Environment Semantics

CAS CS 320: Concepts of Programming Languages

November 5, 2026

Substitution is a good foundation for formal semantics because it's simple (and sufficient for most theoretical purposes) but it's not well-suited for a usable definitional interpreter. Bottom line: it's *slow*. This is because every β-reduction needs to "crawl" the entire body of the called function to look for variables that need to be subbed. To make this concrete, consider the following family of expressions.

$$e_n \triangleq (\lambda x_{n-1}.\ldots.\lambda x_0.x_0)(\lambda x.x)\ldots(\lambda x.x)$$

Every β-step in the evaluation of e_n requires a substitution of the form

$$[\lambda x.x/x_k](\lambda x_{k-1}.\ldots.\lambda x_0.x_0) = \lambda x_{k-1}.\ldots.\lambda x_0.x_0$$

which takes time that's linear in k (the substitution function has to go under each binder to see if there's an x_k (there isn't)). This means evaluating e_n via our substitution semantics takes $\Omega(n^2)$ time. And we can verify this by benchmarking our big-step CBV substitution evaluator for the λ-calculus (more on that later).

This is, of course, a hand-picked example but it's a fairly robust phenomenon. Our goal here is to address this, and it turns out there's a simple fix. Earlier we've said functional PLs like the λ-calculus don't need state in their formal semantics, but this doesn't mean it's *disallowed*, or that it's not useful to introduce some state. This is the thrust of environment semantics; rather than substituting at application, we *store variable bindings* as state that we can refer to during evaluation.

Environment semantics for the λ-calculus. An ***environment*** is a mapping of variables to values.[1] Note that a variable can be mapped to at most one value. We call a variable and its associated value in the environment a ***variable binding*** (notation: $x \mapsto v$) and we say that x is ***bound*** to v. We notate concrete environments by a collection of variable bindings surrounded in curly brackets, e.g.,

$$\{x \mapsto \lambda y.y \ , \ y \mapsto 2 \ , \ z \mapsto \mathsf{true}\}$$

[1] We can also define environments so that variables are mapped to general expressions. This is necessary when defining CBN environment semantics. We'll be focusing on CBV environment semantics here.

meta-variable	$\mathcal{E}$	`env`
empty environment	$\varnothing$	`Env.empty`
the value x is bound to in $\mathcal{E}$	$\mathcal{E}(x)$	`Env.find x env`
$\mathcal{E}$ with x bound to v	$\mathcal{E}[x \mapsto v]$	`Env.add x v env`
x is bound to v in $\mathcal{E}$	$\mathcal{E}(x) = v$	`Env.find_opt x env = Some v`

Figure 1: Basics of environments

$$\frac{}{\langle\ \mathcal{E}\ ,\ x\ \rangle \Downarrow \mathcal{E}(x)}\ \textsc{var} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \lambda x.e\ \rangle \Downarrow \lambda x.e}\ \textsc{fun}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \lambda x.e \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad \langle\ \mathcal{E}[x \mapsto v_2]\ ,\ e\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ e_1 e_2\ \rangle \Downarrow v}\ \textsc{app}$$

Figure 2: Big-step CBV environment semantics for the λ-calculus with dynamic scoping

We use OCaml `Map`s to represent environments.

```
module Env = Map.Make(String) (* maps with string keys *)
```

The basic notation for environments is given in Figure 1, along with corresponding functions in the `Env` module.

Big-step environment semantics rules are defined over judgments of the form

$$\langle\ \mathcal{E}\ ,\ e\ \rangle \Downarrow v$$

which we read as "the expression e evaluates to v in the environment $\mathcal{E}$." Recall that $\langle\ \mathcal{E}\ ,\ e\ \rangle$ is what we called a ***configuration***, i.e., an expression together with state. The first version of the semantic rules we'll consider is given in Figure 2. It's worth comparing these rules to the big-step CBV substitution semantics for the λ-calculus we've given previously.

- The FUN rule is roughly the same; functions evaluate to themselves (and the environment is ignored).
- Now we actually have a VAR rule; since variables aren't substituted at application time, we need to figure out at evaluation time what they're bound to. We do this by *looking up the variable in the environment.* This is the core of environment semantics; instead of substituting into variables eagerly, we store their values in the environment, and then determine their values lazily when we come across them.
- The APP rule also parallels the corresponding rule in the substitution semantics, but instead of substituting, we add a binding to the environment.

Exercise 1. *Determine a value v such that*

$$\langle\ \varnothing\ ,\ (\lambda x.\lambda y.x)((\lambda x.x)(\lambda y.y))(\lambda z.z)\ \rangle \Downarrow v$$

Also provide a derivation.

Exercise 2. *Determine a value v such that*

$$\langle\ \varnothing\ ,\ (\lambda f.(\lambda x.fx)(\lambda y.x))(\lambda z.z)\ \rangle \Downarrow v$$

Also provide a derivation.

Exercise 3. *Determine an irreducible expression e such that*

$$(\lambda f.(\lambda x.fx)(\lambda y.x))(\lambda z.z) \longrightarrow^{\star} e$$

according to the small-step CBV semantics given previously. Provide a trace. Additionally, compare this to the previous problem. Is e a value according to the small-step CBV semantics given previously?

Dynamic and Lexical Scoping. The rules in Figure 2 work, but they don't model the same computational behavior as the substitution semantics we've given previously.

Exercise 4. ***(Challenge)*** *Determine an expression e such that $e \Downarrow v_1$ according to the CBV substitution semantics (as given in previous notes) and $e \Downarrow v_2$ according to the environment semantics as given in Figure 2, and $v_1 \neq v_2$.*

The problem comes down to how variables are *scoped*. Scope is an overused term so it's difficult to give a formal definition of scope that agrees with every PL's notion of it. In rough terms, the ***scope*** of a binding is "when and where" it's available. We also say that the scope of a variable is the scope of its corresponding binding. There are two major scoping disciplines: ***dynamic scoping*** and ***lexical scoping***. That said, dynamic scoping has fallen out of fashion; most if not all modern PLs implement some form of lexical scoping.

Dynamic scoping is a "temporal" approach to binding availability. Under dynamic scoping a variable binding is available if it was added to the environment at some point earlier in time during the execution of the program. The most notable PL with dynamic scoping is Bash. Here's an example that highlights this:

```
> function f { X=1 }  # define f
> f                   # call f
> echo $X             # print X
1
```

The details here aren't terribly important; the key point is that it's somewhat surprising (at least if you're used to modern PLs) that `X` is bound to anything at all, let alone `1`. Since the function is called *before* printing `X`, and the

function binds `X`, it follows under dynamic scoping that the binding for `X` is available. The analogous Python code doesn't fly.

```
>>> def f():   # define f
...     x = 1
...
>>> f()        # call f
>>> print(x)   # print x
NameError: name 'x' is not defined
```

This is because Python implements ***lexical scoping***. Under lexical scoping, the availability of a variable binding is delimited by a textual (i.e., lexical) feature of the program. For example, a variable defined in the body of a Python function is not available outside of the function definition, i.e., the end of the function definition is the end of the scope of the variable. OCaml also implements lexical scoping, and the scope is made incredibly clear by the syntax; given an expression of the form `let x = e1 in e2` the scope of `x` is `e2`. Outside of `e2`, the binding for `x` defined by this let-expression is not available.

Why is lexical scoping ubiquitous? It's partly historical, and partly because it's easier to reason about as a programmer. Imagine that you're programming and you call a function in a library that you didn't write. Under dynamic scoping, this function call has the potential to pollute the environment with new bindings, and even shadow existing bindings. That's scary. We generally like the impact of code to be encapsulated to some degree, and even more so when it's code we didn't write. Under lexical scoping, variable bindings have a locality to them, which makes the behavior of our program easier to track.

Why dynamically scope? Again, it's partly historical, but it's also because it's *far* easier to implement.

The upshot: the environment semantic rules above (Figure 2) implement dynamic scoping. To make it easier to demonstrate this, we extend our language to include let-expressions and Booleans. The abstract syntax is extended to include:

$$\begin{aligned} e &::= \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{true} \mid \mathsf{false} \\ v &::= \lambda x.e \mid \mathsf{true} \mid \mathsf{false} \end{aligned}$$

The small-step CBV substitution semantics is extended as:

$$\frac{e_1 \longrightarrow e_1'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \longrightarrow \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2}\ \text{LET}_1 \qquad \frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e_2 \longrightarrow [v/x]e_2}\ \text{LET}_2$$

The big-step CBV environment semantics is extended as:

$$\frac{}{\langle\ \mathcal{E}\ ,\ \mathsf{true}\ \rangle \Downarrow \mathsf{true}}\ \text{TRUE} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \mathsf{false}\ \rangle \Downarrow \mathsf{false}}\ \text{FALSE}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}[x \mapsto v_1]\ ,\ e_2\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \rangle \Downarrow v}\ \text{LET}$$

Exercise 5. *Give a trace of the following multi-step judgment.*

$$\mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ \mathsf{let}\ y = \lambda z.x\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{false}\ \mathsf{in}\ y\ \mathsf{true} \longrightarrow^\star \mathsf{true}$$

Exercise 6. *Give a derivation of the following judgment.*

$$\langle\ \varnothing\ ,\ \mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ \mathsf{let}\ y = \lambda z.x\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{false}\ \mathsf{in}\ y\ \mathsf{true}\ \rangle \Downarrow \mathsf{false}$$

Exercise 7. *Based on the previous exercises, give a closed λ-expression e (without let-expressions or Booleans) such that $e \longrightarrow^\star \lambda x.x$ and $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow \lambda x.\lambda y.x$. Also provide a trace and derivation, respectively. Note that this solves the challenge problem above. Hint: Think about how the semantics of let-expressions compares to that of application.*

The previous exercises demonstrate that a variable in the body of a function may be dynamically re-bound by let-expressions. This isn't a "bad" thing, but it doesn't model the lexical scoping we're familiar with from our substitution semantics. The issue: because variable bindings are determined lazily, the value of x in $\lambda z.x$ isn't determined until the function is *called*. The binding may *change* between the time the function is defined and when it's called.

Closures and function values. A ***closure*** is an expression together with an environment, written $(\!|\ \mathcal{E}\ ,\ e\ |\!)$. It's similar to a configuration, and in a sense this is on purpose. Closures are used to "remember" what the environment looked like at some point during evaluation; if the environment changes, closures provide a back-up to refer to. This can be used to address the issue above: a function needs to remember what the environment looked like when it was defined; this is the environment we need to use when we evaluate applications. Thus, *we take the values of functions to be closures.*

$$v ::= (\!|\ \mathcal{E}\ ,\ \lambda x.e\ |\!) \mid \mathsf{true} \mid \mathsf{false}$$

Note that this is the first time a value isn't necessarily an expression. The big-step CBV environment semantics for this λ-calculus (with lexical scoping) is given in Figure 3. The core feature of this system is the APP rule, and the important part is colored in red. A function evaluates to a closure that "remembers" the state of the environment when it was defined. The body of the function is then evaluated *in the environment from the closure.*

Exercise 8. *Give a derivation of the following judgment using the rules in Figure 3.*

$$\langle\ \varnothing\ ,\ \mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ \mathsf{let}\ y = \lambda z.x\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{false}\ \mathsf{in}\ y\ \mathsf{true}\ \rangle \Downarrow \mathsf{true}$$

Exercise 9. *Let e be the following expression*

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\\ \mathsf{let}\ g = \lambda y.x\ \mathsf{in}\\ \mathsf{let}\ x = \mathsf{false}\ \mathsf{in}\\ \mathsf{let}\ f = \lambda y.gx\ \mathsf{in}\\ \mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\\ f\end{array}$$

$$\frac{}{\langle\ \mathcal{E}\ ,\ x\ \rangle \Downarrow \mathcal{E}(x)}\ \textsc{var} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \lambda x.e\ \rangle \Downarrow (\!|\ \mathcal{E}\ ,\ \lambda x.e\ |\!)}\ \textsc{fun}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow (\!|\ \mathcal{E}'\ ,\ \lambda x.e\ |\!) \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad \langle\ \mathcal{E}'[x \mapsto v_2]\ ,\ e\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ e_1 e_2\ \rangle \Downarrow v}\ \textsc{app}$$

$$\frac{}{\langle\ \mathcal{E}\ ,\ \mathsf{true}\ \rangle \Downarrow \mathsf{true}}\ \textsc{true} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \mathsf{false}\ \rangle \Downarrow \mathsf{false}}\ \textsc{false}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}[x \mapsto v_1]\ ,\ e_2\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \rangle \Downarrow v}\ \textsc{let}$$

Figure 3: Big-step CBV environment semantics for the λ-calculus with let-expressions and Booleans

Determine the value v such that $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$. You don't need to provide a derivation.

Exercise 10. *Determine the value v such that*

$$\langle\ \varnothing\ ,\ (\lambda f.\lambda x.fx)(\mathsf{let}\ z = \mathsf{true}\ \mathsf{in}\ \lambda y.yz)\ \rangle \Downarrow v$$

You don't need to provide a derivation.

Exercise 11. *Determine an expression e and a value v such that $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$ under dynamic scoping (Figure 2) but $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$ is not derivable under lexical scoping (Figure 3).*

Exercise 12. *Determine an expression e and a value v such that $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$ under lexical scoping (Figure 3) but $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$ is not derivable under dynamic scoping (Figure 2).*

Exercise 13. *Extend the type system for the domain-free STLC as given previously to the domain-free STLC with let-expressions and Booleans, as described above.*

Exercise 14. *Using the type system you gave in the previous exercise, show that the semantics with dynamic scoping (Figure 2) doesn't satisfy type safety, i.e., determine an expression e, type τ, and value v such that $\varnothing \vdash e : \tau$ and $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$ but it's not the case that $\varnothing \vdash v : \tau$.*

Recursion and named closures. Environment semantics also has the benefit that it's easier to define recursion than in substitution semantics.[2] But we don't get it immediately.

Exercise 15. *Explain why let-expressions as defined above are not recursive.*

[2]It's possible to define in substitution semantics via what are called fixpoint combinators.

In this setting, a function is recursive if it's given a name via a let-expression, and the name can be used in the body of the function. Recursive functions pose an issue for environment semantics because they seem to require infinite regress. If a function named f can be used in its own body, then it must evaluate to a closure $(\!|\ \mathcal{E}\ ,\ \lambda x.e\ |\!)$ such that $\mathcal{E}(f) = (\!|\ \mathcal{E}\ ,\ \lambda x.e\ |\!)$. But then this closure would have the structure

$$(\!|\ \{\ldots f \mapsto (\!|\ \{\ldots f \mapsto (\!|\ \{\ldots f \mapsto (\!|\ \mathcal{E}\ ,\ \lambda x.e\ |\!)\}\ ,\ \lambda x.e\ |\!)\ldots\}\ ,\ \lambda x.e\ |\!)\ldots\}\ ,\ \lambda x.e\ |\!)$$

and so on *ad infinitum*.

One solution is to build this infinite structure *lazily*. At application time, we can put a copy of the called closure into its own calling environment. In order to do this, (1) we need to distinguish those closures that are recursive and (2) closures need to know their own names.

A ***named closure*** is an expression, together with an environment, *together with a name*, written $(\!|\ \mathcal{E}\ ,\ e\ ,\ x\ |\!)$. If a closure has a name, then we treat it as recursive.

We update the semantics to allow the values given to variables in let-expressions to be named closures (Figure 4). We also add some additional constructs to make our examples more interesting. We only do this when the expression that defines the value in the binding is an anonymous function; it doesn't make sense to make a let-expression recursive otherwise because there'd be no way for it to terminate (i.e., no way to write a base case). Unlike OCaml, we will take all let-expressions to be (potentially) recursive.

Exercise 16. *Determine the value v such that*

$$\langle\ \varnothing\ ,\ \mathsf{let}\ f = \lambda x.fx\ \mathsf{in}\ f\ \rangle \Downarrow v$$

Also provide a derivation.

Exercise 17. *Determine the value v such that*

$$\langle\ \{f \mapsto (\!|\ \varnothing\ ,\ \lambda x.\mathsf{if}\ x = 1\ \mathsf{then}\ g\ (x+1)\ \mathsf{else}\ x\ ,\ g\ |\!)\}\ ,\ f\ 1\ \rangle \Downarrow v$$

You don't need to provide a derivation.

Exercise 18. *In the PL given from Figure 4, write an expression that evaluates to a function that sums the first n integers, given n as an argument.*

Exercise 19. *In the PL given from Figure 4, write an expression that evaluates to a function that multiplies m and n, given m and n as arguments (note that we can emulate subtraction by adding the negation).*

Exercise 20. *In the PL given from Figure 4, write an expression that evaluates to a function that determines the n^{th} Fibonacci number.*

Exercise 21. *Extend the type system for the domain-free STLC to the PL given in Figure 4.*

Syntax.

$$
\begin{aligned}
e &::= \lambda x.e \mid ee \mid x \mid n \mid e\ o\ e \\
&\mid \ \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{true} \mid \mathsf{false} \\
&\mid \ \mathsf{let}\ x = e\ \mathsf{in}\ e \\
o &::= \ = \ \mid \ +
\end{aligned}
$$

Values.

$$v = (\!|\ \mathcal{E}\ ,\ \lambda x.e\ |\!) \mid (\!|\ \mathcal{E}\ ,\ \lambda x.e\ ,\ x\ |\!) \mid n \mid \mathsf{true} \mid \mathsf{false}$$

Semantics.

$$\frac{}{\langle\ \mathcal{E}\ ,\ x\ \rangle \Downarrow \mathcal{E}(x)}\ \textsc{var} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ n\ \rangle \Downarrow n}\ \textsc{int} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \lambda x.e\ \rangle \Downarrow (\!|\ \mathcal{E}\ ,\ \lambda x.e\ |\!)}\ \textsc{fun}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow (\!|\ \mathcal{E}'\ ,\ \lambda x.e\ |\!) \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \qquad \langle\ \mathcal{E}'[x \mapsto v_2]\ ,\ e\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ e_1 e_2\ \rangle \Downarrow v}\ \textsc{app}_1$$

$$\frac{\begin{array}{c}\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow (\!|\ \mathcal{E}'\ ,\ \lambda x.e\ ,\ f\ |\!) \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2 \\ \langle\ \mathcal{E}'[f \mapsto (\!|\ \mathcal{E}'\ ,\ \lambda x.e\ ,\ f\ |\!)][x \mapsto v_2]\ ,\ e\ \rangle \Downarrow v\end{array}}{\langle\ \mathcal{E}\ ,\ e_1 e_2\ \rangle \Downarrow v}\ \textsc{app}_2$$

$$\frac{}{\langle\ \mathcal{E}\ ,\ \mathsf{true}\ \rangle \Downarrow \mathsf{true}}\ \textsc{true} \qquad \frac{}{\langle\ \mathcal{E}\ ,\ \mathsf{false}\ \rangle \Downarrow \mathsf{false}}\ \textsc{false}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathsf{true} \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \rangle \Downarrow v_2}\ \textsc{ifTrue}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow \mathsf{false} \qquad \langle\ \mathcal{E}\ ,\ e_3\ \rangle \Downarrow v_3}{\langle\ \mathcal{E}\ ,\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \rangle \Downarrow v_3}\ \textsc{ifFalse}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ +\ e_2\ \rangle \Downarrow v_1 + v_2}\ \textsc{add}$$

$$\frac{\langle\ \mathcal{E}\ ,\ e_1\ \rangle \Downarrow v_1 \qquad \langle\ \mathcal{E}\ ,\ e_2\ \rangle \Downarrow v_2}{\langle\ \mathcal{E}\ ,\ e_1\ =\ e_2\ \rangle \Downarrow v_1 = v_2}\ \textsc{eq}$$

$$\frac{\langle\ \mathcal{E}[x \mapsto (\!|\ \mathcal{E}\ ,\ \lambda x_1.e_1\ ,\ x\ |\!)]\ ,\ e_2\ \rangle \Downarrow v}{\langle\ \mathcal{E}\ ,\ \mathsf{let}\ x = \lambda x_1.e_1\ \mathsf{in}\ e_2\ \rangle \Downarrow v}\ \textsc{letRec}$$

Figure 4: The λ-calculus with recursive let-expressions, conditionals and integers

Exercise 22. *Demonstrate that well-typed expressions according to the system from the previous exercise are not necessarily normalizing, i.e., determine an expression e and type τ such that $\varnothing \vdash e : \tau$ but there is no value v such that $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$.*

Exercise 23. ***(Challenge)*** *Show that your type system from the previous exercise satisfies type safety, i.e., $\varnothing \vdash e : \tau$ and $\langle\ \varnothing\ ,\ e\ \rangle \Downarrow v$ implies $\varnothing \vdash v : \tau$. You can assume the substitution lemma without proving it.*

A little experiment. We come back to the example given at the top of the notes. We can build the expressions $\{e_n\}_{n \in \mathbb{N}}$ programmatically in OCaml.

```
type expr =
  | Lam of string * expr
  | App of expr * expr
  | Var of string

let app e1 e2 = App (e1, e2)
let lam e x = Lam (x, e)
let id = Lam ("x", Var "x")

let x n = "x" ^ string_of_int n
let f n = List.(fold_left lam (Var (x 0)) (init n x))
let e n = List.(fold_left app (f n) (init n (fun _ -> id)))
```

We can then look at two basic implementations of evaluation, one for the big-step CBV substitution semantics and one for the big-step CBV environment semantics (Figure 5). It's worth emphasizing how similar these implementations are, particularly just in terms of amount of code written. Likewise, it's worth noting how similar the implementations are to their corresponding formal specifications. Benchmarking these two implementations on evaluating `e n` yields the expected difference in running times.

n	`eval_sub` (ns)	`eval_env` (ns)
1	9.36	14.23
10	188.79	65.42
100	28,313.46	1,122.70
1,000	3,507,667.08	11,596.02
10,000	420,107,774.93	135,101.47

These numbers give a quadratic trend for `eval_sub` and a linear trend for `eval_env`. But really, it suffices to note that the difference becomes quite drastic as n increases.

```
let subst e' x =
let rec go = function
| Lam (y, e) -> Lam (y, if x = y then e else go e)
| App (e1, e2) -> App (go e1, go e2)
| Var y -> if x = y then e' else Var y
in go

let d_lam e =
match e with
| Lam(x, e) -> (x, e)
| _ -> assert false

let rec eval_sub = function
| Lam (y, e) -> Lam (y, e)
| App (e1, e2) ->
  let (x, e) = d_lam (eval_sub e1) in
  eval_sub (subst (eval_sub e2) x e)
| Var y -> assert false

module Env = Map.Make(String)

let rec eval_env env = function
| Lam (y, e) -> Lam (y, e)
| App (e1, e2) ->
  let (x, e) = d_lam (eval_env env e1) in
  eval_env (Env.add x (eval_env env e2) env) e
| Var y -> Env.find y env

let eval_env = eval_env Env.empty
```

Figure 5: Implementations of big-step CBV substitution semantics and big-step CBV environment semantics for the λ-calculus