The Environment Model of Evaluation

Jim Royer
CIS 352
March 27, 2018

The Environment Model of Evaluation 1 / 31

References

- Structure and Interpretation of Computer Programs, 2/e, §3.2: The Environment Model of Evaluation, by Harold Abelson and Gerald Sussman, MIT Press, 1996.

LFP = LC + λ + function application + variables

LFP Expressions

\[ E ::= n \mid b \mid \ell \mid \text{iop } E \mid \text{cop } E \mid \text{if } E \text{ then } E \text{ else } E \]
\[ \mid \text{let } x = E \text{ in } E \]

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 (\( \lambda \)-calculus + let) part of LFP.

Application via substitution and its problems

Call by name

\[ \Downarrow_{\text{cbn}}: \langle E_1, s \rangle \Downarrow \langle \lambda x. E'_1, s' \rangle \Downarrow \langle E'_1[E_2/x], s' \rangle \Downarrow \langle V, s'' \rangle \]
\[ \langle (E_1 E_2), s \rangle \Downarrow \langle V, s'' \rangle \]

Call by value

\[ \Downarrow_{\text{cbv}}: \langle E_1, s \rangle \Downarrow \langle \lambda x. E'_1, s' \rangle \Downarrow \langle E_2, s' \rangle \Downarrow \langle V_2, s'' \rangle \Downarrow \langle E'_1[V_2/x], s'' \rangle \Downarrow \langle V, s''' \rangle \]
\[ \langle (E_1 E_2), s \rangle \Downarrow \langle V, s''' \rangle \]

- 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 & their values.
- When you need the value of \( x \), look it up.
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.

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

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

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

---

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

**Preliminary versions of these rules:**

\[
\rho \vdash \langle e_1, s \rangle \Downarrow_V \langle \lambda x. e'_1, s' \rangle \\
\rho \vdash \langle e_2, s' \rangle \Downarrow_V \langle v_2, s'' \rangle \\
\rho[x \mapsto v_2] \vdash \langle e'_1, s'' \rangle \Downarrow_V \langle v, s''' \rangle \\
\rho \vdash \langle (e_1, e_2), s \rangle \Downarrow_V \langle v, s''' \rangle \\
\rho \vdash \langle \lambda x. e, s \rangle \Downarrow_V \langle \lambda x. e, s \rangle
\]

**Examples/Exercises:** Let \( \rho = \{ x \mapsto 7, y \mapsto 3 \} \).
- \( \rho \vdash \langle \text{let } f = \lambda x. (x + y) \text{ in } (f 10), s \rangle \Downarrow_V ?? \)
- \( \rho \vdash \langle \text{let } f = \lambda x. (x + y) \text{ in } (let y = 100 \text{ in } (f 10)), s \rangle \Downarrow_V ?? \)
Scoping

Definition (Variable Scope)

The scope of a variable binding/declaration is the region of a program where the binding is valid, i.e., when you use the variable, it uses that declaration for the binding (meaning) of the name.

A Java example (static/lexical scoping)

```java
{  
    int i = 23;
    for (int i = 1; i<11; i++) { ... }  
    System.out.println(i);  
    ...  
}
```

- the outer i’s scope
- the inner i’s scope

Dynamic Scoping, 1

Re: \( \lambda \)-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.

```
ρ \vdash (e_1, s) \Downarrow V (λx.e'_1, s')  
ρ \vdash (e_2, s') \Downarrow V (v_2, s'')
```

Dynamic-App:

```
ρ[x \mapsto v_2] \vdash (e'_1, s'') \Downarrow V (v, s'''')

ρ \vdash ((e_1 e_2), s) \Downarrow V (v, s'''')
```

Example: Let \( ρ = \{ x \mapsto 7, y \mapsto 3 \} \) and consider

```
ρ \vdash \langle \text{let } f = λx.x + y \text{ in let } g = λy.f(y + 100) \text{ in } (f 10) + (g 0), s \rangle \Downarrow ??
```

Dynamic Scoping, 2

Under dynamic scoping, when you apply a function in environment \( ((λx.e'_1) e_2) \) in environment \( ρ \)

you evaluate \( e'_1 \) in environment \( ρ[x \mapsto v_2] \).

Question:

Is this a bug or a feature?

Dynamic Scoping, 3

```
ρ \vdash (e_1, s) \Downarrow V (λx.e'_1, s')  
ρ \vdash (e_2, s') \Downarrow V (v_2, s'')
```

Dynamic-App:

```
ρ[x \mapsto v_2] \vdash (e'_1, s'') \Downarrow V (v, s'''')

ρ \vdash ((e_1 e_2), s) \Downarrow V (v, s'''')
```

What goes right under dynamic scoping?

```
\text{let } f = λn. \text{ if } n \leq 0 \text{ then 1 else } n * (f (n - 1)) \text{ in } (f 3)
```

History

Discovered and formalized in early (≈1960s) Lisp implementations.
Lexical Scoping, 1

**Re:** \(\lambda\)-expressions, functions, procedures, etc.,

there are two sorts of environments you have to worry about:

- The environment in force when the function is **created**.
- 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, \(ep\), is an expression \(e\) with an environment \(\rho\) such that \(\{v\} \subseteq \text{domain}(\rho)\), 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.

\[
\text{Lexical-Fun: } \rho \vdash (\lambda x.s, s) \Downarrow ((\lambda x.s)\rho, s)
\]

- 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.

\[
\begin{align*}
\rho \vdash (e_1, s) & \Downarrow ((\lambda x.e')\rho', s') \\
\rho \vdash (e_2, s') & \Downarrow (v_2, s'') \\
\rho' \vdash (e_1', s'') & \Downarrow (v, s'''') \\
\rho \vdash (e_1, e_2, s) & \Downarrow ((v, s''''))
\end{align*}
\]

\[
\text{Lexical-App: } \rho'[x \mapsto v_2] \vdash (e_1', s''') \Downarrow (v, s''''')
\]

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

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

\[ \rho_1 = [a \mapsto 1, \ b \mapsto 2] \]
\[ e_1 = \text{let } q = \lambda a. (a + b) \text{ in} \]
\[ \quad \text{let } a = 5 \times b \text{ in} \]
\[ \quad \text{let } b = a \times b \text{ in} \]
\[ \quad (q \ 100) \]

What the value of \( e_1 \) in environment \( \rho_1 \) under call-by-value with
- lexical scoping?
- dynamic scoping?

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

\[ \rho_1 = [a \mapsto 1, \ b \mapsto 2] \]
\[ e_1 = \text{let } q = \lambda a. (a + b) \text{ in} \]
\[ \quad \text{let } a = 5 \times b \text{ in} \]
\[ \quad \text{let } b = a \times b \text{ in} \]
\[ \quad (q \ 100) \]

value of \( e_1 \rho_1 \): 102

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

\[ \rho_1 = [a \mapsto 1, \ b \mapsto 2] \]
\[ e_1 = \text{let } q = \lambda a. (a + b) \text{ in} \]
\[ \quad \text{let } a = 5 \times b \text{ in} \]
\[ \quad \text{let } b = a \times b \text{ in} \]
\[ \quad (q \ 100) \]

value of \( e_1 \rho_1 \): 120

Puzzle 2

\[ \rho_1 = [a \mapsto 1, \ b \mapsto 2] \]
\[ e_2 = \text{let } p = \lambda a. (a + b) \text{ in} \]
\[ \quad \text{let } q = \lambda b. (a + (p \ b)) \text{ in} \]
\[ \quad \text{let } a = 10 \text{ in} \]
\[ \quad \text{let } b = 20 \]
\[ \quad \text{in } (q \ 100) \]

What is the value of \( e_2 \) in environment \( \rho_1 \) under call-by-value with
- lexical scoping?
- dynamic scoping?
Puzzle 2(a): Call-by-value, lexical scoping

\[
\begin{array}{|c|c|c|}
\hline
\text{tag} & \text{Environment} & \text{Expression} \\
\hline
\rho_1: & \begin{cases} 
  a \mapsto 1 \\
  b \mapsto 2 
\end{cases} & \text{let } p = \ldots \\
\hline
\rho_2: & \begin{cases} 
  p \mapsto (\lambda a.(a+b))\rho_1 
\end{cases} & \text{let } q = \ldots \\
\hline
\rho_3: & \begin{cases} 
  q \mapsto (\lambda b.(a+(p\ b)))\rho_2 
\end{cases} & \text{let } a = \ldots \\
\hline
\rho_4: & a \mapsto 10 & \text{let } b = \ldots \\
\hline
\rho_5: & b \mapsto 20 & (q\ 100) \\
\hline
\rho_6: & b \mapsto 100 & a + (p\ b) \\
\hline
\rho_7: & a \mapsto 100 & (a + b) \\
\hline
\end{array}
\]

value of \(e_2\rho_1\): \(1+(100+2) = 103\)

Puzzle 2(b): Call-by-value, dynamic scoping

\[
\begin{array}{|c|c|c|}
\hline
\text{tag} & \text{Environment} & \text{Expression} \\
\hline
\rho_1: & \begin{cases} 
  a \mapsto 1 \\
  b \mapsto 2 
\end{cases} & \text{let } p = \ldots \\
\hline
\rho_2: & \begin{cases} 
  p \mapsto (\lambda a.(a+b))\rho_1 
\end{cases} & \text{let } q = \ldots \\
\hline
\rho_3: & \begin{cases} 
  q \mapsto (\lambda b.(a+(p\ b)))\rho_2 
\end{cases} & \text{let } a = \ldots \\
\hline
\rho_4: & a \mapsto 10 & \text{let } b = \ldots \\
\hline
\rho_5: & b \mapsto 20 & (q\ 100) \\
\hline
\rho_6: & b \mapsto 100 & a + (p\ b) \\
\hline
\rho_7: & a \mapsto 100 & (a + b) \\
\hline
\end{array}
\]

value of \(e_2\rho_1\): \(10+(100+100) = 210\)

Lexical Scoping, 4: Closures + States = Objects

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

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

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

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

\[
\begin{align*}
\text{function } mbx(x) &= \{ \\
\text{ var } bx &= (\text{new } x) \\
\text{ return } (\text{function } foo(v) \\
\{ bx : = bx + v; \text{ return } bx \}) \\
\}
\end{align*}
\]

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

Call by name

\[
\begin{align*}
\text{Subst-App-cbn: } & \langle E_1, s \rangle \Downarrow_N \langle \lambda x. E'_1, s' \rangle \\
& \langle E'_1[\text{E}_2 / x], s' \rangle \Downarrow_N \langle V, s'' \rangle \\
& \langle (E_1\ E_2), s \rangle \Downarrow_N \langle V, s'' \rangle \\
\end{align*}
\]

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

**Answer:**
Thunks \(\equiv\) closures of arbitrary expressions, not just \(\lambda\)-expressions.

Lexical Scoping, 6

The Call-By-Name Version

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

\[ \frac{\rho \vdash (e_1, s) \downarrow_N (v, s'')} {\rho \vdash (e_1, s) \downarrow_N (v, s'')} \]

Let: \[ (e, s) \downarrow_N (v, s') \]

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

Puzzle 3

Consider \[ \rho_0 \vdash (e_0, s_0) \downarrow_N (v_1, s_1) \].

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

Puzzle 3(a): Call-by-value

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

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

\[ \rho_0 \vdash (e_0, s_0) \downarrow_N (v_1, s_1) ? \]

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

Puzzle 3(b): Call-by-name

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

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

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

- call-by-value evaluation?
- call-by-name evaluation?

Puzzle 4(b): Call-by-name

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

$$\text{unfolding}_{\text{subst}}: \langle E[(\text{rec } x.E)/x], s \rangle \downarrow \langle V, s' \rangle$$

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

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

An environment-based version of unfold  
(There are better ways!)

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

Try:

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