Environment-Passing Semantics and Variable Scoping

CPSC 509: Programming Language Principles

Ronald Garcia*

24 February 2014

TFL: A Tiny Functional Language

Our basis language in these notes is a tiny functional language we'll call TFL. It builds on the call-by-value language of procedures by adding a let construct.

$$
\begin{array}{lcl}
x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad t \in \textsc{Term}, \quad v \in \textsc{Value}, \quad \textsc{Pgm} = \{\, t \in \textsc{Term} \mid FV(t) = \emptyset \,\} \\
t & ::= & x \mid n \mid t\ t \mid \lambda x.t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \mid t + t \\
v & ::= & n \mid \lambda x.t
\end{array}
$$

Here is a big-step semantics for TFL.

$\boxed{\Downarrow \subseteq \textsc{Pgm} \times \textsc{Value}}$

$$\frac{}{n \Downarrow n}\ \text{(num)} \qquad \frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1 + t_2 \Downarrow n}\ \text{(plus)}\quad n = n_1 + n_2 \qquad \frac{}{\lambda x.t \Downarrow \lambda x.t}\ \text{(lambda)}$$

$$\frac{t_1 \Downarrow \lambda x.t_{11} \quad t_2 \Downarrow v_2 \quad [v_2/x]t_{11} \Downarrow v}{t_1\ t_2 \Downarrow v}\ \text{(app)} \qquad \frac{(\lambda x.t_2)\ t_1 \Downarrow v}{\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow v}\ \text{(let)}$$

The (lambda) rule says "to evaluate a lambda, just return it: you're done". The (app) rule for applications evaluates the operator t_1, expecting a lambda abstraction $\lambda x.t_{11}$ as the result, and evaluates the operand t_2, expecting a value v_2, and then substitutes v_2 into the body of the operator (i.e. t_{11}). For the let rule, we "cheat", and just define let in terms of lambda abstractions and application: this makes explicit the close relationship between variable binding and procedure application. How would you write an equivalent rule for let directly? The direct approach is more typical: I do it as above just to make the point that we can reasonably define some language features in terms of others, and doing so codifies the language designer's intent, but the resulting reasoning principles are not always what you want by default.

The evaluator for this semantics looks similar to the evaluator for small-step semantics:

$$
\begin{array}{rcl}
\textsc{Obs} & = & \{\, \mathsf{procedure} \,\} \cup \mathbb{Z} \\
\mathit{eval} & : & \textsc{Pgm} \to \textsc{Obs} \\
\mathit{eval}(t) & = & \mathsf{procedure}\ \text{if}\ t \Downarrow \lambda x.t_0 \\
\mathit{eval}(t) & = & n\ \text{if}\ t \Downarrow n.
\end{array}
$$

Figure 1 presents derivation tree for evaluating the following program:

```
let x = 7
in let plusx = λy.y + x
   in plusx 3
```

$$\dfrac{\dfrac{\dfrac{}{\lambda x.\mathsf{let}\ \mathsf{plusx} = \lambda y.y + x\ \mathsf{in}\ \mathsf{plusx}\ 3 \Downarrow \lambda x.\mathsf{let}\ \mathsf{plusx} = \lambda y.y + x\ \mathsf{in}\ \mathsf{plusx}\ 3} \quad \dfrac{}{7 \Downarrow 7} \quad \dfrac{\dfrac{}{\lambda \mathsf{plusx}.\mathsf{plusx}\ 3 \Downarrow \lambda \mathsf{plusx}.\mathsf{plusx}\ 3} \quad \dfrac{}{\lambda y.y + 7 \Downarrow \lambda y.y + 7} \quad \dfrac{\dfrac{}{\lambda y.y + 7 \Downarrow \lambda y.y + 7} \quad \dfrac{}{3 \Downarrow 3} \quad \dfrac{\dfrac{}{3 \Downarrow 3} \quad \dfrac{}{7 \Downarrow 7}}{3 + 7 \Downarrow 10}}{(\lambda y.y + 7)\ 3 \Downarrow 10}}{(\lambda \mathsf{plusx}.\mathsf{plusx}\ 3)\ \lambda y.y + 7 \Downarrow 10}}{\mathsf{let}\ \mathsf{plusx} = \lambda y.y + 7\ \mathsf{in}\ \mathsf{plusx}\ 3 \Downarrow 10}}{\left(\lambda x.\mathsf{let}\ \mathsf{plusx} = \lambda y.y + x\ \mathsf{in}\ \mathsf{plusx}\ 3\right)\ 7 \Downarrow 10}}{\mathsf{let}\ x = 7\ \mathsf{in}\ \mathsf{let}\ \mathsf{plusx} = \lambda y.y + x\ \mathsf{in}\ \mathsf{plusx}\ 3 \Downarrow 10}$$

Figure 1: TFL Derivation of a simple program's evaluation

Thinking about Scope

The language definition above is very succinct and explains how function calls pass arguments to functions in terms of substitution $[v/x]t$, which is defined as a mathematical function outside of the operational semantics. This is a concise and clear specification, but it's not particularly helpful if we want to understand other kinds of variable lookup. To enable a discussion of different varible scoping disciplines, we now consider a different model of the TFL language.

Here we use a mechanism, called an *environment*. An environment is a finite partial function from *variables* to *denotable objects*, which in this case are values:[1]

$$\rho \in \textsc{Env} = \textsc{Var} \xrightarrow{\text{fin}} \textsc{Do}$$
$$\textsc{Do} = \textsc{Value}$$

The smallest environment is the empty one $\emptyset$, and given an environment ρ we use the notation $\rho[x \mapsto v]$ to represent the environment that either extends (if $x \notin dom(\rho)$) or differs from (if $x \in dom(\rho)$) ρ regarding its mapping for x:

$$\cdot[\cdot \mapsto \cdot] : \textsc{Env} \times \textsc{Var} \times \textsc{Value} \to \textsc{Env}$$
$$\rho[x \mapsto v](y) = v \qquad \text{if } y = x$$
$$\rho[x \mapsto v](y) = \rho(y) \quad \text{otherwise.}$$

Notice how our definition of environment update uses equations that talk *not* about what $\rho[x \mapsto v]$ *equals*, but rather the equational properties of *applying* $\rho[x \mapsto v]$ to variables. This is a perfectly fine equational definition of the $\cdot[\cdot \mapsto \cdot]$ function itself.

Lexical (a.k.a. Static) Scoping

We'll now define an *environment-passing* semantics for TFL, which is a semantics that, instead of using substitution, stores variable bindings in the environment. This semantics, like the one for IMP, is defined using pairs:

$$(\cdot \vdash \cdot \Downarrow \cdot) \subseteq \textsc{Env} \times \Lambda \times \textsc{Value}$$

The environment-passing semantics for TFL follow:

$$\frac{}{\rho \vdash x \Downarrow \rho(x)}\ \text{(var)} \qquad \frac{}{\rho \vdash n \Downarrow n}\ \text{(num)} \qquad \frac{\rho \vdash t_1 \Downarrow n_1 \quad \rho \vdash t_2 \Downarrow n_2}{\rho \vdash t_1 + t_2 \Downarrow n}\ \text{(plus)} \quad n = n_1 + n_2$$

$$\frac{}{\rho \vdash \lambda x.t \Downarrow (\lambda x.t, \rho)}\ \text{(lambda)} \qquad \frac{\rho_1 \vdash t_1 \Downarrow (\lambda x.t_{11}, \rho_2) \quad \rho_1 \vdash t_2 \Downarrow v_2 \quad \rho_2[x \mapsto v_2] \vdash t_{11} \Downarrow v}{\rho_1 \vdash t_1\ t_2 \Downarrow v}\ \text{(app)}$$

$$\frac{\rho \vdash (\lambda x.t_2)\ t_1 \Downarrow v}{\rho \vdash \text{let } x = t_1 \text{ in } t_2 \Downarrow v}\ \text{(let)}$$

There are a few things to notice about these semantics. First off, the result of evaluating a lambda abstraction is now a pair of that abstraction and the environment that it¡ was evaluated in. Given a pair $(\lambda x.t, \rho)$, the environment ρ must be defined for all the free variables of $\lambda x.t$. Thus, the pair represents a closed lambda abstraction, and for this reason it is called a *closure*. In this semantics, closures are among our values.

$$v ::= n \mid (\lambda x.t, \rho) \quad \text{where } FV(\lambda x.t) \subseteq dom(\rho)$$

[1] "Denote" is just a fancy word for "stands for". Denotable objects are thus the objects in the language that variables are allowed to stand for. What is allowed as a denotable object is a design decision. For instance, I may only want to allow variables to hold numbers and never functions. Such a language is more like Pascal than like Haskell.

The (app) rule also changes to account for closures and the avoidance of substitution. Now, evaluating the operator t_1 results in a closure, and rather than substituting the result v_2 of the operand into the body t_1, the body t_{11} is evaluated as-is, but in an environment that extends the closure environment ρ_2 with a binding for x. The environment ρ_2 accounts for all the free variables in $\lambda x.t$, providing variable bindings that would have been immediately substituted into $\lambda x.t$ in the previous semantics. Extending the environment takes the place of performing a substitution.

Another important item of note. For these semantics *we need not consider α-equivalent terms!* Now that we do not perform substitution, we can simply rely on the environment to keep track of concrete variables. In short, environments obviate need for lexical scoping.

We can reason about which variable reference goes with which variable binding exactly as we did in the other semantics for TFL, and it's pretty straightforward. This kind of variable scoping is called *static scoping*, because we can discern what variable binding a variable refers to simply by looking at the initial program. That is to say, we can determine variable scope statically (i.e. without running the program).

How would the evaluation derivation in Figure 1 change if it ended with the following judgment?

$$\emptyset \vdash \left(\begin{array}{l}\mathsf{let\ x} = 7 \\ \mathsf{in\ let\ plusx} = \lambda \mathsf{y.y} + \mathsf{x} \\ \quad \mathsf{in\ plusx}\ 3\end{array}\right) \Downarrow 10$$

Dynamic Scoping

We now consider a new variant of the TFL language, which we'll call TFL_{dyn}. This language is *dynamically scoped* which means that variable references are resolved using the "most recent" binding of that variable in the *dynamic extent* of evaluation. Ideally we can unpack what this means by considering the semantics themselves.

The environment-passing semantics for TFL_{dyn} follow:

$$\frac{}{\rho \vdash x \Downarrow \rho(x)}\ \text{(var)} \qquad \frac{}{\rho \vdash n \Downarrow n}\ \text{(num)} \qquad \frac{\rho \vdash t_1 \Downarrow n_1 \quad \rho \vdash t_2 \Downarrow n_2}{\rho \vdash t_1 + t_2 \Downarrow n}\ \text{(plus)}\ \text{where } n = n_1 + n_2$$

$$\frac{}{\rho \vdash \lambda x.t \Downarrow \lambda x.t}\ \text{(lambda)} \qquad \frac{\rho_1 \vdash t_1 \Downarrow \lambda x.t_{11} \quad \rho_1 \vdash t_2 \Downarrow v_2 \quad \rho_1[x \mapsto v_2] \vdash t_{11} \Downarrow v}{\rho_1 \vdash t_1\ t_2 \Downarrow v}\ \text{(app)}$$

$$\frac{\rho \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow v}{\rho \vdash (\lambda x.t_2)\ t_1 \Downarrow v}\ \text{(let)}$$

The differences between the dynamically scoped and statically scoped semantics are highlighted above. Most importantly, the evaluation rule for lambda abstractions now drops the environment and simply returns the lambda abstraction itself. In contrast to the original TFL semantics, though, this lambda abstraction may have free variables, and they need to be resolved somehow. This phenomenon is central to why scoping has become dynamic.

In any case, values have now reverted to being numbers or lambda abstractions.

$$v ::= n \mid \lambda x.t$$

Because of this, the rule for function application also changes. Now, there is no longer a second environment ρ_2. Instead, the body t_{11} of the operator is evaluated using the environment ρ_1 that exists *when the function is being applied*, not the one that existed *when the function was created*. This has massive implications for the runtime behavior of the language.

Consider a slight variation on our earlier programming example:

```
let x = 7
in let plusx = λy.y + x
   in let x = 3
      in plusx 3
```

In the TFL language, the result is the same as before, 10, because the value of x in the body of plusx is resolved statically to be 7 (If you are not sure, run it in the substitution-based semantics).

In the TFL_{dyn} language, however, the value of x ends up being looked up in the environment that exists at the point where plusx is called. Looking at this code, we can see that x is bound to 3 at the point in the program when this call happens. Thus, x is found to be 3, and the result of the computation is 6.

Dynamic scope can be difficult to reason about. It's not always obvious which variable binding will be in scope when a variable is looked up. To give you some flavor of how evaluation proceeds, consider the evaluation of the following program (Figure 2).

$$(\lambda x.(\lambda x.\lambda y.x + y)\ 5\ 3)\ 7$$

This program is *similar* to translating away the let expressions in the following program:

```
let x = 7
in let x = 5
   in let y = 3
      in x + y
```

but they are *not quite* the same in a critical way. Try translating this program's let expressions away by hand. How does the result compare to the prior program?

Under static scoping (i.e. TFL), our let-less program evaluates to 8, but under dynamic scoping (i.e. TFL_{dyn}) the result is altogether different. Observe how currying causes the binding of x to 5 to be lost, leaving only the binding of x to 7 when it is time to look up x's value.

Harking back to our earlier discussion of alpha-equivalence, we once again do not consider α-equivalent terms. In the case of lexical scoping, they were not needed. In the case of dynamic scoping, they don't even work! That is to say, changing the name of a bound variable can change the meaning of the program if the language has dynamic scoping. In fact, this is why dynamic scoping is considered such a bad idea: local decisions about variable names have global implications. There is no encapsulation of variable-naming decisions.

$$\dfrac{\dfrac{\dfrac{\rho_1 \vdash t_3 \Downarrow t_3 \quad \rho_1 \vdash 5 \Downarrow 5 \quad \dfrac{\rho_2 \vdash \lambda y.x + y \Downarrow \lambda y.x + y}{\rho_1 \vdash (\lambda x.\lambda y.x + y)\ 5 \Downarrow \lambda y.x + y} \quad \dfrac{\rho_3 \vdash x \Downarrow 7 \quad \rho_3 \vdash y \Downarrow 3}{\rho_3 \vdash x + y \Downarrow 10}}{\rho_1 \vdash 3 \Downarrow 3}}{\rho_1 \vdash (\lambda x.\lambda y.x + y)\ 5\ 3 \Downarrow 10}}{\dfrac{\rho_0 \vdash (\lambda x.(\lambda x.\lambda y.x + y)\ 5\ 3)\ 7 \Downarrow 10}{\rho_0 \vdash (\lambda x.(\lambda x.\lambda y.x + y)\ 5\ 3) \Downarrow (\lambda x.(\lambda x.\lambda y.x + y)\ 5\ 3)} \quad \dfrac{\rho_0 \vdash 7 \Downarrow 7}{}}$$

$$\rho_0 = \emptyset$$
$$\rho_1 = \rho_0[x \mapsto 7] = (x \mapsto 7)$$
$$\rho_2 = \rho_1[x \mapsto 5] = (x \mapsto 5)$$
$$\rho_3 = \rho_1[y \mapsto 3] = (x \mapsto 7, y \mapsto 3)$$

Figure 2: Derivation of Evaluation in TFL_{dyn}

Reduction Semantics for Dynamic Scoping

The environment-passing semantics in the last section were inspired by early implementations of languages like LISP[2], which got environments wrong, and ended up with dynamic scoping. The Scheme programming language corrected this design issue, taking inspiration from Church's lambda calculus as well as Carl Hewitt's Actor Model from the PLANNER programming language.

An easier way to understand dynamic scoping is to look at it as a stack-based lookup discipline. In particular, we can write a reduction semantics for dynamic scoping that produces the same results as the environment-based semantics, but that makes it far easier to trace what's going on. Technically we should prove that the two semantics are equivalent, but let's save that for another day: for now examples may help you get the idea.

The reduction semantics for the language follows. We start with the evaluation contexts:

$$
\begin{array}{l}
E \in \text{ECTXT} \\
E \quad ::= \quad \Box \mid E[\Box\ t] \mid E[v\ \Box] \mid E[\mathsf{let}\ x = \Box\ \mathsf{in}\ t] \mid E[\mathsf{let}\ x = v\ \mathsf{in}\ \Box] \mid E[\Box + t] \mid E[v + \Box]
\end{array}
$$

Now we have evaluation contexts for let expressions: we will not be defining them in terms of λ. In fact, we will essentially do the reverse. Also, unlike previous reduction semantics with let, this one allows evaluation in the *body* of the let expression. In essence, the let binding captures the value of a variable in the context, which is our representation of the "dynamic scope" of any variable reference.

Our notions of reduction follow:

$$
\begin{array}{rcl}
n_1 + n_2 & \rightsquigarrow & n_1 + n_2 \\
(\lambda x.t)\ v & \rightsquigarrow & \mathsf{let}\ x = v\ \mathsf{in}\ t \\
\mathsf{let}\ x = v\ \mathsf{in}\ E[x] & \rightsquigarrow & \mathsf{let}\ x = v\ \mathsf{in}\ E[v] \quad x \notin BV(E) \\
\mathsf{let}\ x = v_1\ \mathsf{in}\ v_2 & \rightsquigarrow & v_2
\end{array}
$$

The first notion of reduction is typical, but from there things differ. Function application is now converted into let binding. This is how a variable binding is stored in the context for later lookup while evaluating the function body t.

The third rule is where the magic happens. Now that evaluation contexts can go inside the bodies of let expressions, it's possible to arrive at a free variable x. The behaviour of this operation is to find the innermost let binding for x and make a *copy* of the value that is bound to it. To capture the idea of innermost binding, we require a function on evaluation contexts that checks what variables are bound by the context:

$$
\begin{array}{rcl}
BV : \text{ECTXT} \to \mathcal{P}(\text{VAR}) & & \\
BV(\Box) & = & \emptyset \\
BV(E[\Box\ t]) & = & BV(E) \\
BV(E[v\ \Box]) & = & BV(E) \\
BV(E[\mathsf{let}\ x = \Box\ \mathsf{in}\ t]) & = & BV(E) \\
BV(E[\mathsf{let}\ x = v\ \mathsf{in}\ \Box]) & = & BV(E) \cup \{\, x \,\}
\end{array}
$$

In essence, the third notion of reduction includes the context surrounding the variable reference, meaning that this term could be *huge* depending on how big E is. This is quite different from the big-step rule for variable references, which appeals to an environment.

Finally, when a value v_2 is produced in the context of a let binding, then the let is no longer needed, so it is *popped* off of the context (i.e., "runtime stack"!) and the value propagates up.

Now let's consider our example program from earlier under this model.

[2] And, sadly, the current implementation of Emacs Lisp, though there is now an option to enable lexical scoping for a buffer!

$$
\begin{aligned}
& (\lambda x.(\lambda x.\lambda y.x + y)\ 5\ 3)\ 7 \\
\longrightarrow\ & \text{let } x = 7 \text{ in } (\lambda x.\lambda y.x + y)\ 5\ 3 \\
\longrightarrow\ & \text{let } x = 7 \text{ in } (\text{let } x = 5 \text{ in } \lambda y.x + y)\ 3 \\
\longrightarrow\ & \text{let } x = 7 \text{ in } (\lambda y.x + y)\ 3 \\
\longrightarrow\ & \text{let } x = 7 \text{ in let } y = 3 \text{ in } x + y \\
\longrightarrow\ & \text{let } x = 7 \text{ in let } y = 3 \text{ in } 7 + y \\
\longrightarrow\ & \text{let } x = 7 \text{ in let } y = 3 \text{ in } 7 + 3 \\
\longrightarrow\ & \text{let } x = 7 \text{ in let } y = 3 \text{ in } 10 \\
\longrightarrow\ & \text{let } x = 7 \text{ in } 10 \\
\longrightarrow\ & 10
\end{aligned}
$$

The key reduction step is the third one, which removes the $x = 5$ binding from the context. Then looking up x later yields 7.