

# CS450

## Structure of Higher Level Languages

Lecture 23: Language  $\lambda_D$ : adding definitions correctly

Tiago Cogumbreiro

# Today we will...

- Introduce the **mutable** semantics of  $\lambda$ -calculus with environments
- Study mutation as a side-effect

Introducing the  $\lambda_D$

# Language $\lambda_D$ : Terms

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

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

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

# Language $\lambda_D$ : Expressions

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

$$v \Downarrow_E v \quad (\mathbf{E}\text{-val})$$

$$x \Downarrow_E E(x) \quad (\mathbf{E}\text{-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \quad (\mathbf{E}\text{-lam})$$

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

Can you explain why the order is important?

# Language $\lambda_D$ : Expressions

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

$$v \Downarrow_E v \quad (\mathbf{E}\text{-val})$$

$$x \Downarrow_E E(x) \quad (\mathbf{E}\text{-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \quad (\mathbf{E}\text{-lam})$$

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

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  $\blacktriangleright$  to represent the order of side-effects.

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

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

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

# 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  $H_2$ .

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

# Let us use our rule sheet!

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

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

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

$$v \Downarrow_E v \quad \text{ (E-val)}$$

$$x \Downarrow_E E(x) \quad \text{ (E-var)}$$

$$\lambda x. t \Downarrow_E (E, \lambda x. t) \quad \text{ (E-lam)}$$

# Examples

# Evaluating Example 2

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

Input

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

# Evaluating Example 2

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

Input

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

Output

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

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

# Example 2: step 2

Input

```

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



# Example 2: step 2

Input

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

Output

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

$$\frac{\overline{20 \Downarrow_{E_0} 20} \quad \blacktriangleright \quad \overline{E_0 \leftarrow [a := 20]}}{\overline{(\text{define } a \ 20) \Downarrow_{E_0} \text{void}}}$$

# Example 2: step 3

Input

```

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

```

---

Env: E0

Term: (b 1)

# Example 2: step 3

Input

```

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

```

Output

```

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

```
---
```

```
Env: E0
```

```
Term: (define (f x) (lambda (y) x))
```

# Example 3

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

Input

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

Output

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

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

# Example 3

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

Input

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

Output

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

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

# Example 3

Input

```

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



# Example 3

Input

```

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

```

Output

```

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

```

$$E_0(f) = (E_0, \lambda x. \lambda y. x)$$

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