CS450

Structure of Higher Level Languages

Lecture 14: Mutable environment semantics and frames

Tiago Cogumbreiro

A quick recap...

- We introduced λ_D-Racket, to capture the semantics of `define`
- We realized λ_D-Racket could not represent out-of-order definitions.
- We encoded a shared mutable heap

Today we will...

- Introduce the semantics of λ-calculus with environments
- Study mutation as a side-effect
- Introduce mutable environments, composed of frames
- Implement frames

> Section 3.2 of the SICP book. The interactive version of Section 3.2.

Introducing the $\lambda^{\star}_{D}$-calculus

$\lambda_D^\star$-calculus: λ-calculus with definitions

We highlight in **red** an operation that produces a side effect: *mutating an environment*.

$$\frac{e \Downarrow_E v \qquad E \leftarrow [x := v]}{(\texttt{define}\ x\ e) \Downarrow_E \texttt{void}}\ \texttt{(E-def)}$$

$$\frac{t_1 \Downarrow_E v_1 \qquad t_2 \Downarrow_E v_2}{t_1; t_2 \Downarrow_E v_2}\ \texttt{(E-seq)}$$

$\lambda_D^\star$-calculus: λ-calculus with definitions

- Because we have side-effects, the order in which we evaluate each sub-expression is important.

$$v \Downarrow_E v \qquad (\texttt{E-val})$$

$$x \Downarrow_E E(x) \qquad (\texttt{E-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \qquad (\texttt{E-lam})$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \qquad e_a \Downarrow_E v_a \qquad E_b \leftarrow E_f + [x := v_a] \qquad t_b \Downarrow_{E_b} v_b}{(e_f\ e_a) \Downarrow_E v_b}\ (\texttt{E-app})$$

- Can you explain why the order is important?

$\lambda_D^\star$-calculus: λ-calculus with definitions

Because we have side-effects, the order in which we evaluate each sub-expression is important.

$$v \Downarrow_E v \qquad (\texttt{E-val})$$

$$x \Downarrow_E E(x) \qquad (\texttt{E-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \qquad (\texttt{E-lam})$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \qquad e_a \Downarrow_E v_a \qquad \color{red}{E_b \leftarrow E_f + [x := v_a]} \qquad t_b \Downarrow_{E_b} v_b}{(e_f\ e_a) \Downarrow_E v_b} \ (\texttt{E-app})$$

Can you explain why the order is important? Otherwise, we might evaluate the body of the function e_b without observing the assignment $x := v_a$ in E_b.

Mutable operations on environments

Mutable operations on environments

Put

$$E \leftarrow [x := v]$$

Take a reference to an environment E and mutate its contents, by adding a new binding.

Push

$$E \leftarrow E' + [x := v]$$

Create a new environment referenced by E which copies the elements of E' and also adds a new binding.

Making side-effects explicit

Mutation as a side-effect

Let us use a triangle ▶ to represent the order of side-effects.

$$\frac{e \Downarrow_E v \quad \blacktriangleright \quad E \leftarrow [x := v]}{(\texttt{define}\ x\ e) \Downarrow_E \texttt{void}}\ (\texttt{E-def})$$

$$\frac{t_1 \Downarrow_E v_1 \quad \blacktriangleright \quad t_2 \Downarrow_E v_2}{t_1; t_2 \Downarrow_E v_2}\ (\texttt{E-seq})$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \ \blacktriangleright\ e_a \Downarrow_E v_a \ \blacktriangleright\ E_b \leftarrow E_f + [x := v_a] \ \blacktriangleright\ t_b \Downarrow_{E_b} v_b}{(e_f\ e_a) \Downarrow_E v_b}\ (\texttt{E-app})$$

Implementing side-effect mutation

Making the heap explicit

> We can annotate each triangle with a heap, to make explicit which how the global heap should be passed from one operation to the next. In this example, defining a variable takes an input global heap H and produces an output global heap H_2.

$$\frac{\blacktriangleright_{H} \quad e \Downarrow_E v \quad \blacktriangleright_{H_1} \quad \color{red}{E \leftarrow [x := v]} \quad \blacktriangleright_{H_2}}{\blacktriangleright_{H} \ (\texttt{define}\ x\ e) \Downarrow_E \texttt{void} \blacktriangleright_{H_2}} \ \texttt{(E-def)}$$

Let us use our rule sheet!

$$\frac{e \Downarrow_E v \quad \blacktriangleright \quad E \leftarrow [x := v]}{(\texttt{define}\ x\ e) \Downarrow_E \texttt{void}}\ (\texttt{E-def})$$

$$\frac{t_1 \Downarrow_E v_1 \quad \blacktriangleright \quad t_2 \Downarrow_E v_2}{t_1 ; t_2 \Downarrow_E v_2}\ (\texttt{E-seq})$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \quad \blacktriangleright \quad e_a \Downarrow_E v_a \quad \blacktriangleright \quad E_b \leftarrow E_f + [x := v_a] \quad \blacktriangleright \quad t_b \Downarrow_{E_b} v_b}{(e_f\ e_a) \Downarrow_E v_b}\ (\texttt{E-app})$$

$$v \Downarrow_E v \qquad (\texttt{E-val})$$

$$x \Downarrow_E E(x) \qquad (\texttt{E-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \qquad (\texttt{E-lam})$$

Examples

Evaluating Example 2 of Lecture 13

```
(define b (lambda (x) a))
(define a 20)
(b 1)
```

Input

```
E0: []
Term: (define b (lambda (y) a))
```

Evaluating Example 2 of Lecture 13

```
(define b (lambda (x) a))
(define a 20)
(b 1)
```

Input

```
E0: []
Term: (define b (lambda (y) a))
```

Output

```
E0: [
  (b . (closure E0 (lambda (y) a)))
]
Expression: #<void>
```

$$\dfrac{\dfrac{}{\lambda y.a \Downarrow_{E_0} (E_0, \lambda y.a)} \quad \blacktriangleright \quad \dfrac{}{E_0 \leftarrow [b := (E_1, \lambda y.a)]}}{(\texttt{define}\ b\ \lambda y.a) \Downarrow_{E_0} \texttt{void}}$$

Example 2: step 2

Input

```
E0: [
  (b . (closure E0 (lambda (y) a)))
]
Term: (define a 20)
```

Example 2: step 2

Input

```
E0: [
  (b . (closure E0 (lambda (y) a)))
]
Term: (define a 20)
```

Output

```
E0: [
  (a . 20)
  (b . (closure E0 (lambda (y) a)))
]
Expression: #<void>
```

$$\dfrac{\dfrac{}{20 \Downarrow_{E_0} 20} \quad \blacktriangleright \quad \dfrac{}{E_0 \leftarrow [a := 20]}}{(\mathbf{define}\ a\ 20) \Downarrow_{E_0} \mathtt{void}}$$

Example 2: step 3

Input

```
E0: [
  (a . 20)
  (b . (closure E0 (lambda (y) a)))
]
Term: (b 1)
```

Example 2: step 3

Input

```
E0: [
  (a . 20)
  (b . (closure E0 (lambda (y) a)))
]
Term: (b 1)
```

Output

```
E0: [
  (a . 20)
  (b . (closure E0 (lambda (y) a)))
]
E1: [ E0
  (y . 1)
]
Expression: 20
```

$$\frac{b \Downarrow_{E_0} (E_0, \lambda y.a) \;\blacktriangleright\; 1 \Downarrow_{E_0} 1 \;\blacktriangleright\; E_1 \leftarrow E_0 + [y := 1] \;\blacktriangleright\; a \Downarrow_{E_1} 20}{(b\ 1) \Downarrow_{E_0} 20}$$

Example 3

```
(define (f x) (lambda (y) x))
(f 10)
```

Input

```
E0: []
Term: (define (f x) (lambda (y) x))
```

Example 3

```
(define (f x) (lambda (y) x))
(f 10)
```

Input

```
E0: []
Term: (define (f x) (lambda (y) x))
```

Output

```
E0: [
  (f . (closure E0
           (lambda (x) (lambda (y) x))))
]
Value: void
```

Example 3

```
(define (f x) (lambda (y) x))
(f 10)
```

Input

```
E0: []
Term: (define (f x) (lambda (y) x))
```

Output

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
Value: void
```

$$\frac{\lambda x.\lambda y.x \Downarrow_{E_0} (E_0, \lambda x.\lambda y.x)}{(\texttt{define}\ f\ \lambda x.\lambda y.x) \Downarrow_{E_0} \texttt{void}}$$

Example 3

```
(define (f x) (lambda (y) x))
(f 10)
```

Input

```
E0: []
Term: (define (f x) (lambda (y) x))
```

Output

```
E0: [
  (f . (closure E0
            (lambda (x) (lambda (y) x))))
]
Value: void
```

$$\frac{\lambda x.\lambda y.x \Downarrow_{E_0} (E_0, \lambda x.\lambda y.x) \qquad \blacktriangleright \qquad E_0 \leftarrow [f := (E_0, \lambda x.\lambda y.x)]}{(\texttt{define}\ f\ \lambda x.\lambda y.x) \Downarrow_{E_0} \texttt{void}}$$

Example 3

Input

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
Term: (f 10)
```

Example 3

Input

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
Term: (f 10)
```

Output

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
E1: [ E0 (x . 10) ]
Value: (closure E1 (lambda (y) x))
```

Example 3

Input

```
E0: [
  (f . (closure E0
         (lambda (x) (lambda (y) x))))
]
Term: (f 10)
```

Output

```
E0: [
  (f . (closure E0
         (lambda (x) (lambda (y) x))))
]
E1: [ E0 (x . 10) ]
Value: (closure E1 (lambda (y) x))
```

$$\frac{\dfrac{E_0(f) = (E_0, \lambda x.\lambda y.x)}{f \Downarrow_{E_0} (E_0, \lambda x.\lambda y.x)} \quad \dfrac{}{10 \Downarrow_{E_0} 10} \quad \dfrac{}{E_1 \leftarrow E_0 + [x := 10]} \quad \dfrac{}{\lambda y.x \Downarrow_{E_1} (E_1, \lambda y.x)}}{(f\ 10) \Downarrow_{E_0} (E_1, \lambda y.x)}$$

Visualizing the environment

Environment visualization

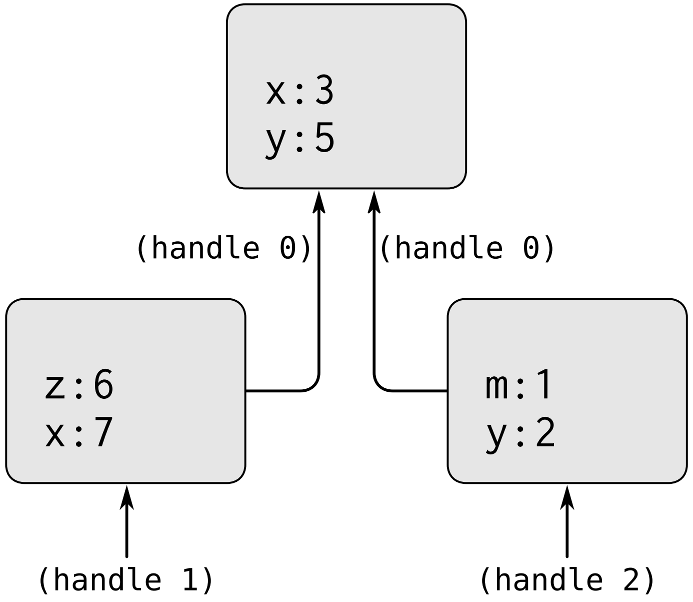

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

```
; E0 = (handle 0)
E0: [
  (x . 3)
  (y . 5)
]
; E1 = (handle 1)
E1:  [ E0
  (z . 6)
  (x . 7) ; shadows E0.x
  ; (y . 5)
]
; E2 = (handle 2)
E2:  [ E0
  (m . 1)
  (y . 2) ; shadows E0.y
  ; (x . 3)
]
```

Environment visualization

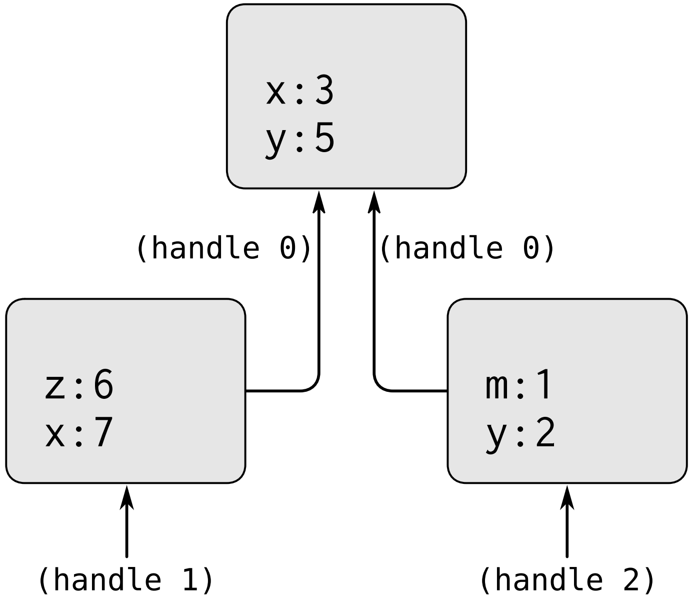

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

The heap at runtime

- arrows are *references*, or heap handles:
- boxes are *frames*: labelled by their handles
- each frame has local variable bindings (eg, `m:1`, and `y:2`)

Environment visualization

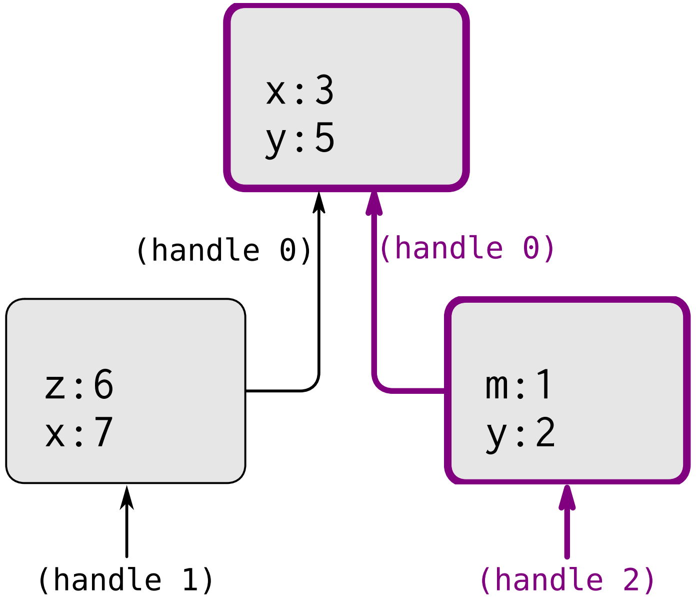

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

The heap at runtime

- arrows are *references*, or heap handles:
- boxes are *frames*: labelled by their handles
- each frame has local variable bindings (eg, `m:1`, and `y:2`)
- an *environment* represents a *sequence of frames*, connected via references. For instance, the environment that consists of frame 3 linked to frame 1.
- variable lookup follows the reference order. For instance, lookup a variable in frame 3 and then in frame 1.

Quiz

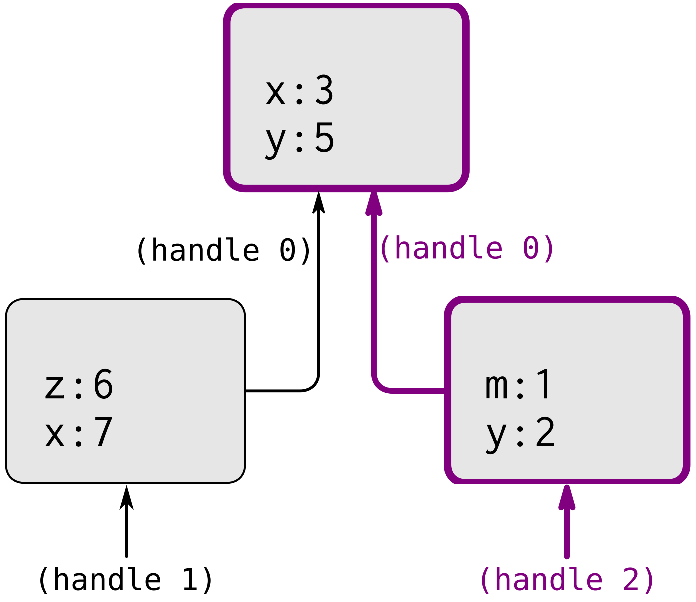

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

List all variable bindings in environment `(handle 1)`

Please, write down your answer, your email address, and leave your paper in my desk before your leave.

> This quiz counts toward your attendance grade.

Implementing mutable environments

Implementing mutable environments

Heap

- A heap contains *frames*

Frame

- a reference to its parent frame (except for the root frame which does not refer any other frame)
- a map of local bindings

```
E0: [
  (a . 20)
  (b . (closure E0 (lambda (y) a)))
]
E1:  [ E0
 (y . 1)
]
```

Example of a frame: `[ E0 (y . 1) ]`
Example of a root frame: `[ (a . 20) (b . (closure E0 (lambda (y) a)) ]`

Let us implement frames...

(demo time)

Usage examples

```
; (closure E0 (lambda (y) a)
(define c (s:closure (handle 0) (s:lambda (list (s:variable 'y)) (s:variable 'a))))
;E0: [
;   (a . 20)
;   (b . (closure E0 (lambda (y) a)))
;]
(define f1
  (frame-put
    (frame-put root-frame (s:variable 'a) (s:number 10))
    (s:variable 'b) c))
(check-equal? f1 (frame #f (hash (s:variable 'a) (s:number 10) (s:variable 'b) c)))
; Lookup a
(check-equal? (s:number 10) (frame-get f1 (s:variable 'a)))
; Lookup b
(check-equal? c (frame-get f1 (s:variable 'b)))
; Lookup c that does not exist
(check-equal? #f (frame-get f1 (s:variable 'c)))
```

More usage examples

```
; E1:  [ E0
;   (y . 1)
; ]
(define f2 (frame-push (handle 0) (s:variable 'y) (s:number 1)))
(check-equal? f2 (frame (handle 0) (hash (s:variable 'y) (s:number 1))))
(check-equal? (s:number 1) (frame-get f2 (s:variable 'y)))
(check-equal? #f (frame-get f2 (s:variable 'a)))
;; We can use frame-parse to build frames
(check-equal? (parse-frame '[ (a . 10) (b . (closure E0 (lambda (y) a)))]) f1)
(check-equal? (parse-frame '[ E0 (y . 1) ]) f2))
```

Frames

```
(struct frame (parent locals))
```

- `parent` is either `#f` or is a reference to the parent frame
- `locals` is a hash-table with the local variables of this frame

Constructors

```
(struct frame (parent locals) #:transparent)
(define root-frame (frame #f (hash)))
(define (frame-push parent var val)
  (frame parent (hash var val)))
(define (frame-put frm var val)
  (frame (frame-parent frm)
         (hash-set (frame-locals frm) var val)))
(define (frame-get frm var)
  (hash-ref (frame-locals frm) var #f))
```

Description

- `root-frame` creates an orphan empty frame (hence `#f`). This function is needed to represent the top-level environment.
- `frame-push` takes a reference that points to the parent frame, and initializes a hash-table with one entry (`var`, `val`). This function is needed for $E \leftarrow E' + [x := v]$
- `frame-put` updates the current frame with a new binding. This function is needed for $E \leftarrow [x := v]$