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 \( e \Rightarrow v \) can be easily turned into idiomatic OCaml code
  - The language’s expressions \( e \) and values \( v \) have corresponding OCaml datatype 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
Micro-OCaml Expression Grammar

e ::= x | n | e + e | let x = e in e

- e, 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

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

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 \text{ is an expression } e
Abstract Syntax = Structure

- Here, the grammar for e is describing its abstract syntax tree (AST), i.e., e’s structure

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

corresponds to (in defn interpreter)

```plaintext
type id = string
type num = int
type exp =
  | Ident of id
  | Num of num
  | Plus of exp * exp
  | Let of id * exp * exp
```
Values

- An expression’s final result is a value. What can values be?
  \[
  v ::= n
  \]

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

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

- These rules will allow us to show things like
  
  - $1+3 \Rightarrow 4$
    - $1+3$ 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., $e_1 \Rightarrow n_1$
  - 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 $v$, i.e., $e_1 \Rightarrow v$
  - 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 of Inference

- We can use a more compact notation for the rules we just presented: **rules of inference**
  - Has the following format:
    
    \[
    \begin{array}{c}
    H_1 \quad \ldots \quad H_n \\
    \hline
    C
    \end{array}
    \]
  - 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 will use inference rules to speak about evaluation.
Rules of Inference: Num and Sum

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

2. 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., $e_1 + e_2 \Rightarrow n_3$
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., \( \text{let} \ x = e_1 \ \text{in} \ e_2 \Rightarrow v_2 \)

\[
\begin{array}{c|c|c}
  e_1 \Rightarrow v_1 & e_2\{v_1/x\} \Rightarrow v_2 \\
  \hline
  \text{let} \ x = e_1 \ \text{in} \ e_2 \Rightarrow v_2
\end{array}
\]
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 let x = 4 in x+3 ⇒ 7
## Derivations

<table>
<thead>
<tr>
<th>$n$ $\Rightarrow$ $n$</th>
<th>$e_1$ $\Rightarrow$ $n_1$</th>
<th>$e_2$ $\Rightarrow$ $n_2$</th>
<th>$n_3$ is $n_1 + n_2$</th>
<th>$e_1 + e_2$ $\Rightarrow$ $n_3$</th>
</tr>
</thead>
</table>

**Goal:** show that

let $x = 4$ in $x + 3 \Rightarrow 7$

\[
4 \Rightarrow 4 \quad 3 \Rightarrow 3 \quad 7 \text{ is } 4 + 3
\]

\[
4 \Rightarrow 4 \quad 4 + 3 \Rightarrow 7
\]

\[
\text{let } x = 4 \text{ in } x + 3 \Rightarrow 7
\]
Quiz 1

What is derivation of the following judgment?

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

(a)

$$2 \Rightarrow 2 \quad 3 + 8 \Rightarrow 11$$

$$-----------$$

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

(b)

$$3 \Rightarrow 3 \quad 8 \Rightarrow 8$$

$$-----------$$

$$3 + 8 \Rightarrow 11$$

$$2 \Rightarrow 2$$

$$-----------$$

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

(c)

$$8 \Rightarrow 8$$

$$3 \Rightarrow 3$$

$$11 \text{ is } 3+8$$

$$-----------$$

$$2 \Rightarrow 2 \quad 3 + 8 \Rightarrow 11$$

$$13 \text{ is } 2+11$$

$$-----------$$

$$2 + (3 + 8) \Rightarrow 13$$
Quiz 1

What is derivation of the following judgment?

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

(a)

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

(b)

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

(c)

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

Trace of evaluation of `eval` function corresponds to a derivation by the rules

- `e1 ⇒ n1`  
- `e2 ⇒ n2`  
- `n3 is n1+n2`  
- `e1 + e2 ⇒ n3`  
- `el ⇒ v1`  
- `e2{v1/x} ⇒ v2`  
- `let x = e1 in e2 ⇒ v2`
Derivations = Interpreter Call Trees

\[
\begin{align*}
4 \Rightarrow 4 & \quad 3 \Rightarrow 3 & \quad 7 \text{ is } 4+3 \\
\hline
4 \Rightarrow 4 & \quad 4+3 \Rightarrow 7 \\
\hline
\text{let } x = 4 \text{ in } x+3 \Rightarrow 7
\end{align*}
\]

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

\[
\begin{align*}
\text{eval Num } 4 \Rightarrow 4 & \quad \text{eval Num } 3 \Rightarrow 3 & \quad 7 \text{ is } 4+3 \\
\hline
\text{eval } (\text{subst } 4 \text{ "x"}) \\
\text{eval Num } 4 \Rightarrow 4 & \quad \text{Plus} (\text{Ident}("x"), \text{Num 3}) \Rightarrow 7 \\
\hline
\text{eval Let("x", Num 4, Plus(Ident("x"), Num 3))} \Rightarrow 7
\end{align*}
\]
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 $e \not\Rightarrow v$
- 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

- 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
  - 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

- $\bullet$ 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: $A'$ \textit{shadows} definitions in $A$
- For brevity, can write $\bullet, A$ as just $A$
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

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

Look up variable \( x \) in environment \( A \)

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

Extend environment \( A \) with mapping from \( x \) to \( v_1 \)

\[ A; e_1 \Rightarrow n_1 \]
\[ A; e_2 \Rightarrow n_2 \]
\[ n_3 \text{ is } n_1 + n_2 \]
\[ A; e_1 + e_2 \Rightarrow n_3 \]
What is a derivation of the following judgment?

•; let x=3 in x+2 ⇒ 5

(a) 

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

(b) 

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

(c) 

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

What is a derivation of the following judgment?

•; let x=3 in x+2 ⇒ 5

(a)  

\[
\begin{align*}
\text{x} & \Rightarrow 3 \\
2 & \Rightarrow 2 \\
\text{5 is} & \\
\hline
3+2 & \\
\text{-------------------} \\
\text{3} & \Rightarrow 3 \\
\text{x+2} & \Rightarrow 5 \\
\text{----------------------} \\
\text{let x=3 in x+2} & \Rightarrow 5
\end{align*}
\]

(b)  

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

(c)  

\[
\begin{align*}
\text{x:2; x} & \Rightarrow 3 \\
x:2; 2 & \Rightarrow 2 \\
\text{5 is x:2+2} & \\
\hline
\text{x:2; x:3+2} & \\
\text{----------------} \\
\text{•; let x=3 in x+2} & \Rightarrow 5
\end{align*}
\]
Definitional Interpreter: Environments

```ml
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
```
Definitional Interpreter: Evaluation

```ocaml
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
```
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:

```ocaml
type exp =
    | Val of value
    | ... (* as before *)
    | Eq0 of exp
    | If of exp * exp * exp

type value =
    Int of int
    | Bool of bool
```
## Rules for Eq0 and Booleans

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

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

<table>
<thead>
<tr>
<th>$A; \text{true} \Rightarrow \text{true}$</th>
<th>$A; e \Rightarrow 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A; \text{false} \Rightarrow \text{false}$</td>
<td>$A; \text{eq0 } e \Rightarrow \text{true}$</td>
</tr>
<tr>
<td></td>
<td>$A; e \Rightarrow v \quad v \neq 0$</td>
</tr>
<tr>
<td></td>
<td>$A; \text{eq0 } e \Rightarrow \text{false}$</td>
</tr>
</tbody>
</table>
Rules for Conditionals

<table>
<thead>
<tr>
<th>A; e₁ (\Rightarrow) true</th>
<th>A; e₂ (\Rightarrow) v</th>
</tr>
</thead>
<tbody>
<tr>
<td>A; if e₁ then e₂ else e₃ (\Rightarrow) v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A; e₁ (\Rightarrow) false</th>
<th>A; e₃ (\Rightarrow) v</th>
</tr>
</thead>
<tbody>
<tr>
<td>A; if e₁ then e₂ else e₃ (\Rightarrow) v</td>
<td></td>
</tr>
</tbody>
</table>

- 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 ⇒ 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?

•; 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
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
    | 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

Basically both rules for eq0 in this one snippet

Both if rules here
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`
    
    \[ \vdash \text{e} : \text{int} \]
    
    \[ \vdash \text{eq0 e : bool} \]

- **Conditionals**
  
  \[ \vdash \text{e1 : bool} \quad \vdash \text{e2 : t} \quad \vdash \text{e3 : t} \]
  
  \[ \vdash \text{if e1 then e2 else e3 : t} \]
Handling Binding

What about the types of variables?

- Taking inspiration from the environment-style operational semantics, what could you do?

Change judgment to be \( G \vdash e : t \) which says \( e \) has type \( t \) under type environment \( G \)

- \( G \) is a map from variables \( x \) to types \( t \)
  - Analogous to map A, maps vars to types, not values

What would be the rules for \texttt{let}, and variables?
Type Checking with Binding

- **Variable lookup**
  \[
  G(x) = t \\
  \text{G} \vdash x : t
  \]

- **Let binding**
  \[
  G \vdash e_1 : t_1 \\
  G, x : t_1 \vdash e_2 : t_2 \\
  \text{G} \vdash \text{let } x = e_1 \text{ in } e_2 : t_2
  \]

  analogous to

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

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