CPSC 311: Environment-based semantics **(DRAFT)** (“lec-env”)

Joshua Dunfield
University of British Columbia

November 1, 2016

1 The trouble with substitution

We’ve defined dynamic semantics in two different ways: big-step and small-step. In both, we used substitution to define the meaning of expressions with identifiers (variables): for a Let, we evaluate its bound expression, then use the resulting value to replace all the instances of the bound identifier. Functions (Lam/App), recursion (Rec), Pair-case, and so on were also given meaning via substitution.

Our notion of substitution is directly descended from Alonzo Church’s λ-calculus, but as a general (and less precise) notion, substitution is older: in algebra we can substitute 5 for x in

$$x^2 + 3$$

to get $5^2 + 3$. (I don’t think the ancient syllogisms of Greece and India—“Socrates is a man, all men are mortal, therefore Socrates is mortal”; “This hill is smoky; whatever is smoky is fiery (for example: a kitchen); therefore this hill is fiery”[1]—are truly *substitution*: there are no variables.)

The connection to the λ-calculus, which is equivalent in power to Turing machines, guarantees that substitution is *a* “right way” of defining how features like Let and App work. It does not mean that substitution is *the* right way of defining how those features work. In fact, substitution is (almost?) never used to implement interpreters.

Substitution has several disadvantages, compared to other methods:

- *Inefficiency:* Every time our interpreter needs to compute $[e2/x]e1$, our implementation `subst` searches for (Id x) throughout the entire expression $e1$. It must do this even if $e1$ is very large, and (Id x) appears only once (or even not at all!).

  After substituting, we may need to apply further substitutions over the result. If $e1$ is small—say, (Add (Id x) (Id x))—but $e2$ is large, the resulting expression will also be large.

- *Obscurity:* Giving a function (or other expression) a name is important for clarity and convenience; we would rather write `{App double 5}` than `{App {Lam y {+ y y}} 5}`, even though they give the same result. But a substitution-based interpreter that prints the expressions it’s evaluating will show you the latter. This is perhaps most aggravating with recursive functions (Rec u (Lam ...).

Against these, we should weigh substitution’s advantages:

- *Simplicity:* The definition of substitution is more straightforward than other methods, and it is *self-contained*, making it portable across other changes in semantics (e.g. between big-step and small-step semantics).

[1] Adapted from Vidyabhusana, *A History of Indian Logic* (1920), p. 61.

- *Versatility:* While substitution doesn't "scale" in terms of performance (see "Inefficiency" above), it "scales up" well across a variety of language features. The same style of defining substitution that we used to add Let to our tiny language of arithmetic expressions also works for functions that return functions ("first-class" functions) and for recursive functions. Environments are more brittle: adding new features sometimes requires us to define environments in a way that is more complicated (rather than just being *longer*, as is the case with substitution).

Whether or not you prefer environments, you should learn about them, especially if you plan to take CPSC 411.

2 Environments

The idea of environment-based dynamic semantics is that, to evaluate $(\mathsf{Let}\ x\ e\ eBody)$, we won't evaluate e to v and then substitute v for x; instead, we will evaluate e to v, and "remember" that x has the value v. This fact will be stored in an *environment* that maps identifiers to values. If and when we need to evaluate an instance of x in $eBody$, that is, if we need to evaluate $(\mathsf{Id}\ x)$, we look up x and use the value we find, which will be v.

In a sense, we are simulating substitution: if we had substituted v for x, we would find v inside the body, at the places in $eBody$ that were originally $(\mathsf{Id}\ x)$.

Because environments are more brittle than substitution, I think it's better start with a small language (arithmetic expressions and Let), define the simplest possible environments, and then carefully evolve our notion of an environment as we restore language features.

2.1 Mapping identifiers to expressions

To get an idea of what is needed, consider the expression (in abstract syntax)

$$\big(\mathsf{Let}\ x\ (\mathsf{Num}\ 3)\ (\mathsf{Let}\ y\ (\mathsf{Num}\ 4)\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ y)))\big)$$

If we don't use substitution, when we evaluate $(\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ y))$ we need to remember that x was bound to $(\mathsf{Num}\ 3)$, and y was bound to $(\mathsf{Num}\ 4)$. We need a "lookup table" that maps identifiers to expressions.

In Typed Fun, we used a typing context Γ to map identifiers to types, and defined what those contexts were with a grammar:

$$\begin{array}{llll} \text{Typing contexts} & \Gamma ::= & \emptyset & \text{empty context (no assumptions)} \\ & & \mid x : A, \Gamma & x \text{ has type } A\text{, with more assumptions } \Gamma \end{array}$$

We'll do the same for environments:

$$\begin{array}{llll} \text{Environments} & env ::= & \emptyset & \text{empty environment} \\ & & \mid x{=}e, env & x \text{ bound to } e\text{, with “more environment” } env \end{array}$$

(It would be more standard to use the Greek letter rho (ρ), rather than "env", but we've used enough Greek letters for now.)

For consistency with typing contexts Γ, environments env will grow to the left, like `cons`-lists in Racket.

Let's consider an even smaller example than the one above.

$$(\mathsf{Let}\ y\ (\mathsf{Num}\ 4)\ (\mathsf{Add}\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ y)))$$

If this expression is the entire program, it's not inside any Lets, so the environment env is empty when we start evaluating it.

Regardless of how environments work, $(\mathsf{Num}\ 4)$ should still evaluate to $(\mathsf{Num}\ 4)$. But now we need to remember that y is $(\mathsf{Num}\ 4)$, so we'll need to evaluate the body $(\mathsf{Add}\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ y))$ under the environment

$$y{=}(\mathsf{Num}\ 4),\ \emptyset$$

(It's okay to write this as just $y{=}(\mathsf{Num}\ 4)$; here, I want to emphasize that we started with $\emptyset$, and are growing the environment leftwards.)

Then, while evaluating $(\mathsf{Id}\ y)$ in $(\mathsf{Add}\ (\mathsf{Num}\ 3)\ (\mathsf{Id}\ y))$, we will look up $(\mathsf{Id}\ y)$ in the current environment $y{=}(\mathsf{Num}\ 4),\ \emptyset$, and evaluation will behave as if we were evaluating $(\mathsf{Add}\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 4))$.

Just as we used Γ in the typing judgment $\Gamma \vdash e : A$, we'll use env in a new *environment-based evaluation* judgment form

$$env \vdash e \Downarrow v$$

We'll also assume that a "lookup function" *lookup*(env, x) has been defined, so that

$$lookup(env, x) = e$$

if the environment env contains $x{=}e$. (In our Racket code, we have a function `look-up-id`.)

I think we have enough to revise the evaluation rules. What were those?

$\boxed{e \Downarrow v}$ Expression e evaluates to value v (substitution-based)

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num}$$

$$\frac{e1 \Downarrow (\mathsf{Num}\ n1) \qquad e2 \Downarrow (\mathsf{Num}\ n2)}{(\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 + n2)}\ \text{Eval-add}$$

$$\frac{e1 \Downarrow (\mathsf{Num}\ n1) \qquad e2 \Downarrow (\mathsf{Num}\ n2)}{(\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 - n2)}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-let}$$

$$\frac{}{(\mathsf{Id}\ x)\ \text{free-variable-error}}\ \text{Eval-free-identifier}$$

$\boxed{env \vdash e \Downarrow v}$ Under environment env, expression e evaluates to value v

$$\frac{}{env \vdash (\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Env-num}$$

$$\frac{e1 \Downarrow (\mathsf{Num}\ n1) \qquad e2 \Downarrow (\mathsf{Num}\ n2)}{(\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 + n2)}\ \text{Env-add}$$

$$\frac{e1 \Downarrow (\mathsf{Num}\ n1) \qquad e2 \Downarrow (\mathsf{Num}\ n2)}{(\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 - n2)}\ \text{Env-sub}$$

$$\frac{e1 \Downarrow v1 \qquad \Downarrow v2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Env-let}$$

$$\frac{}{(\mathsf{Id}\ x) \Downarrow}\ \text{Env-id}$$

$$\frac{}{\text{unknown-id-error}}\ \text{Env-unknown-id}$$

Exercise 1. I left some blank space in the "Env-..." rules. Fill it in with whatever is needed. Env-num is finished, and you can follow that pattern for some of the other rules.

The completed rules are on the next page.

$\boxed{e \Downarrow v}$ Expression e evaluates to value v (substitution-based)

$$\frac{}{(\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Eval-num}$$

$$\frac{e1 \Downarrow (\mathsf{Num}\ n1) \qquad e2 \Downarrow (\mathsf{Num}\ n2)}{(\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 + n2)}\ \text{Eval-add}$$

$$\frac{e1 \Downarrow (\mathsf{Num}\ n1) \qquad e2 \Downarrow (\mathsf{Num}\ n2)}{(\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 - n2)}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad [v1/x]e2 \Downarrow v2}{(\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-let}$$

$$\frac{}{(\mathsf{Id}\ x)\ \mathsf{free\text{-}variable\text{-}error}}\ \text{Eval-free-identifier}$$

$\boxed{env \vdash e \Downarrow v}$ Under environment env, expression e evaluates to value v

$$\frac{}{env \vdash (\mathsf{Num}\ n) \Downarrow (\mathsf{Num}\ n)}\ \text{Env-num}$$

$$\frac{env \vdash e1 \Downarrow (\mathsf{Num}\ n1) \qquad env \vdash e2 \Downarrow (\mathsf{Num}\ n2)}{env \vdash (\mathsf{Add}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 + n2)}\ \text{Env-add}$$

$$\frac{env \vdash e1 \Downarrow (\mathsf{Num}\ n1) \qquad env \vdash e2 \Downarrow (\mathsf{Num}\ n2)}{env \vdash (\mathsf{Sub}\ e1\ e2) \Downarrow (\mathsf{Num}\ n1 - n2)}\ \text{Env-sub}$$

$$\frac{env \vdash e1 \Downarrow v1 \qquad x{=}v1, env \vdash e2 \Downarrow v2}{env \vdash (\mathsf{Let}\ x\ e1\ e2) \Downarrow v2}\ \text{Env-let}$$

$$\frac{lookup(env, x) = e}{env \vdash (\mathsf{Id}\ x) \Downarrow e}\ \text{Env-id}$$

$$\frac{lookup(env, x)\ \text{undefined}}{env \vdash (\mathsf{Id}\ x)\ \mathsf{unknown\text{-}id\text{-}error}}\ \text{Env-unknown-id}$$

2.2 The Shadow Chancellor Strikes Back

(At some point, the UK Parliament becomes indistinguishable from a bad fantasy novel.)

With substitution, we saw that expressions that repeatedly bind the same identifier are evaluated with the inner binding "shadowing" the outer one, so that

$$(\mathsf{Let}\ x\ (\mathsf{Num}\ 1)\ (\mathsf{Let}\ x\ (\mathsf{Num}\ 2)\ (\mathsf{Id}\ x)))$$

evaluates to $(\mathsf{Num}\ 2)$, not $(\mathsf{Num}\ 1)$. The environment-based semantics will behave the same way, but only because of a particular way we're defining *lookup*: it starts looking from the left.

$$\frac{\dfrac{}{\emptyset \vdash (\mathsf{Num}\ 1) \Downarrow (\mathsf{Num}\ 1)}\ \text{Env-num} \qquad \dfrac{\dfrac{}{x{=}(\mathsf{Num}\ 1), \emptyset \vdash (\mathsf{Num}\ 2) \Downarrow (\mathsf{Num}\ 2)}\ \text{Env-num} \qquad \dfrac{lookup\big((x{=}(\mathsf{Num}\ 2), x{=}(\mathsf{Num}\ 1), \emptyset), x\big) = (\mathsf{Num}\ 2)}{x{=}(\mathsf{Num}\ 2), x{=}(\mathsf{Num}\ 1), \emptyset \vdash (\mathsf{Id}\ x) \Downarrow (\mathsf{Num}\ 2)}\ \text{Env-id}}{x{=}(\mathsf{Num}\ 1), \emptyset \vdash (\mathsf{Let}\ x\ (\mathsf{Num}\ 2)\ (\mathsf{Id}\ x)) \Downarrow (\mathsf{Num}\ 2)}\ \text{Env-let}}{\emptyset \vdash \big(\mathsf{Let}\ x\ (\mathsf{Num}\ 1)\ (\mathsf{Let}\ x\ (\mathsf{Num}\ 2)\ (\mathsf{Id}\ x))\big) \Downarrow (\mathsf{Num}\ 2)}\ \text{Env-let}$$

We should really define *lookup* using rules; I'll leave that as an exercise (next page).

■ **Exercise 2.** Fill in the rules below, which derive a judgment $lookup(env, x) = e$:
(Feel free to translate "backwards" from the Racket implementation of `look-up-id`.)

$$\frac{}{\mathit{lookup}(\emptyset, x)\ \_\_\_\_\_\_\_\_\_}\ \text{lookup-empty}$$

$$\frac{}{\mathit{lookup}\big((x{=}e, \mathit{env}), x\big)\ \_\_\_\_\_\_\_\_\_}\ \text{lookup-found} \qquad \frac{}{\mathit{lookup}\big((y{=}e, \mathit{env}), x\big)\ \_\_\_\_\_\_\_\_\_}\ \text{lookup-next}$$

2.3 Question Period

Question:
The expression after the "⇓" should always be a value. Shouldn't Env-id evaluate e to v?

It could, but it doesn't need to: the expressions we put into environments are all values. The only rule that adds anything to the environment is Env-let, and the expression it adds is $v1$, which is a value.

Question: Could Env-let *not* evaluate $e1$, and put $e1$ into the environment, instead?

In that case, Env-id *would* need to evaluate the expression it gets from *lookup*. That would give us an "expression strategy" for Let. That's inconsistent with our substitution-based semantics, but it's not wrong; it's just not what I want to do.

Question: What if Env-let puts $e1$ into the environment, and Env-id evaluates that expression to get $v1$, and then *updates the environment* with $v1$? Would that gives us lazy evaluation?

You could certainly implement that—for example, using Racket's mutable "boxes". Moreover, we could model it using rules. But the rules would need to be rather different from the above rules, which derive the judgment form $\mathit{env} \vdash e \Downarrow v$. That judgment can't model a change to the environment; the above rules can only add to the environment *inside* a premise. So if your environment is

$$\underbrace{x{=}(\mathsf{Add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 1)), \emptyset}_{\mathit{env}}$$

and you evaluate $(\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x))$, you can't "transmit" the updated env from the first premise to the second premise. Rules and derivations aren't mutable.

$$\frac{\mathit{env} \vdash (\mathsf{Id}\ x) \Downarrow (\mathsf{Num}\ 2) \qquad \mathit{env} \vdash (\mathsf{Id}\ x) \Downarrow (\mathsf{Num}\ 2)}{\underbrace{x{=}(\mathsf{Add}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 1)), \emptyset}_{\mathit{env}} \vdash (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{Id}\ x)) \Downarrow (\mathsf{Num}\ 4)}\ \text{Env-add}$$

However, you could change the judgment form to something like

$$\mathit{env} \vdash e \Downarrow v, \mathit{env}'$$

which could be read "starting in environment env, evaluating expression e produces value v and an environment env'." Then the conclusion of the rule for Id could have the "updated" environment as env'.

We'll need to do something like this to model mutable state (hopefully, next week).