 \Rightarrow v2 \)
Derivations = Interpreter Call Trees

\[ 4 \Rightarrow 4 \quad 3 \Rightarrow 3 \quad 7 \text{ is } 4+3 \]
\[ 4 \Rightarrow 4 \quad 4+3 \Rightarrow 7 \]
\[ \text{let } x = 4 \text{ in } x+3 \Rightarrow 7 \]

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

\[ \text{eval } \text{Num} \ 4 \Rightarrow 4 \quad \text{eval } \text{Num} \ 3 \Rightarrow 3 \quad 7 \text{ is } 4+3 \]
\[ \text{eval } \text{(subst} \ 4 \text{ “x”} \]
\[ \text{eval } \text{Num} \ 4 \Rightarrow 4 \quad \text{Plus}(\text{Ident} (“x”), \text{Num} \ 3)) \Rightarrow 7 \]
\[ \text{eval } \text{Let} (“x”, \text{Num} \ 4, \text{Plus}(\text{Ident} (“x”), \text{Num} \ 3)) \Rightarrow 7 \]
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 $\text{eval } e = \{ v \mid e \Rightarrow v \}$
  - 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

- $\cdot$ is the empty environment
- $A, x:v$ is the environment that extends $A$ with a mapping from $x$ to $v$
  - We write $x:v$ instead of $\cdot, x:v$ for brevity
  - NB. if $A$ maps $x$ to some $v'$, then that mapping is *shadowed* by $v$ in $A, x:v$

Lookup $A(x)$ is defined as follows

\[
\begin{align*}
(x) & = \text{undefined} \\
(A, y:v)(x) & = \begin{cases} 
  v & \text{if } x = y \\
  A(x) & \text{if } x \not\leftrightarrow y \text{ and } A(x) \text{ defined} \\
  \text{undefined} & \text{otherwise}
\end{cases}
\end{align*}
\]
Definitional Interpreter: Environments

An environment is just a list of mappings, which are just pairs of variable to value - called an association list

```ocaml
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
```

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** \( A \):
  \[ A(x) = v \]
  \[ A; x \Rightarrow v \]

- **Extend environment** \( A \) **with mapping from** \( x \) **to** \( v \):
  \[ A; e_1 \Rightarrow v_1 \]
  \[ A, x : v_1 ; e_2 \Rightarrow v_2 \]
  \[ A ; \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2 \]

- **Add** \( e_1 \) **and** \( e_2 \) **in** \( A \):
  \[ A; e_1 \Rightarrow n_1 \]
  \[ A; e_2 \Rightarrow n_2 \]
  \[ n_3 \text{ is } n_1 + n_2 \]
  \[ A; e_1 + e_2 \Rightarrow n_3 \]
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) 
\[
\begin{align*}
  \ & x \Rightarrow 3 \\
  \ & 2 \Rightarrow 2 \\
  \hline
  \ & 5 \text{ is } 3+2 \\
  \ & 3 \Rightarrow 3 \\
  \ & x+2 \Rightarrow 5 \\
  \hline
  \ & \text{let } x=3 \text{ in } x+2 \Rightarrow 5
\end{align*}
\]

(b) 
\[
\begin{align*}
  \ & x:3; x \Rightarrow 3 \\
  \ & x:3; 2 \Rightarrow 2 \\
  \hline
  \ & 5 \text{ is } 3+2 \\
  \ & \cdot; 3 \Rightarrow 3 \\
  \ & x:3; x+2 \Rightarrow 5 \\
  \hline
  \ & \cdot; \text{let } x=3 \text{ in } x+2 \Rightarrow 5
\end{align*}
\]

(c) 
\[
\begin{align*}
  \ & x:2; x\Rightarrow3 \\
  \ & x:2; 2\Rightarrow2 \\
  \hline
  \ & 5 \text{ is } 3+2 \\
  \ & \cdot; \text{let } x=3 \text{ in } x+2 \Rightarrow 5
\end{align*}
\]
Quiz 2

What is a derivation of the following judgment?

•; let x=3 in x+2 ⇒ 5

(a)  
\[
\begin{align*}
&x \Rightarrow 3 \quad 2 \Rightarrow 2 \quad 5 \text{ is } 3+2 \\
&3 \Rightarrow 3 \quad x+2 \Rightarrow 5 \\
\text{let } x=3 \text{ in } x+2 \Rightarrow 5
\end{align*}
\]

(b)  
\[
\begin{align*}
&x:3; \ x \Rightarrow 3 \quad x:3; \ 2 \Rightarrow 2 \quad 5 \text{ is } 3+2 \\
&•; \ 3 \Rightarrow 3 \quad x:3; \ x+2 \Rightarrow 5 \\
&•; \ \text{let } x=3 \text{ in } x+2 \Rightarrow 5
\end{align*}
\]

(c)  
\[
\begin{align*}
&x:2; \ x \Rightarrow 3 \quad x:2; \ 2 \Rightarrow 2 \quad 5 \text{ is } 3+2 \\
&•; \ let \ x=3 \text{ in } x+2 \Rightarrow 5
\end{align*}
\]
Adding Conditionals to Micro-OCaml

\[ e ::= x | v | e + e | \text{let } x = e \text{ in } e \\
  | \text{eq0 } e | \text{if } e \text{ then } e \text{ else } e \]

\[ v ::= n | \text{true} | \text{false} \]

- In terms of interpreter definitions:

\[
\text{type } \text{exp} = \\
| \text{Val of value} \\
| ... (* as before *) \\
| \text{Eq0 of exp} \\
| \text{If of exp * exp * exp} \\
\]

\[
\text{type } \text{value} = \\
| \text{Int of int} \\
| \text{Bool of bool} \\
\]
Rules for Eq0 and Booleans

- **Booleans evaluate to themselves**
  - $A; \text{false} \Rightarrow \text{false}$

- **eq0 tests for 0**
  - $A; \text{eq0 } 0 \Rightarrow \text{true}$
  - $A; \text{eq0 } 3+4 \Rightarrow \text{false}$
### Rules for Conditionals

- **Notice that only one branch is evaluated**
  - \( A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v \)
  
<table>
<thead>
<tr>
<th>A; e₁ ⇒ true</th>
<th>A; e₂ ⇒ v</th>
</tr>
</thead>
<tbody>
<tr>
<td>A; if e₁ then e₂ else e₃ ⇒ v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A; e₁ ⇒ false</th>
<th>A; e₃ ⇒ v</th>
</tr>
</thead>
<tbody>
<tr>
<td>A; if e₁ then e₂ else e₃ ⇒ v</td>
<td></td>
</tr>
</tbody>
</table>

- \( A; \text{if eq0 0 then 3 else 4 ⇒ 3} \)
- \( A; \text{if eq0 1 then 3 else 4 ⇒ 4} \)
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?

\[ \text{\textbullet; if eq0 3-2 then 5 else 10 } \Rightarrow 10 \]

(a)
\[
\begin{align*}
\text{\textbullet; } 3 & \Rightarrow 3 \\
\text{\textbullet; } 2 & \Rightarrow 2 \\
3-2 & \text{ is 1}
\end{align*}
\]

\[ \text{\textbullet; eq0 3-2 } \Rightarrow \text{false} \\
\text{\textbullet; 10 } \Rightarrow 10
\]

\[ \text{\textbullet; if eq0 3-2 then 5 else 10 } \Rightarrow 10
\]

(b)
\[
\begin{align*}
3 & \Rightarrow 3 \\
2 & \Rightarrow 2 \\
3-2 & \text{ is 1}
\end{align*}
\]

\[ \text{eq0 3-2 } \Rightarrow \text{false} \\
10 & \Rightarrow 10
\]

\[ \text{if eq0 3-2 then 5 else 10 } \Rightarrow 10
\]

(c)
\[
\begin{align*}
3 & \Rightarrow 3 \\
2 & \Rightarrow 2 \\
3-2 & \text{ is 1}
\end{align*}
\]

\[ \text{3-2 } \Rightarrow 1 \\
1 & \neq 0
\]

\[ \text{eq0 3-2 } \Rightarrow \text{false} \\
\text{10 } \Rightarrow 10
\]

\[ \text{if eq0 3-2 then 5 else 10 } \Rightarrow 10
\]
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

Pattern match will fail if \texttt{e1} or \texttt{e2} is not an \texttt{Int}; this is dynamic type checking! (But \texttt{Match_failure} not the best way to signal an error)

Basically both rules for \texttt{eq0} in this one snippet

Both \texttt{if} rules here
Adding Closures to Micro-OCaml

\[ e ::= x | v | e + e | \text{let } x = e \text{ in } e \]
\[ | \text{eq0 } e | \text{if } e \text{ then } e \text{ else } e \]
\[ | ee | \text{fun } x \rightarrow e \]

\[ v ::= n | \text{true} | \text{false} | (A, \lambda x.e) \]

- 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

- **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 $v_1$

- **Left to you:** How will the definitional interpreter change?
Rule for Closures: Dynamic Scoping

- 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

- Easy to see dynamic scoping was an implementation error!
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 $t ::= \text{bool} \mid \text{int}$
- Judgment $\vdash e : t$ 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`

  \[ \vdash \text{true} : \text{bool} \quad \vdash \text{false} : \text{bool} \]

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

  \[ \vdash e : \text{int} \quad \vdash \text{eq0} e : \text{bool} \]

- Conditionals

  \[ \vdash e_1 : \text{bool} \quad \vdash e_2 : t \quad \vdash e_3 : t \]

  \[ \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]
Handling Binding

- What about the types of variables?
  - Taking inspiration from the environment-style operational semantics, what could you do?

- Change judgment to be $G \vdash e : t$ which says $e$ has type $t$ under type environment $G$
  - $G$ is a map from variables $x$ to types $t$
    - Analogous to map $A$, but maps vars to types, not values

- What would be the rules for $let$, and variables?
Type Checking with Binding

- Variable lookup

\[
\frac{G(x) = t}{G \vdash x : t}
\]

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

- Let binding

\[
\frac{G \vdash e_1 : t_1 \quad G, x : t_1 \vdash e_2 : t_2}{G \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}
\]

\[
\frac{A; e_1 \Rightarrow v_1 \quad A, x : v_1; e_2 \Rightarrow v_2}{A; \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2}
\]
Scaling up

- Operational semantics (and similarly styled typing rules) 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
Scaling up: Web Assembly

WebAssembly Specification

Release 1.1 (Draft, Mar 12, 2021)

Editor: Andreas Rossberg

Latest Draft: https://webassembly.github.io/spec/core/
Issue Tracker: https://github.com/webassembly/spec/issues/

- Introduction
  - Introduction
  - Overview

- Structure
  - Conventions
  - Values
  - Types
  - Instructions
  - Modules

- Validation
  - Conventions
Formal Notation

Note:
This section gives a brief explanation of the notation for specifying execution formally. For the interested reader, a more thorough introduction can be found in respective textbooks. [2]

The formal execution rules use a standard approach for specifying operational semantics, rendering them into reduction rules. Every rule has the following general form:

\[ \text{configuration} \xrightarrow{\text{rule}} \text{configuration} \]

A configuration is a syntactic description of a program state. Each rule specifies one step of execution. As long as there is at most one reduction rule applicable to a given configuration, reduction – and thereby execution – is deterministic. WebAssembly has only very few exceptions to this, which are noted explicitly in this specification.

For WebAssembly, a configuration typically is a tuple \((S; F; \text{instr})\) consisting of the current store \(S\), the call frame \(F\) of the current function, and the sequence of instructions that is to be executed. (A more precise definition is given later.)

To avoid unnecessary clutter, the store \(S\) and the frame \(F\) are omitted from reduction rules that do not touch them.