CPSC 311: Definition of Programming Languages: Environments: Closures

19-closures

Joshua Dunfield
University of British Columbia

November 5, 2015

1 Attack of the Dynamic Scope

On the way to Monday's tutorials, *someone* noticed that Fun is broken! (Typed Fun is *not* broken, though! Types win again?)

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

$$(\mathsf{with}\ x\ (\mathsf{num}\ 1)\ (\mathsf{with}\ x\ (\mathsf{num}\ 2)\ (\mathsf{id}\ x)))$$

the instance $(\mathsf{id}\ x)$ refers to the inner binding occurrence (and therefore to $(\mathsf{num}\ 2)$).

Scoping in WAE actually *was* properly lexical: attempting to evaluate an identifier that has *no* enclosing binder, as in

$$(\mathsf{with}\ x\ (\mathsf{id}\ y)\ (\mathsf{with}\ x\ (\mathsf{num}\ 2)\ (\mathsf{id}\ x)))$$

would cause a free variable error.

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

$$\Big(\mathsf{with}\ f\ \big(\mathsf{lam}\ y\ (\mathsf{id}\ z)\big)\ \big(\mathsf{with}\ z\ (\mathsf{num}\ 2)\ (\mathsf{app}\ (\mathsf{id}\ f)\ (\mathsf{num}\ 0))\big)\Big)$$

The rule Eval-with substitutes $(\mathsf{lam}\ y\ (\mathsf{id}\ z))$ for f, and then evaluates

$$\big(\mathsf{with}\ z\ (\mathsf{num}\ 2)\ (\mathsf{app}\ (\mathsf{lam}\ y\ (\mathsf{id}\ z))\ (\mathsf{num}\ 0))\big)$$

in which $(\mathsf{num}\ 2)$ is substituted for z:

$$(\mathsf{app}\ (\mathsf{lam}\ y\ (\mathsf{num}\ 2))\ (\mathsf{num}\ 0))$$

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

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 `tc` (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 (id z)) above, that check would find the free identifier (id z), and raise an error.

(I *really* wish I'd noticed this earlier, because that check would become one of the checks you could eliminate in Assignment 3, Problem 4!)

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.

$\boxed{e \Downarrow v}$ Fun expression e evaluates to value v

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

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

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

$$\frac{e1 \Downarrow v1 \qquad \mathit{subst}(e2, x, v1) \Downarrow v2}{(\mathsf{with}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-with}$$

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

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

$$\frac{e1 \Downarrow (\mathsf{lam}\ x\ eB) \quad e2 \Downarrow v2 \quad \mathit{subst}(eB, x, v2) \Downarrow v}{(\mathsf{app}\ e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

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

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

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

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

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

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

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

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

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

2.1 Boom! Lambda[1]

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

Consider the expression

$$\Big(\text{with } y\ (\text{num } 1)\ \big(\text{with } f\ (\text{lam } x\ (\text{id } y))\ \big(\text{with } y\ (\text{num } 2)\ (\text{app } (\text{id } f)\ (\text{num } 0))\big)\big)\Big)$$

In a substitution-based semantics, the first thing we do is substitute $(\text{num } 1)$ for y:

$$\big(\text{with } f\ (\text{lam } x\ (\text{num } 1))\ \big(\text{with } y\ (\text{num } 2)\ (\text{app } (\text{id } f)\ (\text{num } 0))\big)\big)$$

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 $(\text{id } y)$ will result in $(\text{num } 2)$:

$$\frac{env_{yfy} \vdash (\text{id } f) \Downarrow (\text{lam } x\ (\text{id } y)) \qquad env_{yfy} \vdash (\text{num } 0) \Downarrow (\text{num } 0) \qquad x{=}(\text{num } 0), env_{yfy} \vdash (\text{id } y) \Downarrow (\text{num } 2)}{\underbrace{y{=}(\text{num } 2),\ f{=}(\text{lam } x\ (\text{id } y)),\ y{=}(\text{num } 1),\ \emptyset}_{env_{yfy}} \vdash (\text{app } (\text{id } f)\ (\text{num } 0)) \Downarrow (\text{num } 2)}\ \text{**Env-app-dynamic}$$

The problem is that when $f{=}(\text{lam } x\ (\text{id } y))$ was added to the environment, looking up $(\text{id } y)$ would have given $(\text{num } 1)$, 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 with 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?

$$\frac{\dfrac{}{\begin{array}{r}(\text{lam } x\ (\text{id } y))\\ y{=}(\text{num } 1), \emptyset \vdash \Downarrow (\text{lam } x\ (\text{id } y))\end{array}} \qquad \dfrac{\dfrac{}{\begin{array}{r}(\text{num } 2)\\ env_{fy} \vdash \Downarrow (\text{num } 2)\end{array}} \qquad \dfrac{\checkmark \quad \textit{see derivation above}}{\overbrace{y{=}(\text{num } 2),\ f{=}(\text{lam } x\ (\text{id } y)),\ y{=}(\text{num } 1),\ \emptyset}^{env_{yfy}} \vdash (\text{app } (\text{id } f)\ (\text{num } 0)) \Downarrow (\text{num } 2)}}{\underbrace{f{=}(\text{lam } x\ (\text{id } y)),\ y{=}(\text{num } 1),\ \emptyset}_{env_{fy}} \vdash \big(\text{with } y\ (\text{num } 2)\ (\text{app } \cdots)\big) \Downarrow (\text{num } 2)}\ \text{Env-with}}{y{=}(\text{num } 1), \emptyset \vdash \big(\text{with } f\ (\text{lam } x\ (\text{id } y))\ \big(\text{with } y\ \cdots\big)\big) \Downarrow (\text{num } 2)}\ \text{Env-with}$$

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*.

[1] In honour of the failed renaming of Pie R Squared.

§2 Functions in environment-based semantics

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**:

```
(define-type E
  [num (n number?)]
  [add (lhs E?) (rhs E?)]
  [sub (lhs E?) (rhs E?)]
  [with (name symbol?) (named-expr E?) (body E?)]
  [id (name symbol?)]
  [lam (name symbol?) (body E?)]
  [clo (env Env?) (e E?)]              ; not in concrete syntax
  [app (function E?) (argument E?)]
  )
```

We've already seen a variant of **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`.

$$\frac{}{env \vdash (\mathsf{lam}\ x\ e1) \Downarrow \big(\mathsf{clo}\ env\ (\mathsf{lam}\ x\ e1)\big)}\ \text{Env-lam} \qquad \frac{}{env \vdash (\mathsf{clo}\ env_{\mathsf{old}}\ e) \Downarrow (\mathsf{clo}\ env_{\mathsf{old}}\ e)}\ \text{Env-clo}$$

$$\frac{env \vdash e1 \Downarrow (\mathsf{clo}\ env_{\mathsf{old}}\ (\mathsf{lam}\ x\ eB)) \qquad env \vdash e2 \Downarrow v2 \qquad x{=}v2,\ env_{\mathsf{old}} \vdash eB \Downarrow v}{env \vdash (\mathsf{app}\ e1\ e2) \Downarrow v}\ \text{Env-app}$$

Question: What happens if the lam has a free variable that isn't in the environment env_{old}, 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 $(\mathsf{clo}\ env_{\mathsf{old}}\ (\mathsf{lam} \ldots))$—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 env_{old}.

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 env_{old} to save.

3 Recursive closures

At this point, we could add most of the features of Fun++ (pairs, lists, etc.) without any trouble. Instead, let's try to add the feature that *will* give us trouble: `rec`.

The difficulty with `rec` is that we need to create a closure in which the saved environment *has a binding to that same closure*. The best way I've found for doing this in Racket is to use *boxes*.

3.1 Boxes in Racket

A Racket *box* is like a pointer or reference that points to a value that can be mutated (updated).

- You can create a Racket box with `box`. The way Racket prints a box looks kind of weird. It will get weirder.

  ```
  > (box 5)
  '#&5
  > (box (list 1 2 3))
  '#&(1 2 3)
  > (box "hello")
  '#&"hello"
  ```

- You can get the contents out of a box with `unbox`:

  ```
  > (unbox (box 5))
  5
  > (unbox (box (lambda (x) x)))
  #<procedure>
  > (define box1 (box 5))
  > (unbox box1)
  5
  ```

- You can update a box with `set-box!`:

  ```
  > (set-box! box1 111)          ; contained 5...
  > (unbox box1)                 ; ...now contains 111
  111
  ```

You can make a box's contents *be itself*:

```
> (set-box! box1 box1)
#0='#&#0#
> box1
#0='#&#0#
```

This "line noise" is Racket trying to "draw" a diagram in which the box points back to itself:

- `#0=` means "I am labelling this box 0";
- `'#&` is Racket's usual "here is a box, whose contents follow";
- `#0#` is a reference back to the box labelled 0.

It may be easier to understand if we make a circular "list" (the term "list" often implies that there are no cycles):

```
> (set-box! box1 (list 1 2 box1))
> box1
#0='#&(1 2 #0#)
```

We can understand this as: "Here is a box labelled 0, and inside the box is a list whose first element is 1, whose second element is 2, and whose third element is the box labelled 0."

```
> (set-box! box1 (list 1 2 3 box1 5 6))
> box1
#0='#&(1 2 3 #0# 5 6)
```

Now the box contains a list, whose 4th element points back to the box itself.

We will use this technique to construct a recursive closure: a closure whose environment *env*2 binds an identifier u to *that same closure*.

3.2 Adding a recursive closure

Our *recursive closure* clo-rec will differ from the the previous closure clo: the environment will be in a Racket box.

```
(define-type E
  [num (n number?)]
  [add (lhs E?) (rhs E?)]
  [sub (lhs E?) (rhs E?)]
  [with (name symbol?) (named-expr E?) (body E?)]
  [id (name symbol?)]
  [lam (name symbol?) (body E?)]
  [clo (env Env?) (e E?)]                ; not in concrete syntax
  [clo-rec (box-env box?) (e E?)]        ; not in concrete syntax
  [app (function E?) (argument E?)]
  )
```

Also, because we will use clo-rec as a closure around rec expressions, if we try to evaluate a clo-rec we will evaluate its body, rather than treating it as a value (as we did with clo).

To make this work, we need to change Env-id to evaluate the resulting expression, because the resulting expression might be a clo-rec.

I'm not completely sure this is the best or only way to do this—I found it fairly easy to get something that "worked" for good, terminating Fun code, but harder to make expressions that should be nonterminating actually not terminate…

3.3 Rules for recursive closures

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

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

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

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

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

$$\frac{\mathit{lookup}(env, x) = e \qquad e \Downarrow v}{env \vdash (\mathsf{id}\ x) \Downarrow v}\ \text{Env-id} \qquad \frac{\mathit{lookup}(env, x)\ \text{undefined}}{env \vdash (\mathsf{id}\ x)\ \mathsf{unknown\text{-}id\text{-}error}}\ \text{Env-unknown-id}$$

$$\frac{}{env \vdash (\mathsf{lam}\ x\ e1) \Downarrow \big(\mathsf{clo}\ env\ (\mathsf{lam}\ x\ e1)\big)}\ \text{Env-lam} \qquad \frac{}{env \vdash (\mathsf{clo}\ env_{old}\ e) \Downarrow (\mathsf{clo}\ env_{old}\ e)}\ \text{Env-clo}$$

$$\frac{env \vdash e1 \Downarrow (\mathsf{clo}\ env_{old}\ (\mathsf{lam}\ x\ eB)) \qquad env \vdash e2 \Downarrow v2 \qquad x{=}v2, env_{old} \vdash eB \Downarrow v}{env \vdash (\mathsf{app}\ e1\ e2) \Downarrow v}\ \text{Env-app}$$

$$\frac{env_{old} \vdash e \Downarrow v}{env \vdash (\mathsf{clo\text{-}rec}\ env_{old}\ e) \Downarrow v}\ \text{Env-clo-rec} \qquad \frac{\overbrace{u{=}(\mathsf{clo\text{-}rec}\ env2\ e), env}^{env2} \vdash e \Downarrow v}{env \vdash (\mathsf{rec}\ u\ e) \Downarrow v}\ \text{??Env-rec}$$

$$\frac{\mathcal{E} \Rightarrow \big(u{=}(\mathsf{clo\text{-}rec}\ \mathcal{E}\ e), env\big) \vdash e \Downarrow v}{env \vdash (\mathsf{rec}\ u\ e) \Downarrow v}\ \text{Env-rec}$$