CS450

Structure of Higher Level Languages

Lecture 15: Language λ_D

Tiago Cogumbreiro

Visualizing the environment

Environment visualization

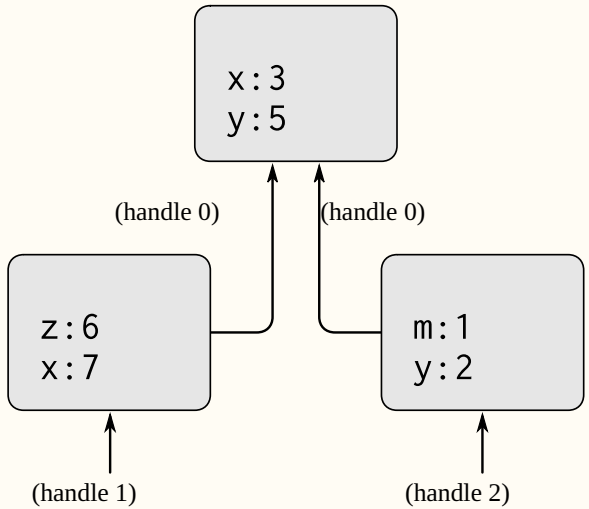

Figure 3.1: A simple environment structure.

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)
]
```

UMass Boston

Environment visualization

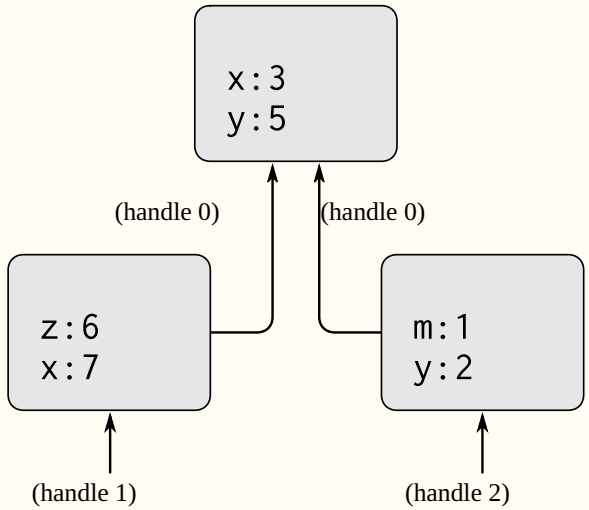

Figure 3.1: A simple environment structure.

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

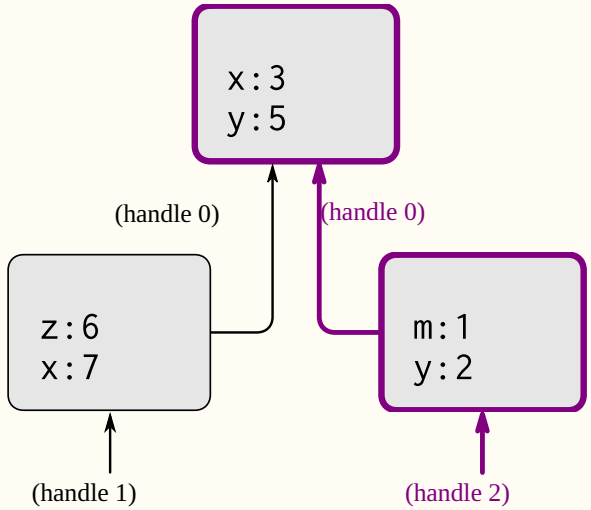

Figure 3.1: A simple environment structure.

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.

UMass Boston

Quiz

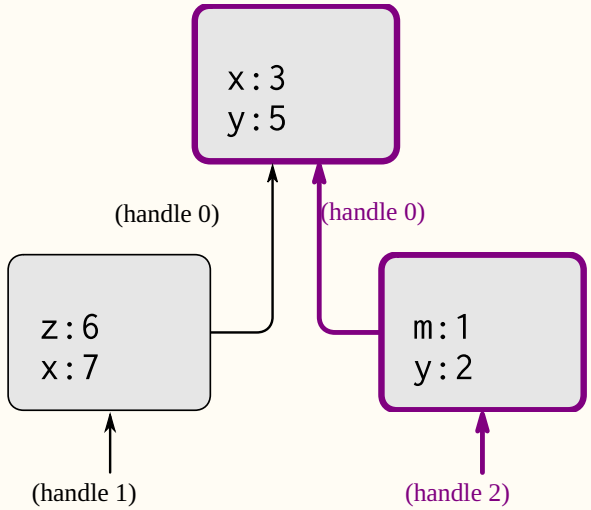

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

List all variable bindings

in environment `(handle 1)`

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)) ]`

UMass Boston

Let us implement frames...

(demo time)

Usage examples

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

More usage examples

```
; E1:  [ E0
;  (y . 1)
; ]
(define f2 (frame-push (handle 0) (d:variable 'y) (d:number 1)))
(check-equal? f2 (frame (handle 0) (hash (d:variable 'y) (d:number 1))))
(check-equal? (d:number 1) (frame-get f2 (d:variable 'y)))
(check-equal? #f (frame-get f2 (d: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 $E \leftarrow [x := v]$

Summary

Today we implement a mutable environment.

Constructors

- **Empty**: The empty, root environment.
- **Put**: $E \leftarrow [x := v]$ updates an existing environment E upon defining a variable. Returns the same frame, and updates the heap.
- **Push**: $E_2 \leftarrow E_1 + [x := v]$ creates a new environment E_2 by extending environment E_1 with one binding $x = v$. Returns the new environment.

Selectors

- **Variable Lookup**: $E(x)$ Looks up variable x in the bindings of the current frame, otherwise recursively looks up the parent frame.

UMass Boston

Environment example

Environment visualization

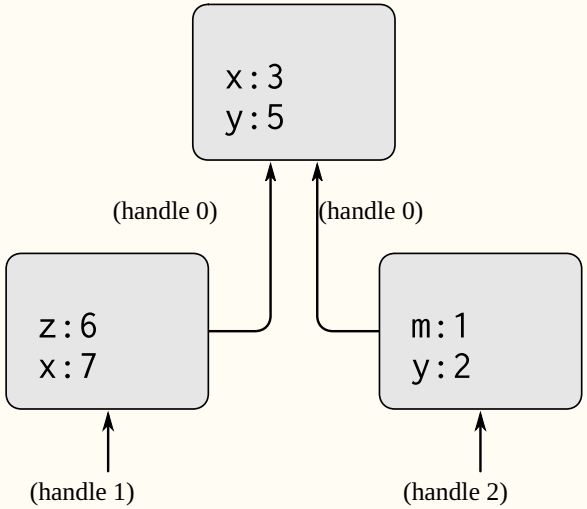

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

Environment operations

Environment example

Environment visualization

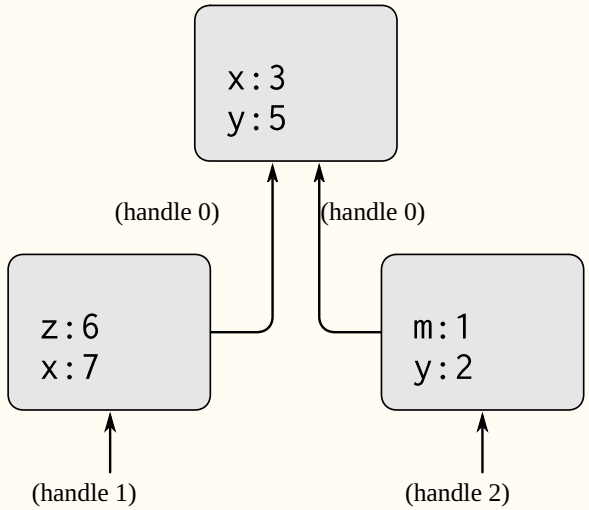

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

Environment operations

```
E0 ← [x := 3]
E0 ← [y := 5]
E1 ← E0 + [z := 6]
E1 ← [x := 7]
E2 ← E0 + [m := 1]
E2 ← [y := 2]
```

Constructors: Root

The root environment

```
(define root-alloc (heap-alloc empty-heap root-frame))
(define root-environ (eff-result root-alloc))
(define root-mem (eff-state root-alloc))
```

Constructors: Put

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

```
(define (environ-put mem env var val)
  (define new-frm (frame-put (heap-get mem env) var val))
  (heap-put mem env new-frm))
```

Example

```
E0 ← [x := 3]
E0 ← [y := 5]
```

In Racket

Constructors: Put

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

```
(define (environ-put mem env var val)
  (define new-frm (frame-put (heap-get mem env) var val))
  (heap-put mem env new-frm))
```

Example

```
E0 ← [x := 3]
E0 ← [y := 5]
```

In Racket

```
(define E0 root-environ)
(define m1
  (environ-put
    (environ-put root-heap E0 (d:variable 'x) (d:number 3))
    E0 (d:variable 'y) (d:number 5)))
```

Constructors: Push

$$E_2 \leftarrow E_1 + [x := v]$$

```
(define (environ-push mem env var val)
  (define new-frame (frame env (hash var val)))
  (heap-alloc mem new-frame))
```

Example In Racket

```
E1 ← E0 + [z := 6]
E1 ← [x := 7]
```

Constructors: Push

$$E_2 \leftarrow E_1 + [x := v]$$

```
(define (environ-push mem env var val)
  (define new-frame (frame env (hash var val)))
  (heap-alloc mem new-frame))
```

Example

```
E1 ← E0 + [z := 6]
E1 ← [x := 7]
```

In Racket

```
(define e1-m2 (environ-push m1 E0 (d:variable 'z) (d:number 6)))
(define E1 (eff-result e1-m2))
(define m2 (eff-state e1-m2))
(define m3 (environ-put m2 E1 (d:variable 'x) (d:number 7)))
```

Continuing the example

Example

```
E0 ← [x := 3]
E0 ← [y := 5]
E1 ← E0 + [z := 6]
E1 ← [x := 7]
E2 ← E0 + [m := 1]
E2 ← [y := 2]
```

In Racket

Continuing the example

Example

```
E0 ← [x := 3]
E0 ← [y := 5]
E1 ← E0 + [z := 6]
E1 ← [x := 7]
E2 ← E0 + [m := 1]
E2 ← [y := 2]
```

In Racket

```
(define E0 root-environ)
(define m1
  (environ-put
    (environ-put root-heap E0 (d:variable 'x) (d:number 3))
    E0 (d:variable 'y) (d:number 5)))
(define e1-m2 (environ-push m1 E0 (d:variable 'z) (d:number 6)))
(define E1 (eff-result e1-m2))
(define m2 (eff-state e1-m2))
(define m3 (environ-put m2 E1 (d:variable 'x) (d:number 7)))
(define e2-m4 (environ-push m3 E0 (d:variable 'm) (d:number 1)))
(define E2 (eff-result e2-m4))
(define m4 (eff-state e2-m4))
(define m5 (environ-put m4 E2 (d:variable 'y) (d:number 2)))
```

Selector: Variable lookup

$$E(x)$$

```
(define (environ-get mem env var)
  (define frm (heap-get mem env))    ;; Load the current frame
  (define parent (frame-parent frm))  ;; Load the parent
  (define result (frame-get frm var)) ;; Lookup locally
  (cond
    [result result] ;; Result is defined, then return it
    [parent (environ-get mem parent var)] ; If parent exists, recurse
    [else (error (format "Variable ~a is not defined" var))]))
```

Example

```
(check-equal? (environ-get m5 E2 (d:variable 'y)) (d:number 2))
(check-equal? (environ-get m5 E2 (d:variable 'm)) (d:number 1))
(check-equal? (environ-get m5 E2 (d:variable 'x)) (d:number 3)))
```

A language of environments

Environment visualization

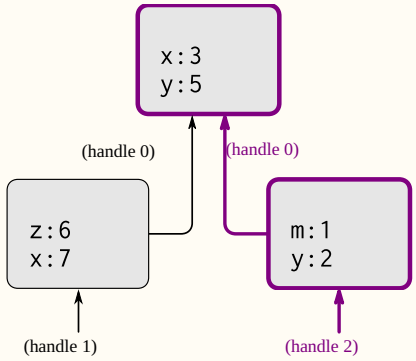

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

```racket
(define parsed-m5
   (parse-mem
     '([E0 . ([x . 3] [y . 5])]
       [E1 . (E0 [x . 7] [z . 6])]
       [E2 . (E0 [m . 1] [y . 2])])))
; Which is the same as creating the following data-structure
(heap
  (hash
    (handle 0)
    (frame #f
      (hash (d:variable 'y) (d:number 5) (d:variable 'x) (d:number 3)))
    (handle 2)
    (frame (handle 0)
      (hash (d:variable 'y) (d:number 2) (d:variable 'm) (d:number 1)))
    (handle 1)
    (frame (handle 0)
      (hash (d:variable 'z) (d:number 6) (d:variable 'x) (d:number 7)))

(check-equal? parsed-m5 m5)
```

UMass Boston