CS450

Structure of Higher Level Languages

Lecture 17: λ_D

Tiago Cogumbreiro

Press arrow keys ← → to change slides.

Shared "mutable" state

with immutable data-structures

Why immutability?

Benefits

- A necessity if we use a language without mutation (such as Haskell)
- Parallelism: A great way to implement fast and safe data-structures in concurrent code (look up copy-on-write)
- Development: Controlled mutation improves code maintainability
- Memory management: counters the problem of circular references (notably, useful in C++ and Rust, see example)

> Encoding shared mutable state with immutable data-structures is a great skill to have.

UMass Boston

Heap

> We want to design a data-structure that represents a ***heap*** (a shared memory buffer) that allows us to: ***allocate*** a new memory cell, ***load*** the contents of a memory cell, and ***update*** the contents of a memory cell.

Constructors

- `empty-heap` returns an empty heap
- `(heap-alloc h v)` creates a new memory cell in heap `h` whose contents are value `v`
- `(heap-put h r v)` updates the contents of memory handle `r` with value `v` in heap `h`

Selectors

- `(heap-get h r)` returns the contents of memory handle `r` in heap `h`

Heap usage

```
(define h empty-heap)          ; h is an empty heap
(define r (heap-alloc h "foo")) ; stores "foo" in a new memory cell
```

> What should the return value of `heap-alloc`?

- Should `heap-alloc` return a copy of `h` extended with "foo"? How do we access the memory cell pointing to "foo"?
- Should `heap-alloc` return a handle to the new memory cell? How can we access the new heap?

Heap usage

```
(define h empty-heap)           ; h is an empty heap
(define r (heap-alloc h "foo")) ; stores "foo" in a new memory cell
```

> What should the return value of `heap-alloc`?

- Should `heap-alloc` return a copy of `h` extended with "foo"? How do we access the memory cell pointing to "foo"?
- Should `heap-alloc` return a handle to the new memory cell? How can we access the new heap?

> Function `heap-alloc` must return a ***pair*** `eff` that contains the new heap and the memory handle.

```
(struct eff (state result) #:transparent)
```

Heap usage example

Spec

```
(define h1 empty-heap)            ; h is an empty heap
(define r (heap-alloc h1 "foo")) ; stores "foo" in a new memory cell
(define h2 (eff-state r))
(define x (eff-result r)) ;
(check-equal? "foo" (heap-get h2 x)) ; checks that "foo" is in x
(define h3 (heap-put h2 x "bar"))     ; stores "bar" in x
(check-equal? "bar" (heap-get h3 x)) ; checks that "bar" is in x
```

Handles must be unique

We want to ensure that the handles we create are **unique**, otherwise allocation could overwrite existing data, which is undesirable.

Spec

```
(define h1 empty-heap)          ; h is an empty heap
(define r1 (heap-alloc h1 "foo")) ; stores "foo" in a new memory cell
(define h2 (eff-state r1))
(define x (eff-result r1))
(define r2 (heap-alloc h2 "bar")) ; stores "foo" in a new memory cell
(define h3 (eff-state r2))
(define y (eff-result r2))
(check-not-equal? x y)  ; Ensures that x != y
(check-equal? "foo" (heap-get h3 x))
(check-equal? "bar" (heap-get h3 y))
```

How can we implement a memory handle?

A simple heap implementation

- Let a handle be an integer
- Recall that the heap only grows (no deletions)
- A handle matches the number of elements already present in the heap
- When the heap is empty, the first handle is 0, the second handle is 1, and so on.

Heap implementation

- We use a hash-table to represent the heap because it has a faster random-access than a linked-list (where lookup is linear on the size of the list).
- We wrap the hash-table in a struct, and the handle (which is a number) in a struct, for better error messages. And because it helps maintaining the code.

```
(struct heap (data) #:transparent)
(define empty-heap (heap (hash)))
(struct handle (id) #:transparent)
(struct eff (state result) #:transparent)
(define (heap-alloc h v)
  (define data (heap-data h))
  (define new-id (handle (hash-count data)))
  (define new-heap (heap (hash-set data new-id v)))
  (eff new-heap new-id))
(define (heap-get h k)
  (hash-ref (heap-data h) k))
(define (heap-put h k v)
  (define data (heap-data h))
  (cond
    [(hash-has-key? data k) (heap (hash-set data k v))]
    [else (error "Unknown handle!")]))
```

Visualizing the environment

Environment visualization

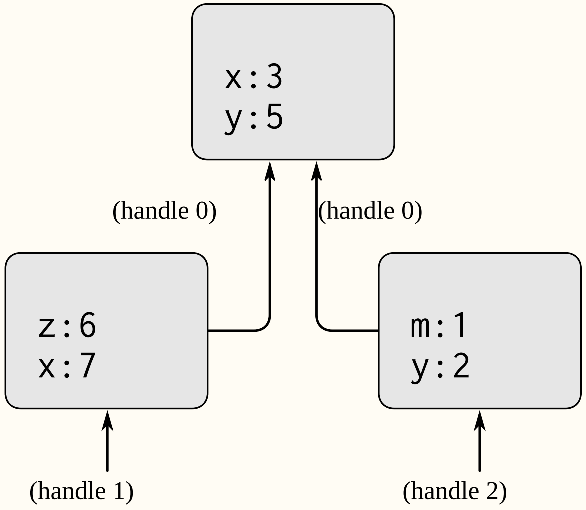

Figure 3.1: A simple environment structure.

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

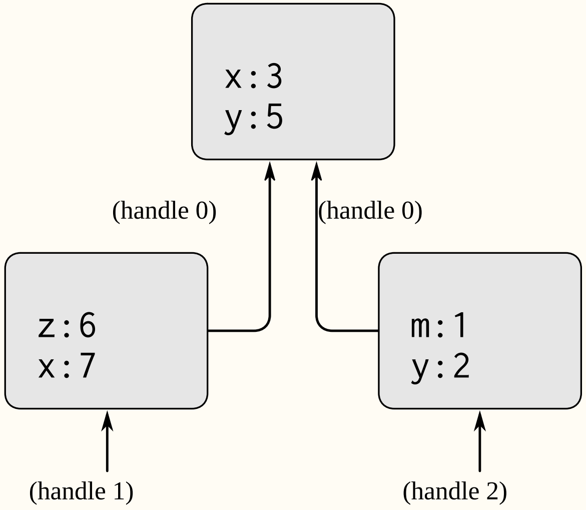

Figure 3.1: A simple environment structure.

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

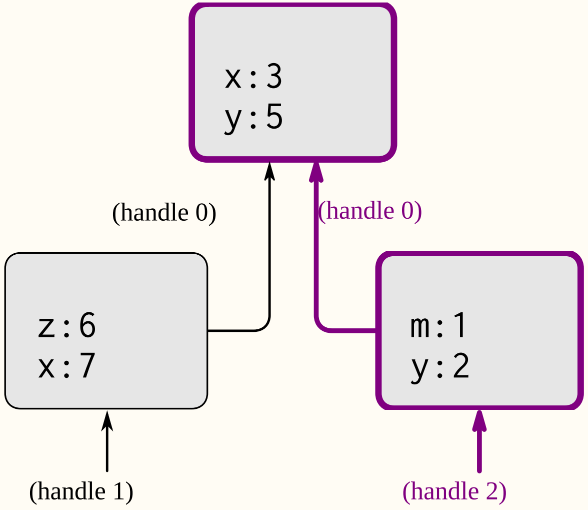

Figure 3.1: A simple environment structure.

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.

UMass Boston

Quiz

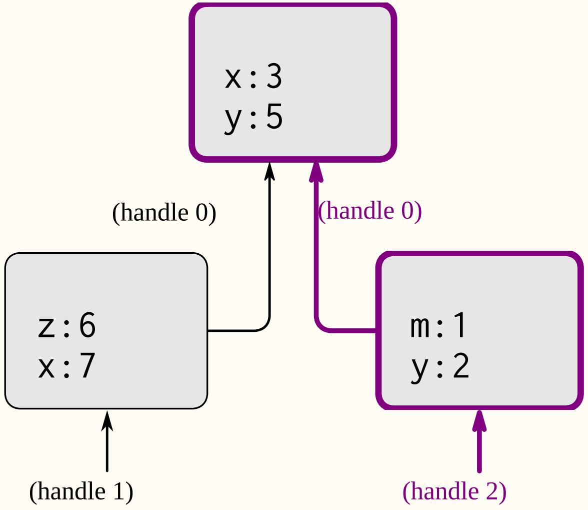

Figure 3.1: A simple environment structure.

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)) ]`

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.

Environment example

Environment visualization

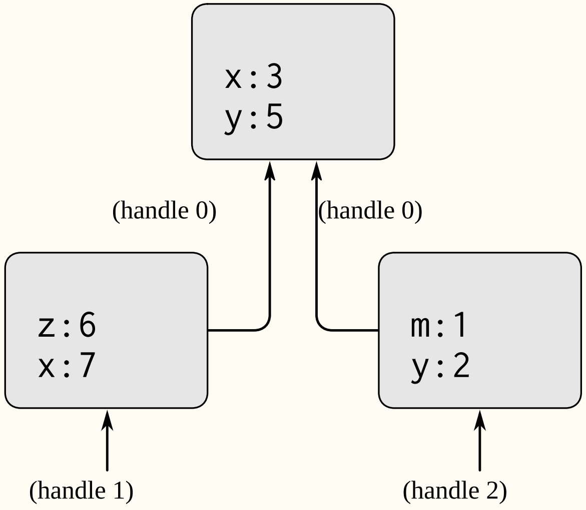

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

Environment operations

Environment example

Environment visualization

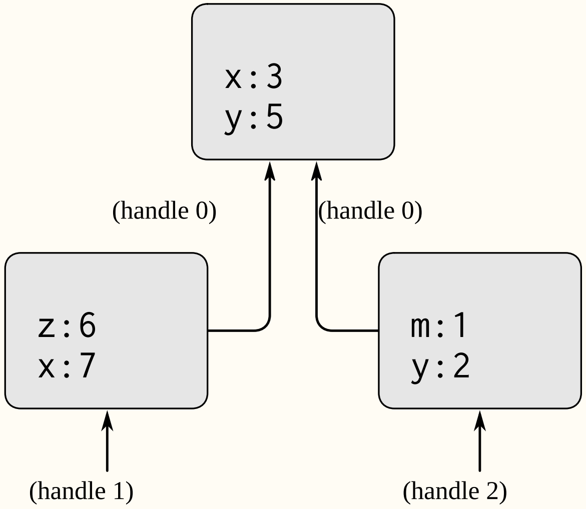

Figure 3.1: A simple environment structure.

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)))
```

UMass Boston

A language of environments

Environment visualization

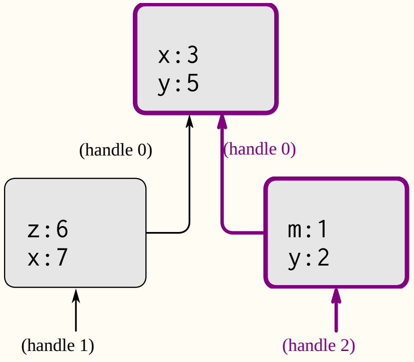

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2

```
(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)
```

$$\frac{\blacktriangleright_{H_1}\ e \Downarrow_E v \quad \blacktriangleright_{H_2} \quad E \leftarrow [x := v]\ \blacktriangleright_{H_3}}{\blacktriangleright_{H_1}\ (\texttt{define}\ x\ e) \Downarrow_E \texttt{void}\ \blacktriangleright_{H_3}}$$

$$\frac{\blacktriangleright_{H_1}\ t_1 \Downarrow_E v_1 \quad \blacktriangleright_{H_2} \quad t_2 \Downarrow_E v_2\ \blacktriangleright_{H_3}}{t_1 ; t_2 \Downarrow_E v_2}$$

$$\blacktriangleright_{H}\ v \Downarrow_E v\ \blacktriangleright_{H}$$

$$\blacktriangleright_{H}\ x \Downarrow_E E(x)\ \blacktriangleright_{H}$$

$$\blacktriangleright_{H}\ \lambda x.t \Downarrow_E (E, \lambda x.t)\ \blacktriangleright_{H}$$

$$\frac{\blacktriangleright_{H_1}\ e_f \Downarrow_E (E_f, \lambda x.t_b)\ \blacktriangleright_{H_2}\ \ e_a \Downarrow_E v_a\ \blacktriangleright_{H_3}\ \ E_b \leftarrow E_f + [x := v_a]\ \blacktriangleright_{H_4}\ \ t_b \Downarrow_{E_b} v_b\ \blacktriangleright_{H_5}}{\blacktriangleright_{H_1}\ (e_f\ e_a) \Downarrow_E v_b\ \blacktriangleright_{H_5}}$$

Notes

- Make sure `(d:eval-term)` handles expressions by calling `(d:eval-exp)`
- Make sure the case for `d:define?` returns the value `(d:void)` not `(void)`, not `d:void`, not `void`
- Make sure the case for `d:apply?` invokes `(d:eval-term)` when handling t_b

Exercise

```
;; e1 ⇓E v1


;; E' <- E + [x := v1]


;; e2 ⇓E' v2
```

Exercise

```
;; e1 ⇓E v1
(define v1+mem1 (d:eval-exp mem env e1))
(define mem1 (eff-state v1+mem1))
(define v1 (eff-result v1+mem1))
;; E' <- E + [x := v1]
(define env2+mem2 (environ-push mem1 env y v1))
(define env2 (eff-result env2+mem2))
(define mem2 (eff-state env2+mem2))
;; e2 ⇓E' v2
(define v2+mem3 (d:eval-exp mem2 env2 e2))
(define mem3 (eff-state v2+mem3))
(define v2 (eff-result v2+mem3))
```