

CPSC 311: Environment-based semantics: Closures (DRAFT) ("lec-closures")

Joshua Dunfield

Alec Thériault

Khurram Jafery

University of British Columbia

November 9, 2016

1 Attack of the Dynamic Scope

Fun is broken! (Typed Fun is *not* broken, though! Types win again?)

Scoping in Fun was supposed to be *lexical*, where an instance of an identifier refers to the nearest enclosing binding occurrence. Thus, in

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

the instance (Id x) refers to the inner binding occurrence (and therefore to (Num 2)).

Before we added Lam and App, scoping actually *was* properly lexical: attempting to evaluate an identifier that has *no* enclosing binder, as in

$$(\text{Let } x \text{ (Id } y) \text{ (Let } x \text{ (Num 2) (Id } x)))$$

would cause a free variable error.

However, scoping in Fun was partly lexical, and partly *dynamic*:

$$\left(\text{Let } f \text{ (Lam } y \text{ (Id } z)) \text{ (Let } z \text{ (Num 2) (App (Id } f) \text{ (Num 0)))} \right)$$

The rule Eval-let substitutes (Lam y (Id z)) for f , and then evaluates

$$(\text{Let } z \text{ (Num 2) (App (Lam } y \text{ (Id } z)) \text{ (Num 0)))}$$

in which (Num 2) is substituted for z :

$$(\text{App (Lam } y \text{ (Num 2)) (Num 0)})$$

which evaluates to (Num 2). Observe that (Lam y (Id z)) doesn't look at its argument y , which in any case is substituted with (Num 0), which is not (Num 2). In PL jargon, we say that the identifier (Id z) in the body of (Lam y (Id z))—which really shouldn't refer to *anything* and should be an error—has been *captured* by the binding (Let z (Num 2) ...).

1.1 A Brief History of Infamy

The story goes that dynamic scope—in which the “most recent” binding is used, rather than the lexically *enclosing* binding—was invented, by accident, in Lisp. Subsequent versions of Lisp corrected this, except for Emacs Lisp (which most of Emacs is written in).

In Fun, we implemented something that “is” lexical scoping, in the sense that an identifier in any correctly lexically-scoped expression will refer to its nearest lexically-enclosing binder. But we also implemented a little dynamic scoping: in an expression with a Lam, we allow free identifiers inside the body, which can then be captured by later bindings.

We corrected this (unknowingly) in Typed Fun: When `typeof` sees an identifier, it checks that the identifier appears in the typing context τc (written Γ in the rules).

The fix in Fun itself is—I’m pretty sure—to add a check in `subst` that makes sure the expression being substituted has no free identifiers (*not* the expression being substituted *into*, which probably does have a free identifier: the identifier being substituted!).

For example, in the example with `f` and `(Lam y (ld z))` above, that check would find the free identifier `(ld z)`, and raise an error.

2 Functions in environment-based semantics

The eruption of dynamic scoping is relevant, however, because another way to accidentally get dynamic scoping is to add functions to an environment semantics. So let's do that, and then fix it.

$e \Downarrow v$ Fun expression e evaluates to value v

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

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

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

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

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

$$\frac{}{(\text{Lam } x \ e1) \Downarrow (\text{Lam } x \ e1)} \text{Eval-lam}$$

$$\frac{e1 \Downarrow (\text{Lam } x \ eB) \quad e2 \Downarrow v2 \quad [v2/x]eB \Downarrow v}{(\text{App } e1 \ e2) \Downarrow v} \text{Eval-app-value}$$

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

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

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

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

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

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

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

$$\frac{}{\text{env} \vdash (\text{Lam } x \ e1) \Downarrow (\text{Lam } x \ e1)} \text{**Env-lam-dynamic}$$

$$\frac{\text{env} \vdash e1 \Downarrow (\text{Lam } x \ eB) \quad \text{env} \vdash e2 \Downarrow v2 \quad x=v2, \text{env} \vdash eB \Downarrow v}{\text{env} \vdash (\text{App } e1 \ e2) \Downarrow v} \text{**Env-app-dynamic}$$

§2 Functions in environment-based semantics

2.1 Boom! Lambda

(In honour of the failed renaming of Pie R Squared.)

The above rules, which seem reasonable—**Env-app-dynamic follows the pattern of Env-let—cause even more dynamic scoping than my oversight in substitution-based Fun.

Consider the expression

$$\left(\text{Let } y \text{ (Num 1)} \left(\text{Let } f \text{ (Lam } x \text{ (Id } y))} \left(\text{Let } y \text{ (Num 2)} \left(\text{App (Id } f) \text{ (Num 0)} \right) \right) \right) \right)$$

In a substitution-based semantics, the first thing we do is substitute (Num 1) for y :

$$\left(\text{Let } f \text{ (Lam } x \text{ (Num 1))} \left(\text{Let } y \text{ (Num 2)} \left(\text{App (Id } f) \text{ (Num 0)} \right) \right) \right)$$

This means that f (will be substituted with) a constant function that always returns (Num 1).

However, with the above **-rules, we add $y=(\text{Num 1})$ to the empty environment, then $f=(\text{Lam } x \text{ (Id } y))$, and then $y=(\text{Num 2})$. Since *lookup* looks at the environment starting from the left, looking up an instance of (Id y) will result in (Num 2):

$$\frac{\text{env}_{yfy} \vdash (\text{Id } f) \Downarrow (\text{Lam } x \text{ (Id } y)) \quad \text{env}_{yfy} \vdash (\text{Num 0}) \Downarrow (\text{Num 0}) \quad x=(\text{Num 0}), \text{env}_{yfy} \vdash (\text{Id } y) \Downarrow (\text{Num 2})}{\underbrace{y=(\text{Num 2}), f=(\text{Lam } x \text{ (Id } y)), y=(\text{Num 1}), \emptyset \vdash (\text{App (Id } f) \text{ (Num 0)}) \Downarrow (\text{Num 2})}_{\text{env}_{yfy}}} \text{**Env-app-dynamic}$$

The problem is that when $f=(\text{Lam } x \text{ (Id } y))$ was added to the environment, looking up (Id y) would have given (Num 1), since that is the nearest enclosing binding. But instead, we used a binding that was nowhere in scope when f was bound.

Under lexical scoping, you can always determine where an identifier’s binder is without “looking into the future”: if a nested Let that comes later happens to shadow an identifier, it won’t matter. Under dynamic scoping, which we have now re-created, this isn’t the case.

■ **Question:** Can you show the rest of the derivation?

$$\frac{\frac{\frac{\text{env}_{fy} \vdash \Downarrow (\text{Num 2})}{(\text{Lam } x \text{ (Id } y))} \quad \frac{\text{env}_{yfy} \vdash \Downarrow (\text{Num 2}) \quad \underbrace{y=(\text{Num 2}), f=(\text{Lam } x \text{ (Id } y)), y=(\text{Num 1}), \emptyset \vdash (\text{App (Id } f) \text{ (Num 0)}) \Downarrow (\text{Num 2})}_{\text{env}_{yfy}} \quad \checkmark \text{ see derivation above}}{f=(\text{Lam } x \text{ (Id } y)), y=(\text{Num 1}), \emptyset \vdash (\text{Let } y \text{ (Num 2)} \text{ (App } \dots)) \Downarrow (\text{Num 2})}_{\text{env}_{fy}}}{y=(\text{Num 1}), \emptyset \vdash \Downarrow (\text{Lam } x \text{ (Id } y))} \text{Env-let}}{y=(\text{Num 1}), \emptyset \vdash (\text{Let } f \text{ (Lam } x \text{ (Id } y))} \text{ (Let } y \text{ } \dots)) \Downarrow (\text{Num 2})} \text{Env-let}$$

2.2 Closures

The solution is to remember something about the environment that existed *when the binding happened*, that is, when the Lam was evaluated.

The easiest way to remember something about the environment is to remember the entire environment, so that’s what we’ll do. The “pairing up” of a Lam with its environment is called a *closure*.

Closures are a new kind of animal; where do they live? For uniformity, it will be easiest (I think) to think of them as expressions. Alternatively, we could make them a new kind of thing in the environment, so that we’d have ordinary value bindings in the environment, and also closure bindings.

This leads to the following **define-type**:

§2 Functions in environment-based semantics

```
(define-type E
  [Num (n number?)]
  [Add (lhs E?) (rhs E?)]
  [Sub (lhs E?) (rhs E?)]
  [Let (name symbol?) (named-expr E?) (body E?)]
  [Id (name symbol?)]
  [Lam (name symbol?) (body E?)]
  [App (function E?) (argument E?)]
  [Clo (env Env?) (e E?)]           ; not in concrete syntax
)
```

We’ve already seen a **define-type** in which a variant didn’t correspond to a single production of the BNF: `Binop`. There, however, the `Binop` variants were generated inside the parser. Here, the parser will never generate a closure `Clo`. Instead, closures will be generated only inside the interpreter `env-interp`.

$$\frac{}{\text{env} \vdash (\text{Lam } x \ e1) \Downarrow (\text{Clo } \text{env} \ (\text{Lam } x \ e1))} \text{Env-lam} \quad \frac{}{\text{env} \vdash (\text{Clo } \text{env}_{\text{old}} \ e) \Downarrow (\text{Clo } \text{env}_{\text{old}} \ e)} \text{Env-clo}$$

$$\frac{\text{env} \vdash e1 \Downarrow (\text{Clo } \text{env}_{\text{old}} \ (\text{Lam } x \ eB)) \quad \text{env} \vdash e2 \Downarrow v2 \quad x=v2, \text{env}_{\text{old}} \vdash eB \Downarrow v}{\text{env} \vdash (\text{App } e1 \ e2) \Downarrow v} \text{Env-app}$$

■ **Question:** What happens if the `Lam` has a free variable that isn’t in the environment `envold`, but is in the newer environment we have when we evaluate `App`? If we need a free-variable check in *subst* for substitution-based semantics, do we also need one for environment-based semantics?

If the `Lam` tries to use an identifier that isn’t in the environment when the `Lam` was evaluated, this will be an error. The error won’t happen until the `Lam`—which is now a `(Clo envold (Lam ...))`—is applied, but it will be a proper error; the identifier will not be captured by a later binding, because that binding won’t be in `envold`.

If our language is typed, this doesn’t matter, because the type checker will catch this error statically.

In practice, particularly in a compiler, we only store the actual free identifiers of the `Lam` in the closure—not the entire environment. After parsing, we can figure out what the free identifiers of the `Lam` are, so we’ll know which bindings from `envold` to save.

3 Recursive closures

At this point, we could add more features (pairs, trees, sums, etc.) without any trouble. Instead, let's try to add the feature that *will* give us trouble: Rec.

Environments $env ::= \emptyset$
 $\quad \quad \quad | x=e, env$
 $\quad \quad \quad | \mathcal{E} \quad \quad \quad \text{environment variable}$
 $\quad \quad \quad | (\text{Env-Rec } \mathcal{E} \ env) \quad \text{recursive environment}$

Definition of environment substitution over environments:

$$\begin{aligned} [\text{env}_1/\mathcal{E}_1]\emptyset &= \emptyset \\ [\text{env}_1/\mathcal{E}_1](x=e, \text{env}_2) &= x=[\text{env}_1/\mathcal{E}_1]e, [\text{env}_1/\mathcal{E}_1]\text{env}_2 \\ [\text{env}_1/\mathcal{E}_1]\mathcal{E}_1 &= \text{env}_1 \\ [\text{env}_1/\mathcal{E}_1]\mathcal{E}_2 &= \mathcal{E}_2 \quad \text{if } \mathcal{E}_1 \neq \mathcal{E}_2 \\ [\text{env}_1/\mathcal{E}_1](\text{Env-Rec } \mathcal{E}_1 \ \text{env}_2) &= (\text{Env-Rec } \mathcal{E}_1 \ \text{env}_2) \\ [\text{env}_1/\mathcal{E}_1](\text{Env-Rec } \mathcal{E}_2 \ \text{env}_2) &= (\text{Env-Rec } \mathcal{E}_2 \ [\text{env}_1/\mathcal{E}_1]\text{env}_2) \quad \text{if } \mathcal{E}_1 \neq \mathcal{E}_2 \end{aligned}$$

Definition of environment substitution over expressions:

$$\begin{aligned} [\text{env}/\mathcal{E}](\text{Num } n) &= (\text{Num } n) \\ [\text{env}/\mathcal{E}](\text{Binop } op \ e1 \ e2) &= (\text{Binop } op \ [\text{env}/\mathcal{E}]e1 \ [\text{env}/\mathcal{E}]e2) \\ [\text{env}/\mathcal{E}](\text{Btrue}) &= (\text{Btrue}) \\ [\text{env}/\mathcal{E}](\text{Bfalse}) &= (\text{Bfalse}) \\ [\text{env}/\mathcal{E}](\text{Ite } e \ e\text{Then } e\text{Else}) &= (\text{Ite } [\text{env}/\mathcal{E}]e \ [\text{env}/\mathcal{E}]e\text{Then } [\text{env}/\mathcal{E}]e\text{Else}) \\ [\text{env}/\mathcal{E}](\text{Id } x) &= (\text{Id } x) \\ [\text{env}/\mathcal{E}](\text{Let } x \ e1 \ e2) &= (\text{Let } x \ [\text{env}/\mathcal{E}]e1 \ [\text{env}/\mathcal{E}]e2) \\ [\text{env}/\mathcal{E}](\text{Lam } x \ eB) &= (\text{Lam } x \ [\text{env}/\mathcal{E}]eB) \\ [\text{env}/\mathcal{E}](\text{App } e1 \ e2) &= (\text{App } [\text{env}/\mathcal{E}]e1 \ [\text{env}/\mathcal{E}]e2) \\ [\text{env}/\mathcal{E}](\text{Rec } u \ eB) &= (\text{Rec } u \ [\text{env}/\mathcal{E}]eB) \\ [\text{env}_1/\mathcal{E}](\text{Clo } \text{env}_2 \ e) &= (\text{Clo } [\text{env}_1/\mathcal{E}]\text{env}_2 \ [\text{env}_1/\mathcal{E}]e) \end{aligned}$$

Definition of environment lookup:

$$\begin{aligned} \text{lookup}((x = e, \text{env}), x) &= e \\ \text{lookup}((y = e, \text{env}), x) &= \text{lookup}(\text{env}, x) \quad \text{if } x \neq y \\ \text{lookup}(\text{Env-Rec } \mathcal{E} \ \text{env}, x) &= \text{lookup}([\text{Env-Rec } \mathcal{E} \ \text{env}]/\mathcal{E} \ \text{env}, x) \end{aligned}$$

§3 Recursive closures

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

$$\frac{}{\text{env} \vdash (\text{Num } n) \Downarrow (\text{Num } n)} \text{Env-num} \quad \frac{\text{env} \vdash e1 \Downarrow v1 \quad \text{env} \vdash e2 \Downarrow v2 \quad v1 \text{ op } v2 = v}{\text{env} \vdash (\text{Binop op } e1 \ e2) \Downarrow v} \text{Env-binop}$$

$$\frac{}{\text{env} \vdash (\text{Btrue}) \Downarrow (\text{Btrue})} \text{Env-true} \quad \frac{}{\text{env} \vdash (\text{Bfalse}) \Downarrow (\text{Bfalse})} \text{Env-false}$$

$$\frac{\text{env} \vdash e \Downarrow (\text{Btrue}) \quad \text{env} \vdash e\text{Then} \Downarrow v}{\text{env} \vdash (\text{Ite } e \ e\text{Then} \ e\text{Else}) \Downarrow v} \text{Env-ite-true}$$

$$\frac{\text{env} \vdash e \Downarrow (\text{Bfalse}) \quad \text{env} \vdash e\text{Else} \Downarrow v}{\text{env} \vdash (\text{Ite } e \ e\text{Then} \ e\text{Else}) \Downarrow v} \text{Env-ite-false}$$

$$\frac{\text{lookup}(\text{env}, x) = e \quad \text{env} \vdash e \Downarrow v}{\text{env} \vdash (\text{Id } x) \Downarrow v} \text{Env-id} \quad \frac{\text{env} \vdash e1 \Downarrow v1 \quad x=v1, \text{env} \vdash e2 \Downarrow v2}{\text{env} \vdash (\text{Let } x \ e1 \ e2) \Downarrow v2} \text{Env-let}$$

$$\frac{}{\text{env} \vdash (\text{Lam } x \ eB) \Downarrow (\text{Clo env } (\text{Lam } x \ eB))} \text{Env-lam} \quad \frac{\text{env}_{\text{old}} \vdash e \Downarrow v}{\text{env} \vdash (\text{Clo env}_{\text{old}} \ e) \Downarrow v} \text{Env-clo}$$

$$\frac{\text{env} \vdash e1 \Downarrow (\text{Clo env}_{\text{old}} (\text{Lam } x \ eB)) \quad \text{env} \vdash e2 \Downarrow v2 \quad x=v2, \text{env}_{\text{old}} \vdash eB \Downarrow v}{\text{env} \vdash (\text{App } e1 \ e2) \Downarrow v} \text{Env-app-value}$$

$$\frac{(\text{Env-Rec } \mathcal{E} \ (u=(\text{Clo } \mathcal{E} \ eB), \text{env})) \vdash eB \Downarrow v \quad \mathcal{E} \notin \text{env}}{\text{env} \vdash (\text{Rec } u \ eB) \Downarrow v} \text{Env-rec}$$

Example

$$\frac{\frac{\text{lookup}(\text{envclo}, u) = \text{eLookup} \quad \text{envclo} \vdash \text{envclo} \Downarrow}{\text{envclo} \vdash (\text{Id } u) \Downarrow} \text{Env-id}}{\frac{\text{lookup}(\text{envrec}, u) = \text{eLookup} \quad \text{envrec} \vdash (\text{Clo } \text{envclo } (\text{Id } u)) \Downarrow}{(\text{Env-Rec } \mathcal{E} (u=(\text{Clo } \mathcal{E} (\text{Id } u)), \emptyset)) \vdash (\text{Id } u) \Downarrow} \text{Env-id}}{\emptyset \vdash (\text{Rec } u (\text{Id } u)) \Downarrow} \text{Env-rec}$$

$$\text{envrec} = (\text{Env-Rec } \mathcal{E} (u=(\text{Clo } \mathcal{E} (\text{Id } u)), \emptyset))$$

$$\begin{aligned}
 \text{lookup}(\text{envrec}, u) &= \text{lookup} \left((\text{Env-Rec } \mathcal{E} (u=(\text{Clo } \mathcal{E} (\text{Id } u)), \emptyset)), u \right) \\
 &= \text{lookup} \left(\left[(\text{Env-Rec } \mathcal{E} (u=(\text{Clo } \mathcal{E} (\text{Id } u)), \emptyset)) / \mathcal{E} \right] (u=(\text{Clo } \mathcal{E} (\text{Id } u)), \emptyset), u \right) \\
 &= \text{lookup} \left((u=(\text{Clo } \text{envclo} (\text{Id } u)), \emptyset), u \right) \\
 &= \text{lookup} \left((u=(\text{Clo } (\text{Env-Rec } \mathcal{E} (u=(\text{Clo } \mathcal{E} (\text{Id } u)), \emptyset)) (\text{Id } u)), \emptyset), u \right) \\
 &= \text{lookup} \left((u=(\text{Clo } (\text{Env-Rec } \mathcal{E} (u=(\text{Clo } \mathcal{E} (\text{Id } u)), \emptyset)) (\text{Id } u)), \emptyset), u \right) \\
 &= \text{lookup} \left((\text{Clo } \underbrace{(\text{Env-Rec } \mathcal{E} (u=(\text{Clo } \mathcal{E} (\text{Id } u)), \emptyset))}_{\text{envclo}} (\text{Id } u)) \right) \\
 &= \text{eLookup}
 \end{aligned}$$