

The Environment Model

Concepts of Programming Languages

Outline

- » Discuss the difference between **dynamic** and lexical scoping
- » Introduce **closures** as a way of implementing lexical scoping in the environment model
- » Give example **derivations** using closures
- » Discuss **recursion** and closures
- » Demo an **implementation** of the lambda calculus⁺ let expressions using closures

Variable Scoping

Two Major Concerns

Two Major Concerns

1. Are variables *mutable*? Can we change their values? Are there restrictions to when we can change the value of a variable?

Two Major Concerns

1. Are variables *mutable*? Can we change their values? Are there restrictions to when we can change the value of a variable?
2. How are variables *scoped*? Dynamically or lexically? Does a binding define its own scope? Is it defined in a block?

Two Major Concerns

1. Are variables *mutable*? Can we change their values? Are there restrictions to when we can change the value of a variable?
2. How are variables *scoped*? Dynamically or lexically? Does a binding define its own scope? Is it defined in a block?

OCaml variables are:

Two Major Concerns

1. Are variables *mutable*? Can we change their values? Are there restrictions to when we can change the value of a variable?
2. How are variables *scoped*? Dynamically or lexically? Does a binding define its own scope? Is it defined in a block?

OCaml variables are:

» immutable

Two Major Concerns

1. Are variables *mutable*? Can we change their values? Are there restrictions to when we can change the value of a variable?
2. How are variables *scoped*? Dynamically or lexically? Does a binding define its own scope? Is it defined in a block?

OCaml variables are:

- » immutable
- » binding defined

Two Major Concerns

1. Are variables *mutable*? Can we change their values? Are there restrictions to when we can change the value of a variable?
2. How are variables *scoped*? Dynamically or lexically? Does a binding define its own scope? Is it defined in a block?

OCaml variables are:

- » immutable
- » binding defined
- » lexically scoped

Mutability

```
let x = 0
let f () =
  let x = 1 in
  ()
print_int x
```

Immutable (OCaml)

```
x = 0
def f():
    global x
    x = 1
print(x)
```

Mutable (Python)

Mutability

```
let x = 0
let f () =
  let x = 1 in
  ()
print_int x
```

Immutable (OCaml)

```
x = 0
def f():
    global x
    x = 1
print(x)
```

Mutable (Python)

Definition. (*informal*) A variable is **mutable** if we are allowed to change its value after it has been declared

Mutability

```
let x = 0
let f () =
  let x = 1 in
  ()
print_int x
```

Immutable (OCaml)

```
x = 0
def f():
    global x
    x = 1
print(x)
```

Mutable (Python)

Definition. (*informal*) A variable is **mutable** if we are allowed to change its value after it has been declared

We think of variables as:

Mutability

```
let x = 0
let f () =
  let x = 1 in
  ()
print_int x
```

Immutable (OCaml)

```
x = 0
def f():
    global x
    x = 1
print(x)
```

Mutable (Python)

Definition. (*informal*) A variable is **mutable** if we are allowed to change its value after it has been declared

We think of variables as:

» names if they're immutable

Mutability

```
let x = 0
let f () =
  let x = 1 in
  ()
print_int x
```

Immutable (OCaml)

```
x = 0
def f():
    global x
    x = 1
print(x)
```

Mutable (Python)

Definition. (*informal*) A variable is **mutable** if we are allowed to change its value after it has been declared

We think of variables as:

- » names if they're immutable
- » (abstract) memory locations when they're mutable

Scope

Scope

Definition. (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scope

Definition. (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scoping rules describe how the scope of bindings works in a program

Scope

Definition. (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scoping rules describe how the scope of bindings works in a program

There are two standard paradigms:

Scope

Definition. (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scoping rules describe how the scope of bindings works in a program

There are two standard paradigms:

» dynamic scoping

Scope

Definition. (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scoping rules describe how the scope of bindings works in a program

There are two standard paradigms:

- » dynamic scoping
- » lexical scoping (static scoping)

Scope

Definition. (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scoping rules describe how the scope of bindings works in a program

There are two standard paradigms:

- » dynamic scoping
- » lexical scoping (static scoping)

Warning. Scope is one of the most unclear terms in computer science, we might mean:

Scope

Definition. (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scoping rules describe how the scope of bindings works in a program

There are two standard paradigms:

- » dynamic scoping
- » lexical scoping (static scoping)

Warning. Scope is one of the most unclear terms in computer science, we might mean:

- » the scope of a variable

Scope

Definition. (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scoping rules describe how the scope of bindings works in a program

There are two standard paradigms:

- » dynamic scoping
- » lexical scoping (static scoping)

Warning. Scope is one of the most unclear terms in computer science, we might mean:

- » the scope of a variable
- » the scope of a binding

Scope

Definition. (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scoping rules describe how the scope of bindings works in a program

There are two standard paradigms:

- » dynamic scoping
- » lexical scoping (static scoping)

Warning. Scope is one of the most unclear terms in computer science, we might mean:

- » the scope of a variable
- » the scope of a binding
- » the scope of a function

Dynamic Scoping

```
f() { x=23; g; }
g() { y=$x; }
f
echo $y
```

Bash

Dynamic Scoping

```
f() { x=23; g; }
g() { y=$x; }
f
echo $y
```

Bash

Dynamic scoping refers to when bindings are determined at runtime based on *computational context*

Dynamic Scoping

$x = 23$
 $y = x$
print(y)

```
f() { x=23; g; }
g() { y=$x; }
f
echo $y
```

Bash

Dynamic scoping refers to when bindings are determined at runtime based on *computational context*

This is a *temporal view*, i.e., what a computation done beforehand which affected the value of a variable

Lexical Scoping

```
x = 0
def f():
    x = 1
    return x
assert(f() == 1)
assert(x == 0)
```

Python

```
let x = 0
let f () =
    let x = 1 in
    x
let _ = assert (f () = 1)
let _ = assert (x = 0)
```

OCaml

Lexical Scoping

```
x = 0
def f():
    x = 1
    return x
assert(f() == 1)
assert(x == 0)
```

Python

```
let x = 0
let f () =
    let x = 1 in
    x
let _ = assert (f () = 1)
let _ = assert (x = 0)
```

OCaml

Lexical (static) scoping refers to the use of textual delimiters to define the scope of a binding

Lexical Scoping

```
x = 0
def f():
    x = 1
    return x
assert(f() == 1)
assert(x == 0)
```

Python

```
let x = 0
let f () =
    let x = 1 in
    x
let _ = assert (f () = 1)
let _ = assert (x = 0)
```

OCaml

Lexical (static) scoping refers to the use of textual delimiters to define the scope of a binding

There are two common ways lexical scope is determined:

Lexical Scoping

```
x = 0
def f():
    x = 1
    return x
assert(f() == 1)
assert(x == 0)
```

Python

```
let x = 0
let f () =
    let x = 1 in
    x
let _ = assert (f () = 1)
let _ = assert (x = 0)
```

OCaml

Lexical (static) scoping refers to the use of textual delimiters to define the scope of a binding

There are two common ways lexical scope is determined:

- » The binding defines it's own scope (**let-bindings**)

Lexical Scoping

```
x = 0
def f():
    x = 1
    return x
assert(f() == 1)
assert(x == 0)
```

Python

```
let x = 0
let f () =
    let x = 1 in
    x
let _ = assert (f () = 1)
let _ = assert (x = 0)
```

OCaml

Lexical (static) scoping refers to the use of textual delimiters to define the scope of a binding

There are two common ways lexical scope is determined:

- » The binding defines it's own scope ([let-bindings](#))
- » A block defines the scope of a variable ([python functions](#))

Tradeoffs

```
f() { x=23; g; }
g() { y=$x; }
f
echo $y
```

dynamic

vs.

```
let x = 0
let f () =
    let x = 1 in
    x
let _ = assert (f () = 1)
let _ = assert (x = 0)
```

lexical

Implementing dynamic scoping is way easier... (we'll see this in lab)

But **every modern programming language** implements lexical scoping

The Environment Model

Why are we do this?

let x = v in ...

Why are we do this?

let x = v in ...

We've already implemented lexical scoping using the substitution model (mini-project 1) *Why do it again?*

Why are we do this?

let x = v in ...

We've already implemented lexical scoping using the substitution model (mini-project 1) *Why do it again?*

Answer. The substitution model is inefficient

Why are we do this?

let x = v in ...

We've already implemented lexical scoping using the substitution model (mini-project 1) *Why do it again?*

Answer. The substitution model is inefficient

Each substitution has to "crawl" through the *entire remainder of the program*

High Level

$$\langle \mathcal{E}, e \rangle \Downarrow \nu$$

High Level

$$\langle \mathcal{E}, e \rangle \Downarrow \nu$$

Idea. We keep track of their values in an *environment*

High Level

$$\langle \mathcal{E}, e \rangle \Downarrow v$$

Idea. We keep track of their values in an *environment*

And evaluate *relative* to the environment, *lazily*
filling in variable values along the way

High Level

$$\langle \mathcal{E}, e \rangle \Downarrow v$$

Idea. We keep track of their values in an *environment*

And evaluate *relative* to the environment, *lazily*
filling in variable values along the way

The configurations in our semantics will have nonempty state

Environment

\mathcal{E}

environment

$\underbrace{\text{Var} \mapsto \text{Vals}}$

partial mapping

dictionary

c.t.

$\{ x \mapsto 1, y \mapsto 2 \}$

\emptyset

empty

env.

$\mathcal{E}(x)$ or $\mathcal{E}[x]$

value of x in \mathcal{E}

$x \notin \text{dom}(\mathcal{E})$

x is not mapped in \mathcal{E}

$\mathcal{E}[x \mapsto v]$

add $x \mapsto v$ to \mathcal{E}

Shadowing

$$\mathcal{E}[x \mapsto v_1][x \mapsto v_2] = \mathcal{E}[x \mapsto v_2]$$

$$\{x \mapsto 1, y \mapsto 2\}[x \mapsto 3] = \{x \mapsto 3, y \mapsto 2\}$$

Lambda Calculus⁺ (Syntax)

```
<expr> ::= λ<var>. <expr>
          | <var>
          | <expr> <expr>
          | let <var> = <expr>
            in <expr>
          | <num>
```

```
<val> ::= λ<var>. <expr>
          | <num>
```

This is a grammar for the lambda calculus with
let-expressions and numbers

Lambda Calculus⁺ (Semantics)

Important. These rules are incorrect!

Lambda Calculus⁺ (Semantics)

Important. These rules are incorrect!

"values evaluate to values"

$$\frac{}{\langle \mathcal{E}, \lambda x . e \rangle \Downarrow \lambda x . e} \quad \frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

Lambda Calculus⁺ (Semantics)

Important. These rules are incorrect!

"values evaluate to values"

$$\frac{}{\langle \mathcal{E}, \lambda x . e \rangle \Downarrow \lambda x . e} \quad \frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

"variables evaluate to their values in the environment"

$$\frac{\mathcal{E}(x) \neq \perp}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

Lambda Calculus⁺ (Semantics)

Important. These rules are incorrect!

"values evaluate to values"

$$\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow \lambda x. e} \quad \frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

"variables evaluate to their values in the environment"

$$\frac{\mathcal{E}(x) \neq \perp}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

$[v_1/x]e_2$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \lambda x. e \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

"applications and let-expressions store arguments in the environment"

Why are these rules incorrect?

let $x = 0$ in

let $f = \lambda y. x$ in

let $x = 1$ in

$f\ 0$

Why are these rules incorrect?

let $x = 0$ in

let $f = \lambda y. x$ in

let $x = 1$ in

$f\ 0$

What's the value of this expression?

Example

$$\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow \lambda x. e}$$

$$\frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\frac{}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \lambda x. e \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

$\langle \{x \mapsto 0, f \mapsto \lambda y. x\} , \text{let } x = 1 \text{ in } f 0 \rangle \Downarrow 1$

Let's derive the above judgment in the given system

$$\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow \lambda x. e} \quad \frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\frac{}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \lambda x. e \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

Example

$$\{ x \mapsto 0, f \mapsto \lambda y. x \} [x \mapsto 1] = \{ x \mapsto 1, f \mapsto \lambda y. x \}$$

$$\frac{}{\langle \{ \dots \}, 0 \rangle \Downarrow 0}$$

$$\frac{}{\langle \{ x \mapsto 1, f \mapsto \lambda y. x \}, f \rangle \Downarrow \lambda y. x}$$

$$\frac{}{\langle \{ x \mapsto 0, f \mapsto \lambda y. x \}, 1 \rangle \Downarrow 1} \quad \frac{}{\langle \{ x \mapsto 1, y \mapsto \lambda y. x \}, f 0 \rangle \Downarrow 1}$$

$$\langle \{ x \mapsto 0, f \mapsto \lambda y. x \}, \text{let } x = 1 \text{ in } f 0 \rangle \Downarrow 1$$

:

$$\langle \emptyset, \text{let } x = 0 \text{ in let } f = \lambda y. x \text{ in let } x = 1 \text{ in } f 0 \rangle \Downarrow 1$$

Closures

Definition/Notation

$$(\mathcal{E}, e)$$

Definition/Notation

$$(\mathcal{E}, e)$$

Definition. (*informal*) A **closure** is a function together with an environment

Definition/Notation

$$(\mathcal{E}, e)$$

Definition. (*informal*) A **closure** is a function together with an environment

The environment *captures* bindings which a function needs

Definition/Notation

$$(\mathcal{E}, e)$$

Definition. (*informal*) A **closure** is a function together with an environment

The environment *captures* bindings which a function needs

Functions need to *remember* what the environment looks like in order to behave correctly according to lexical scoping

Lambda Calculus⁺ (Values)

$$\text{Val} = \mathbb{Z} \cup \text{Cls}$$

Lambda Calculus⁺ (Values)

$$\text{Val} = \mathbb{Z} \cup \text{Cls}$$

A value (a member of the set Val) is a **closure** (a member of the set Cls) or a **number** (a member of the set \mathbb{Z})

Lambda Calculus⁺ (Values)

$$\text{Val} = \mathbb{Z} \cup \text{Cls}$$

A value (a member of the set Val) is a **closure** (a member of the set Cls) or a **number** (a member of the set \mathbb{Z})

Important. Values no longer correspond with *expressions*. We're using the distinction between values and expressions to create a more efficient (and correct) semantics

Lambda Calculus⁺ (Correct Semantics)

values and variables

$$\frac{}{\langle \mathcal{E}, \lambda x . e \rangle \Downarrow (\mathcal{E}, \lambda x . e)}$$

$$\frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\frac{\mathcal{E}(x) \neq \perp}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

application

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (\mathcal{E}', \lambda x . e) \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

let-expressions

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

$$\frac{}{\langle \phi, \lambda x.y \rangle \Downarrow (\phi, \lambda x.y)}$$

$$\frac{}{\langle \phi, 2 \rangle \Downarrow 2}$$

$y \notin \text{dom } (\varepsilon)$

$$\frac{}{\langle \{\{x \mapsto 2\}, y \rangle, y \rangle}$$

$$\langle \phi, (\lambda x.y) 2 \rangle$$

Practice Problem

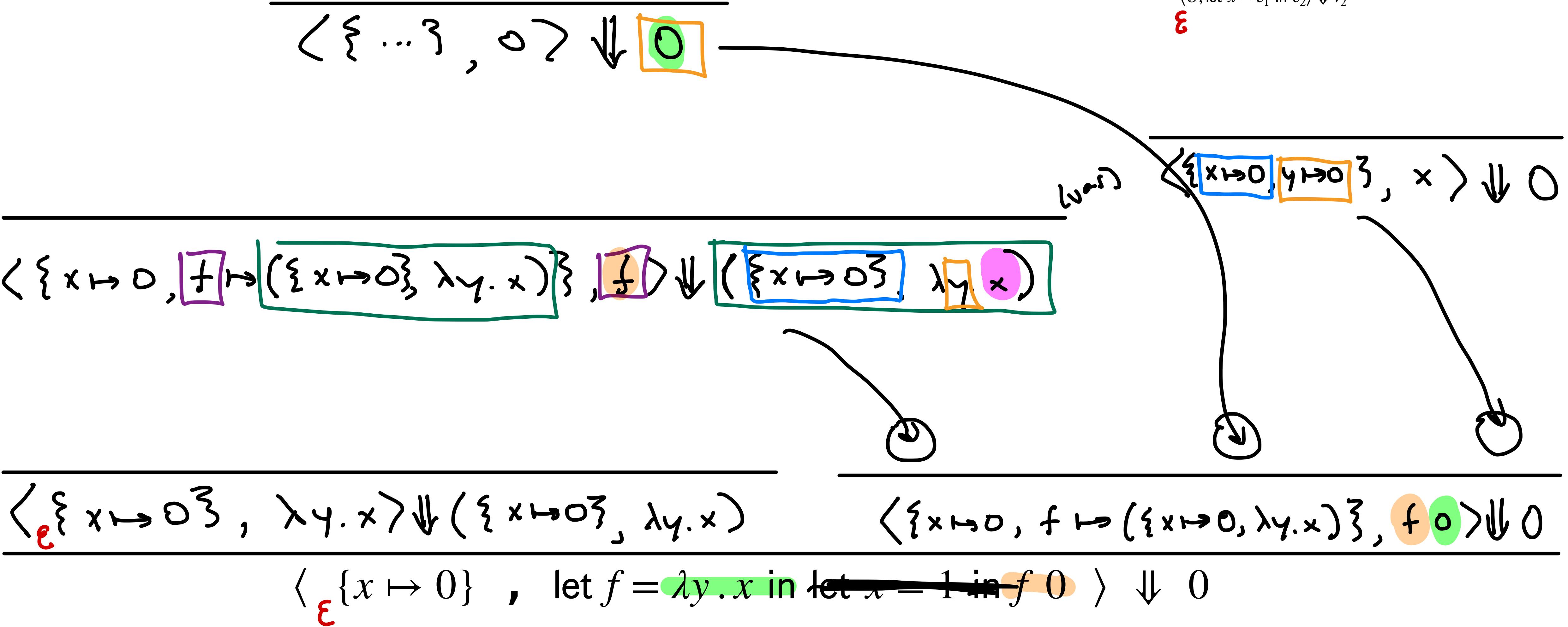
$$\langle \mathcal{E}, \lambda x. e \rangle \Downarrow \{\mathcal{E}, \lambda x. e\} \quad \langle \mathcal{E}, n \rangle \Downarrow n$$

$$(v_n) \frac{\mathcal{E}(x) \neq \perp}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \{\mathcal{E}', \lambda x. e\} \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

ε



Practice Problem

$$\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow \{\mathcal{E}, \lambda x. e\}} \quad \frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\frac{\mathcal{E}(x) \neq \perp}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \{\mathcal{E}', \lambda x. e\} \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

Exercise

$$\frac{}{\langle \{x \mapsto 0, f \mapsto (\{x \mapsto 0\}, \lambda y. x)\}, 1 \rangle \Downarrow 1}$$

$$\frac{}{\langle \{x \mapsto 1, f \mapsto (\{x \mapsto 0\}, \lambda y. x)\}, f 0 \rangle \Downarrow 0}$$

$$\frac{\langle \{x \mapsto 0, f \mapsto (\{x \mapsto 0\}, \lambda y. x)\}, \text{let } x = 1 \text{ in } f 0 \rangle \Downarrow 0}{\vdots}$$

$$\langle \{x \mapsto 0\}, \text{let } f = \lambda y. x \text{ in let } x = 1 \text{ in } f 0 \rangle \Downarrow 0$$

Recursion

High-Level

```
let f x =  
  if x = 0  
  then 1  
  else f (x - 1)  
in f 10
```

High-Level

```
let f x =  
  if x = 0  
  then 1  
  else f (x - 1)  
in f 10
```

What will happen if we evaluate the above program in our environment model (if we've given semantics to if-expressions, subtraction, etc)?

High-Level

```
let f x =  
  if x = 0  
  then 1  
  else f (x - 1)  
in f 10
```

What will happen if we evaluate the above program in our environment model (if we've given semantics to if-expressions, subtraction, etc)?

So far, we've only considered *non-recursive* functions (recursion is difficult...)

High-Level

```
let f x =  
  if x = 0  
  then 1  
  else f (x - 1)  
in f 10
```

What will happen if we evaluate the above program in our environment model (if we've given semantics to if-expressions, subtraction, etc)?

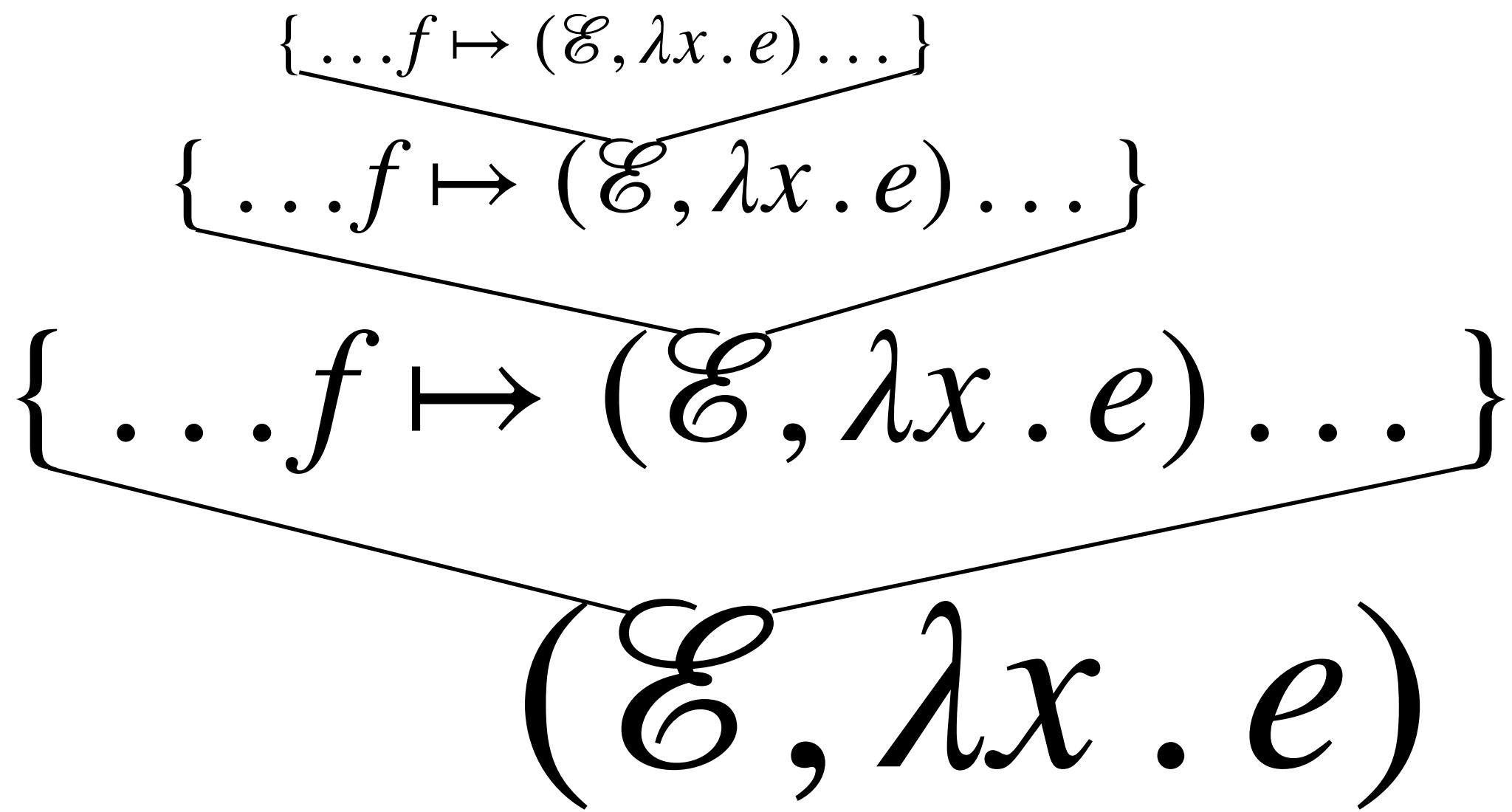
So far, we've only considered *non-recursive* functions (recursion is difficult...)

In the substitution model, there's no natural way to do it (though we can use fix-point combinators...)

The Problem

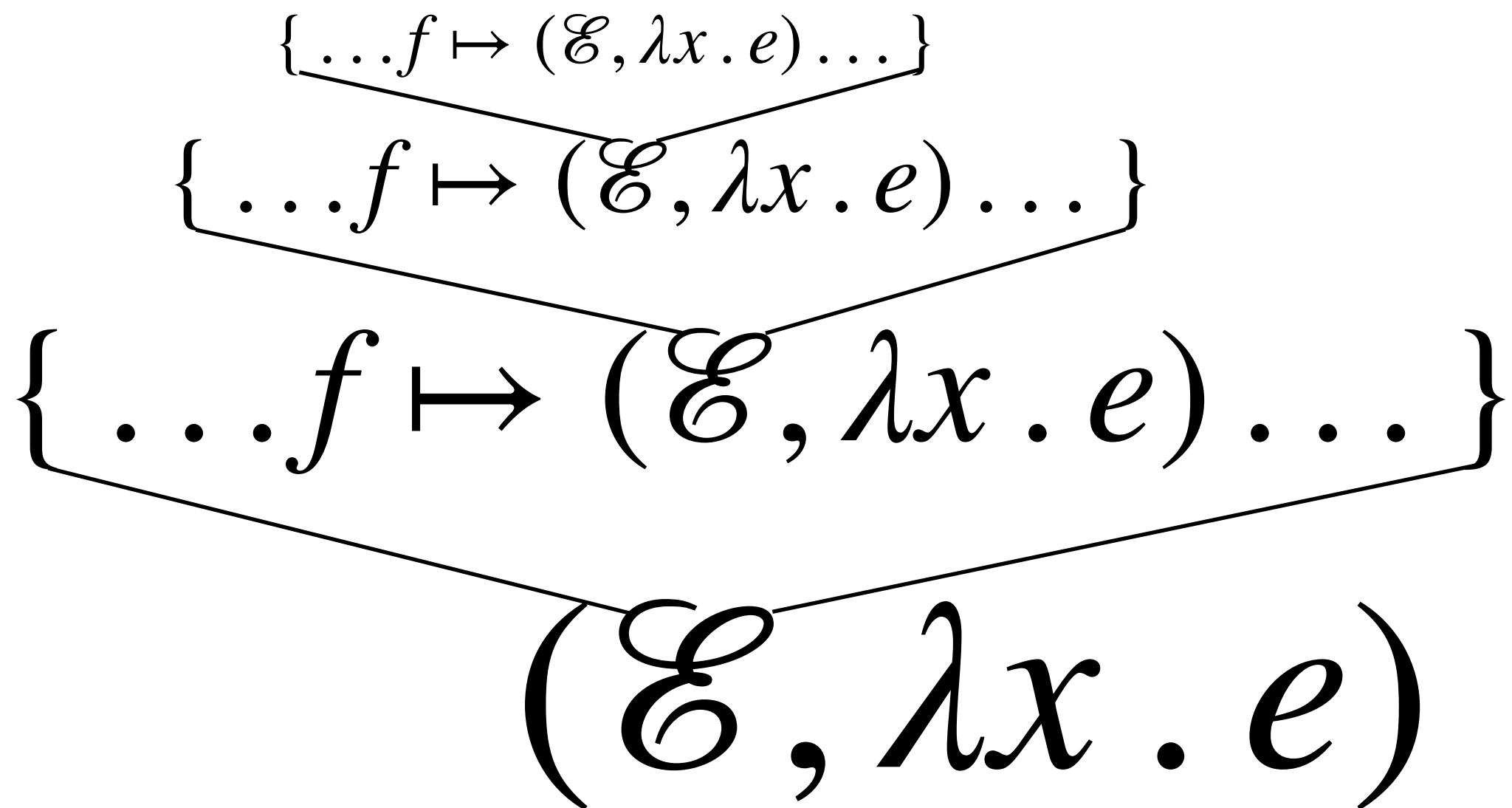
$$\begin{array}{c} \{\dots f \mapsto (\mathcal{E}, \lambda x. e) \dots\} \\ \diagdown \\ \{\dots f \mapsto (\mathcal{E}, \lambda x. e) \dots\} \\ \diagdown \\ \{\dots f \mapsto (\mathcal{E}, \lambda x. e) \dots\} \\ \diagdown \\ (\mathcal{E}, \lambda x. e) \end{array}$$

The Problem



In order to implement recursion, a closure has to "know thyself"

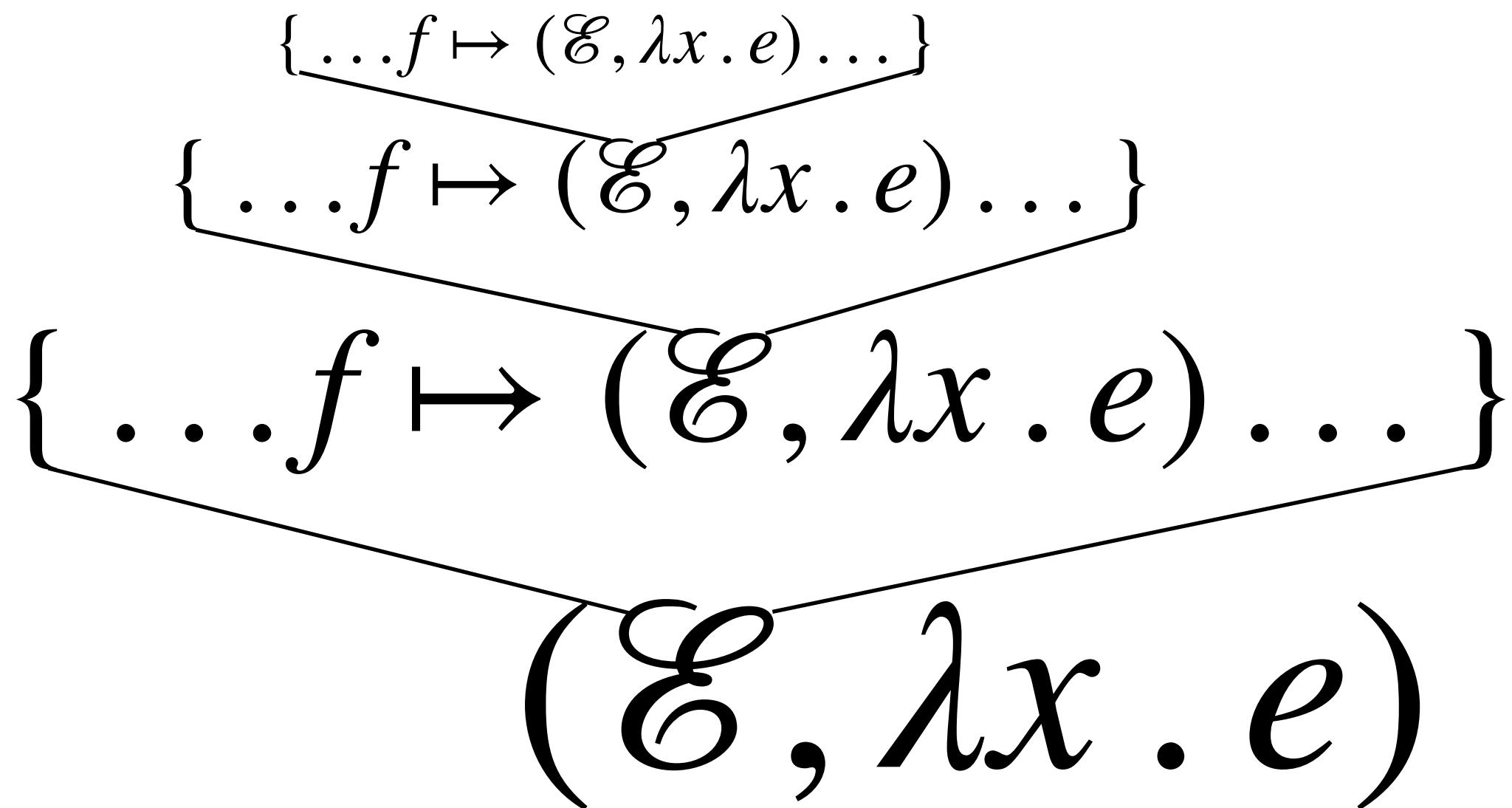
The Problem



In order to implement recursion, a closure has to "know thyself"

But we **can't** implement circular structures like this in OCaml

The Problem



In order to implement recursion, a closure has to "know thyself"

But we **can't** implement circular structures like this in OCaml

We need a way essentially to "simulate" pointers

Solution: Named Closures

(name, \mathcal{E} , $\lambda x . e$)

We need to be able to *name* closures

The idea. Named closures will put themselves
into their environment *when they're called*

Lambda Calculus⁺⁺ (Syntax, Again)

```
<expr> ::= λ<var>. <expr>
          | <var>
          | <expr> <expr>
          | let <var> = <expr>
             in <expr>
          | let rec <var> <var> = <expr>
             in <expr>
          | <num>
```

Lambda Calculus⁺⁺ (Syntax, Again)

```
<expr> ::= λ<var>. <expr>
          | <var>
          | <expr> <expr>
          | let <var> = <expr>
             in <expr>
          | let rec <var> <var> = <expr>
             in <expr>
          | <num>
```

The same grammar as before, but with recursive let-statements

Lambda Calculus⁺⁺ (Syntax, Again)

```
<expr> ::= λ<var>. <expr>
          | <var>
          | <expr> <expr>
          | let <var> = <expr>
             in <expr>
          | let rec <var> <var> = <expr>
             in <expr>
          | <num>
```

The same grammar as before, but with recursive let-statements

Important. A recursive let **must** take an argument

Lambda Calculus⁺⁺ (Semantics)

Lambda Calculus⁺⁺ (Semantics)

values and variables

$$\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow (\mathcal{E}, \lambda x. e)}$$

$$\frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\frac{\mathcal{E}(x) \neq \perp}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

Lambda Calculus⁺⁺ (Semantics)

values and variables

$$\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow (\mathcal{E}, \lambda x. e)}$$

$$\frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\frac{\mathcal{E}(x) \neq \perp}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

application (unnamed closure)

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (\mathcal{E}', \lambda x. e) \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

Lambda Calculus⁺⁺ (Semantics)

values and variables

$$\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow (\mathcal{E}, \lambda x. e)}$$

$$\frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\frac{\mathcal{E}(x) \neq \perp}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

application (unnamed closure)

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (\mathcal{E}', \lambda x. e) \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

application (named closure)

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (f, \mathcal{E}', \lambda x. e) \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[f \mapsto (f, \mathcal{E}', \lambda x. e)][x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

Lambda Calculus⁺⁺ (Semantics)

values and variables

$$\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow (\mathcal{E}, \lambda x. e)}$$

$$\frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\frac{\mathcal{E}(x) \neq \perp}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

application (unnamed closure)

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (\mathcal{E}', \lambda x. e) \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

application (named closure)

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (f, \mathcal{E}', \lambda x. e) \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[f \mapsto (f, \mathcal{E}', \lambda x. e)][x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

let expressions

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

$$\frac{\langle \mathcal{E}[f \mapsto (f, \mathcal{E}, \lambda x. e_1)], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let rec } f x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

Closer Look (Application)

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow (f, \mathcal{E}', \lambda x . e) \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}'[f \mapsto (f, \mathcal{E}', \lambda x . e)][x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

The only change here is that f is put into environment when f is called

This happens *every time* f is called (even within the body of f)

Closer Look (Recursive Definitions)

$$\frac{\langle \mathcal{E}[f \mapsto (f, \mathcal{E}, \lambda x. e_1)], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let rec } f x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

When a recursive function is declared it's given a *named* closure

Remember that we **must** take an argument in the case of a recursive closure

demo
(lambda calculus⁺)

Summary

Functions evaluate to **closures** so that they remember the environment in which they are defined

Recursive function evaluate to **named** closures so that they know how to evaluate themselves(!)