The Environment Model of Evaluation

Jim Royer
CIS 352
March 26, 2015

LFP = LC + λ + function application + variables

LFP Expressions

\[ E ::= n \mid b \mid \ell \mid E \text{ iop } E \mid E \text{ cop } E \mid \text{ if } E \text{ then } E \text{ else } E \]
\[ \mid !E \mid E ::= E \mid \text{ skip } \mid E ; E \mid \text{ while } E \text{ do } E \]
\[ \mid x \mid \lambda x.E \mid E \text{ E} \mid \text{ let } x = E \text{ in } E \]

where

- \( x \in V \), an infinite set of variables
- \( n \in \mathbb{Z} \) (integers), \( b \in B \) (booleans), \( \ell \in L \) (locations)
- \( \text{iop} \in \) (integer-valued binary operations)
- \( \text{cop} \in \) (boolean-valued binary comparisons)

We focus on the (λ-calculus + let) part of LFP.

References


Application via substitution and its problems

Call by name

\[ \downarrow_{-cbn} \frac{(E_1,s) \downarrow (\lambda x.E'_1,s') \downarrow (E'_1[E_2/x],s') \downarrow (V,s'')}{(E_1 E_2),s} \downarrow (V,s'')} \]

Call by value

\[ \downarrow_{-cbv} \frac{(E_1,s) \downarrow (\lambda x.E'_1,s') \downarrow (E_2,s') \downarrow (V_2,s'') \downarrow (E'_1[V_2/x],s'') \downarrow (V,s''')}{(E_1 E_2),s} \downarrow (V,s''')} \]

- Call-by-name and call-by-value are defined above via substitution.
- Substitution is:
  - dandy for nailing down sensible meanings of application.
  - stinko for everyday implementations.
  - E.g., An implementation via substitution constantly needs to modify a program’s source code.

  Idea In place of substituting a value \( v \) for a variable \( x \), keep a dictionary of variables and their values and when you need the value of \( x \), look it up.
Environments  (Warning: Scary Greek letters)

Definition
An environment is just a table of variables and associated values.

Consider an expression \( e = \text{if } z \text{ then } x \text{ else } y + 2 \).
- With environment \{ \( x \mapsto 3 \), \( y \mapsto 4 \), \( z \mapsto \text{tt} \) \}, \( e \) evaluates to 3.
- With environment \{ \( x \mapsto 8 \), \( y \mapsto 5 \), \( z \mapsto \text{ff} \) \}, \( e \) evaluates to 7.
- Etc.

\( \text{lookup}(\rho, x) \) returns the value (if any) of \( x \) in environment \( \rho \).

\( \text{update}(\rho, x, v) \) returns a new environment \( \rho[x \mapsto v] \)
(\( \rho[x \mapsto v] \) is just like \( \rho \) except \( x \) has value \( v \).)

Evaluating variable \( x \) in environment \( \rho \equiv \text{lookup}(\rho, x) \).

Revising call-by-value big-step semantics, 1

Definition
\( \rho \vdash \langle e, s \rangle \Downarrow_V \langle v', s' \rangle \) means that expression \( e \) with environment \( \rho \) and state \( s \) evaluates to value \( v \) and state \( s' \).

\[
\begin{align*}
\text{Var: } & \quad \rho \vdash \langle x, s \rangle \Downarrow_V \langle v, s \rangle \quad (v = \text{lookup}(\rho, x)) \\
\text{Let: } & \quad \rho \vdash \langle e_1, s \rangle \Downarrow_V \langle v_1, s' \rangle \\
& \quad \rho \vdash \langle \text{let } x = e_1 \text{ in } e_2, s \rangle \Downarrow_V \langle v_2, s'' \rangle \\
& \quad \rho \vdash \langle \lambda x. e, s \rangle \Downarrow_V \langle \lambda x. e, s \rangle
\end{align*}
\]

Examples/Exercises: Let \( \rho = \{ x \mapsto 7, y \mapsto 3 \} \).
- \( \rho \vdash \langle x + y, s \rangle \Downarrow_V ?? \)
- \( \rho \vdash \langle \text{let } x = 1 \text{ in } x + y, s \rangle \Downarrow_V ?? \)
- \( \rho \vdash \langle \text{let } x = 1 \text{ in } (\text{let } z = 11 \text{ in } x + y + z), s \rangle \Downarrow_V ?? \)

Revising call-by-value big-step semantics, 2

Preliminary versions of these rules:

\[
\begin{align*}
\rho \vdash \langle e_1, s \rangle \Downarrow_V \langle \lambda x. e'_1, s' \rangle \\
\rho \vdash \langle e_2, s' \rangle \Downarrow_V \langle v_2, s'' \rangle \\
\text{App: } & \quad \rho \vdash \langle e_1 e_2, s \rangle \Downarrow_V \langle v_1, s''' \rangle \\
& \quad \rho \vdash \langle \text{let } f = \lambda x. e + y \text{ in } f 10, s \rangle \Downarrow_V ?? \\
& \quad \rho \vdash \langle \text{let } f = \lambda x. e + y \text{ in } (\text{let } y = 100 \text{ in } f 10), s \rangle \Downarrow_V ??
\end{align*}
\]
Dynamic Scoping, 1

Re: λ-expressions, functions, procedures, etc.,
there are two sorts of environments you have to worry about:
1. The environment in force when the function was created.
2. The environment in force when the function is applied.

\[
\text{Dynamic-App: } \rho \vdash (e_1, s) \Downarrow v \langle \lambda x, e'_1, s' \rangle
\]

\[
\rho \vdash (e_2, s') \Downarrow v \langle v_2, s'' \rangle
\]

\[
\text{Dynamic-App: } \rho[x \mapsto v_2] \vdash (e'_1, s'') \Downarrow v \langle v, s''' \rangle
\]

Under dynamic scoping, when you apply a function in environment
\[
(\lambda x. e'_1) \text{ in environment } \rho
\]
you evaluate \(e'_1\) in environment \(\rho[x \mapsto v_2]\).

Example: Let \(\rho = \{ x \mapsto 7, y \mapsto 3 \}\) and consider
\[
\rho \vdash \langle \text{let } f = \lambda x. x + y \text{ in let } g = \lambda y. f(y + 100) \text{ in } (f 10) + (g 0), s \rangle \Downarrow v ??
\]

Dynamic Scoping, 2

Lexical Scoping, 1

Re: λ-expressions, functions, procedures, etc.,
there are two sorts of environments you have to worry about:
1. The environment in force when the function is created.
2. The environment in force when the function is applied.

In human language, statements need to be understood in context:

"Such a fact is probable, but undoubtedly false."
—Edward Gibbon in “Decline and Fall of the Roman Empire”

- When Gibbon was writing “probable” meant “well-recommended”.
- So in reading Gibbon we have to use a 1700’s English dictionary.
- We pull a similar trick for functions.
**Lexical Scoping, 2**

**Definition**

A closure, $e\rho$, is an expression $e$ with an environment $\rho$ such that $\text{fv}(e) \subseteq \text{domain}(\rho)$, i.e., all of $e$’s free variables are in $\rho$’s dictionary.

Ideas:
- A $\lambda$-expression evaluates to a closure.
- When we create a $\lambda$-expression, we “close” it with its definition-time environment.

\[ \text{Lexical-App: } \rho \vdash (e_1 e_2, s) \Downarrow (v, s'') \]

When we apply a function (i.e., closure $(\lambda x.e')\rho'$), we evaluate $e'$ in $\rho'[x \mapsto v]$, where $v$ is the value of the argument.

\[ \rho \vdash (e_1, s) \Downarrow (\lambda x.e'_1, s') \]
\[ \rho \vdash (e_2, s') \Downarrow (v_2, s'') \]

\[ \rho \vdash ((e_1 e_2), s) \Downarrow ((v, s'')) \]

**Examples/Exercises:** Let $\rho = \{ x \mapsto 7, y \mapsto 3 \}$.

- $\rho \vdash (\text{let } f = \lambda x.(x + y) \text{ in } (f 10), s) \Downarrow ??$
- $\rho \vdash (\text{let } f = \lambda x.(x + y) \text{ in } (\text{let } y = 100 \text{ in } (f 10)), s) \Downarrow ??$
- $\rho \vdash (\text{let } f = \lambda n. \text{ if } n \leq 0 \text{ then } 1 \text{ else } n * (f (n - 1)) \text{ in } (f 3), s) \Downarrow ??$

**Puzzle 1**

- $\rho_1 = [a \mapsto 1, b \mapsto 2]$

\[ e_1 = \text{let } q = \lambda a.(a + b) \text{ in } \text{let } a = 5 * b \text{ in } \text{let } b = a * b \text{ in } (q 100) \]

What is the value of $e_1$ in environment $\rho_1$ under call-by-value with

(a) lexical scoping?

(b) dynamic scoping?
Puzzle 1(a): Call-by-value, lexical scoping

\[ \rho_1 = \begin{bmatrix} a \mapsto 1 \\ b \mapsto 2 \end{bmatrix} \]

\[ e_1 = \text{let } q = \lambda a.(a + b) \text{ in} \]
\[ \text{let } a = 5 \times b \text{ in} \]
\[ \text{let } b = a \times b \text{ in} \]
\[ (q 100) \]

\[ \rho_1: \] value of \( e_1\rho_1 \): 102

Puzzle 1(b): Call-by-value, dynamic scoping

\[ \rho_1 = \begin{bmatrix} a \mapsto 1 \\ b \mapsto 2 \end{bmatrix} \]

\[ e_1 = \text{let } q = \lambda a.(a + b) \text{ in} \]
\[ \text{let } a = 5 \times b \text{ in} \]
\[ \text{let } b = a \times b \text{ in} \]
\[ (q 100) \]

\[ \rho_1: \] value of \( e_1\rho_1 \): 120

Puzzle 2

\[ \rho_1 = [a \mapsto 1, \ b \mapsto 2] \]

\[ e_2 = \text{let } p = \lambda a.(a + b) \text{ in} \]
\[ \text{let } q = \lambda b.(a + (p \ b)) \text{ in} \]
\[ \text{let } a = 10 \text{ in} \]
\[ \text{let } b = 20 \]
\[ \text{in} \ (q 100) \]

What is the value of \( e_2 \) in environment \( \rho_1 \) under call-by-value with
(a) lexical scoping?
(b) dynamic scoping?
Lexical Scoping, 4: Closures + States = Objects

Suppose \((new \; v)\) returns a fresh location initialized to \(v\).

**Warning:** The following is tormented LFP; return is as in HW10.

\[
\begin{align*}
\text{let* } mkbox &= λx. (\text{let } bx = (new \; x) \text{ in } (λy. \{ \text{ bx := bx + y; return bx } \})); \\
&\quad u = (mxbox \; 10); \\
&\quad v = (mxbox \; (100 + (u \; 5))); \\
\text{in } ((u \; 0) + (v \; 0))
\end{align*}
\]

In more familiar notation, \(mkbox\) is roughly:

\[
\text{function } mkbox(x) = \{ \text{ var bx = (new x); return (function foo(v) } \{ \text{ bx := bx + v; return bx } \} ); \}
\]

In Java terms:
- box is a class
- \(mkbox\) is a box-creator
- \(u\) and \(v\) are instance methods
- \(bx\) is an instance variable.

Lexical Scoping, 5: What about call-by-name?

**Call by name**

\[
\begin{align*}
\text{Subst-App-cbn: } & \quad ⟨E_1, s⟩ \downarrow_N (λx.E'_1 / x, s') \quad ⟨E'_1[x/E_2], s'⟩ \downarrow_N (V, s'') \\
&\quad ⟨(E_1 \; E_2), s⟩ \downarrow_N (V, s'')
\end{align*}
\]

**Question:**
With environments, how do we simulate substituting the unevaluated \(E_2\) for \(x\) in \(E'_1\) that call-by-name requires?

**Answer:**
Thunks ≡ closures of arbitrary expressions, not just \(λ\)-expressions.
The Call-By-Name Version

\[ \rho \vdash (e_1,s) \Downarrow_N \rho \{ z \mapsto e' \} \Downarrow_N \rho \{ v_2,s'' \} \]

\[ \rho \vdash (e_2,s) \Downarrow_N \rho \{ v_1,s''' \} \]

Call-by-name/dynamic-scoping makes very little sense, … but we are implementing it any way in Homework 10.

Puzzle 3

\[ \rho_0 = \emptyset \]
\[ s_0 = [ \ell \mapsto 0] \]
\[ e_0 = \text{let} \ g = \lambda x.\{ \ell := !\ell + 1; \text{ return } x \}; \]
\[ z = (g \ 100) \]
\[ \text{in} \ (z + z + z) \]

What are \( v_1 \) and \( s_1 \) we use lexical scoping and
(a) call-by-value evaluation?
(b) call-by-name evaluation?

Puzzle 3(a): Call-by-value

\[ \rho_0 = \emptyset \]
\[ s_0 = [ \ell \mapsto 0] \]
\[ e_0 = \text{let} \ g = \lambda x.\{ \ell := !\ell + 1; \text{ return } x \}; \]
\[ z = (g \ 100) \]
\[ \text{in} \ (z + z + z) \]

What are \( v_1 \) and \( s_1 \) in
\[ \rho_0 \vdash (e_0,s_0) \Downarrow_V (v_1,s_1)? \]

\[ v_1 = 300 \]
\[ s_1 = [\ell \mapsto 1] \]

Puzzle 3(b): Call-by-name

\[ \rho_0 = \emptyset \]
\[ s_0 = [ \ell \mapsto 0] \]
\[ e_0 = \text{let} \ g = \lambda x.\{ \ell := !\ell + 1; \text{ return } x \}; \]
\[ z = (g \ 100) \]
\[ \text{in} \ (z + z + z) \]

What are \( v_1 \) and \( s_1 \) in
\[ \rho_0 \vdash (e_0,s_0) \Downarrow_N (v_1,s_1)? \]

\[ v_1 = 300 \]
\[ s_1 = [\ell \mapsto 3] \]
Puzzle 4

\[ \rho_0 = \emptyset \]
\[ s_0 = [\ell \mapsto 0] \]
\[ e_0 = \text{let}\ g = \lambda x.\{\ell := !\ell + 1; \text{return } x\}; \]
\[ h = \lambda y.2; \]
\[ \text{in} (h (g 89)) \]

Consider \( \rho_0 \vdash (e_0, s_0) \downarrow (v_1, s_1) \).

What are \( v_1 \) and \( s_1 \) we use lexical scoping and

(a) call-by-value evaluation?

(b) call-by-name evaluation?

---

Puzzle 4(a): Call-by-value

\[ \rho_0 = \emptyset \]
\[ s_0 = [\ell \mapsto 0] \]
\[ e_0 = \text{let}\ g = \lambda x.\{\ell := !\ell + 1; \text{return } x\}; \]
\[ h = \lambda y.2; \]
\[ \text{in} (h (g 89)) \]

What are \( v_1 \) and \( s_1 \) in

\[ \rho_0 \vdash (e_0, s_0) \downarrow^V (v_1, s_1)? \]

---

Puzzle 4(b): Call-by-name

\[ \rho_0 = \emptyset \]
\[ s_0 = [\ell \mapsto 0] \]
\[ e_0 = \text{let}\ g = \lambda x.\{\ell := !\ell + 1; \text{return } x\}; \]
\[ h = \lambda y.2; \]
\[ \text{in} (h (g 89)) \]

What are \( v_1 \) and \( s_1 \) in

\[ \rho_0 \vdash (e_0, s_0) \downarrow^V (v_1, s_1)? \]

---

Recursion under lexical scoping, 1

Recall:

\[ E ::= \ldots \mid \text{rec } x.E \]

Informally: “\( \text{rec } x.E \)” reads recursively define \( x \) to be \( E \).

The big-step operational semantics is given by:

\[ \text{unfolding}_{\text{sub}}: \quad \langle E[\text{rec } x.E]/x, s \rangle \downarrow \langle V, s' \rangle \]

\[ \langle \text{rec } x.E, s \rangle \downarrow \langle V, s' \rangle \]
The substitution-based version of unfold

\[
\text{unfolding}_{\text{sub}}: \quad \langle E[(\text{rec} \; x.E)/x], s \rangle \Downarrow \langle V, s' \rangle
\]

\[
\langle \text{rec} \; x.E, s \rangle \Downarrow \langle V, s' \rangle
\]

The environment-based version of unfold

\[
\text{unfolding}_{\text{env}}: \quad \rho[x \mapsto \text{rec} \; x.E] \vdash \langle E, s \rangle \Downarrow \langle V, s' \rangle
\]

\[
\rho \vdash \langle \text{rec} \; x.E, s \rangle \Downarrow \langle V, s' \rangle
\]

Try:

\[
\vdash \langle \text{rec} \; z.(\text{if } !\ell > 0 \text{ then } (!\ell - 1; \; z) \text{ else skip}), \{ \ell :-2 \} \rangle \Downarrow ??
\]