Closures and Environments

Concepts of Programming Languages
Lecture 17

CAS CS 320

Outline

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

Recap

Recall: Lexical Scoping

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

Python

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

OCaml

Recall: Lexical Scoping

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

Python

```ocaml
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

Recall: Lexical Scoping

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

Python

```ocaml
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

<u>There are two common ways lexical scope is determined:</u>

Recall: Lexical Scoping

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

Python

```ocaml
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

<u>There are two common ways lexical scope is determined:</u>

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

Recall: Lexical Scoping

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

Python

```ocaml
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)

Dynamic Scoping

```bash
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

```bash
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

Recall: Environments

$$\{x \mapsto v \ , \ y \mapsto w \ , \ z \mapsto f\}$$

Recall: Environments

$$\{x \mapsto v \ , y \mapsto w \ , z \mapsto f\}$$

An *environment* is a data structure which maintains mappings of variables to values

Recall: Environments

$$\{x \mapsto v\ ,\ y \mapsto w\ ,\ z \mapsto f\}$$

An *environment* is a data structure which maintains mappings of variables to values

Terminology. We call the individual mappings of variables to values **variable bindings**

Recall: Environments

$$\{x \mapsto v\ ,\ y \mapsto w\ ,\ z \mapsto f\}$$

An *environment* is a data structure which maintains mappings of variables to values

Terminology. We call the individual mappings of variables to values **variable bindings**

Usually it's implemented as an association list or a Map in OCaml (more on this in mini-project 2)

Recall: Environments

$$\{x \mapsto v \ , y \mapsto w \ , z \mapsto f\}$$

An *environment* is a data structure which maintains mappings of variables to values

Terminology. We call the individual mappings of variables to values **variable bindings**

Usually it's implemented as an association list or a Map in OCaml (more on this in mini-project 2)

The idea. We will evaluate expressions *relative* to an environment

Recall: Environment Operations

Math	OCaml
$\mathcal{E}$	`env`
$\mathcal{E}[x \mapsto v]$	`add x v env`
$\mathcal{E}(x)$	`find_opt x env`
$\mathcal{E}(x) = \bot$	`find_opt x env = None`

Recall: Environment Operations

Math	OCaml
$\mathcal{E}$	`env`
$\mathcal{E}[x \mapsto v]$	`add x v env`
$\mathcal{E}(x)$	`find_opt x env`
$\mathcal{E}(x) = \perp$	`find_opt x env = None`

Most important operations on environments are the same that are useful for any dictionary-like data structure

Recall: Environment Operations

Math	OCaml
$\mathscr{E}$	`env`
$\mathscr{E}[x \mapsto v]$	`add x v env`
$\mathscr{E}(x)$	`find_opt x env`
$\mathscr{E}(x) = \perp$	`find_opt x env = None`

Shadowing

$$\mathscr{E}[x \mapsto v][x \mapsto w] = \mathscr{E}[x \mapsto w]$$

Most important operations on environments are the same that are useful for any dictionary-like data structure

Important: Adding mappings shadows existing mappings!

Recall: Why are we doing this?

```
let x = v in ...
```

Recall: Why are we doing this?

```
let x = v in ...
```

We've already implemented lexical scoping using the substitution model (mini-project 1)

Recall: Why are we doing this?

```
let x = v in ...
```

We've already implemented lexical scoping using the substitution model (mini-project 1)

Why do it again?

Recall: Why are we doing 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

Recall: Why are we doing this?

```
let x = v in ...
```

We've already implemented lexical scoping using the substitution model (mini-project 1)

Why do it again?

<u>Answer.</u> The substitution model is inefficient

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

Recall: The Environment Model

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

Recall: The Environment Model

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

<u>Idea.</u> We keep track of their values in an *environment*

Recall: The Environment Model

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

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

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

Recall: The Environment Model

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

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

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

Now the **configurations** in our semantics have nonempty state

The Environment Model

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"

$$\frac{}{\langle \mathscr{E}, \lambda x \,.\, e \rangle \Downarrow \lambda x \,.\, e} \qquad \frac{}{\langle \mathscr{E}, n \rangle \Downarrow n}$$

Lambda Calculus$^+$ (Semantics)

Important. These rules are incorrect!

"values evaluate to values"

$$\frac{}{\langle \mathscr{E}, \lambda x\,.\,e \rangle \Downarrow \lambda x\,.\,e} \qquad \frac{}{\langle \mathscr{E}, n \rangle \Downarrow n}$$

"variables evaluate to their values in the environment"

$$\frac{\mathscr{E}(x) \neq \bot}{\langle \mathscr{E}, x \rangle \Downarrow \mathscr{E}(x)}$$

Lambda Calculus$^+$ (Semantics)

Important. These rules are incorrect!

"values evaluate to values"

$$\frac{}{\langle \mathscr{E}, \lambda x \,.\, e \rangle \Downarrow \lambda x \,.\, e} \qquad \frac{}{\langle \mathscr{E}, n \rangle \Downarrow n}$$

"variables evaluate to their values in the environment"

$$\frac{\mathscr{E}(x) \neq \bot}{\langle \mathscr{E}, x \rangle \Downarrow \mathscr{E}(x)}$$

$$\frac{\langle \mathscr{E}, e_1 \rangle \Downarrow v_1 \qquad \langle \mathscr{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathscr{E}, \textsf{let } x = e_1 \textsf{ in } e_2 \rangle \Downarrow v_2}$$

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

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

Why are these rules incorrect?

$$
\begin{array}{l}
\text{let } x = 0 \text{ in} \\
\text{let } f = \lambda y \,.\, x \text{ in} \\
\text{let } x = 1 \text{ in} \\
f\ 0
\end{array}
$$

Why are these rules incorrect?

$$
\begin{array}{l}
\text{let } x = 0 \text{ in} \\
\text{let } f = \lambda y \,.\, x \text{ in} \\
\text{let } x = 1 \text{ in} \\
f\ 0
\end{array}
$$

`What is the value of this expression in OCaml?`

Why are these rules incorrect?

$$
\begin{array}{l}
\text{let } x = 0 \text{ in} \\
\text{let } f = \lambda y \, . \, x \text{ in} \\
\text{let } x = 1 \text{ in} \\
f \; 0
\end{array}
$$

`What is the value of this expression in OCaml?`

We'll see next time that *we've actually implemented* ***dynamic scoping***

Example

$$\frac{}{\langle \mathscr{E}, \lambda x\,.\,e\rangle \Downarrow \lambda x\,.\,e} \qquad \frac{}{\langle \mathscr{E}, n\rangle \Downarrow n}$$

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

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

$$\frac{\langle \mathscr{E}, e_1\rangle \Downarrow v_1 \qquad \langle \mathscr{E}[x \mapsto v_1], e_2\rangle \Downarrow v_2}{\langle \mathscr{E}, \textsf{let } x = e_1 \textsf{ in } e_2\rangle \Downarrow v_2}$$

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

Let's derive the above judgment in the given system

Example

$$\overline{\langle \mathscr{E}, \lambda x\,.\,e\rangle \Downarrow \lambda x\,.\,e} \qquad \overline{\langle \mathscr{E}, n\rangle \Downarrow n}$$

$$\overline{\langle \mathscr{E}, x\rangle \Downarrow \mathscr{E}(x)}$$

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

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

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

$$\vdots$$

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

Closures

Definition/Notation

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

Definition/Notation

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

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

Definition/Notation

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

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

The environment *captures* bindings which a function needs

Definition/Notation

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

<u>Definition.</u> *(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 behavior correctly according to lexical scoping

Lambda Calculus^{+} (Values)

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

Lambda Calculus$^+$ (Values)

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

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)

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

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

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

application

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

let-expressions

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

The Derivation (Again)

$$\frac{}{\langle \mathscr{E}, \lambda x\,.\,e \rangle \Downarrow \{\mathscr{E}, \lambda x\,.\,e\}} \qquad \frac{}{\langle \mathscr{E}, n \rangle \Downarrow n}$$

$$\frac{\mathscr{E}(x) \neq \bot}{\langle \mathscr{E}, x \rangle \Downarrow \mathscr{E}(x)}$$

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

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

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

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

$$\{\ldots f \mapsto (\mathscr{E}, \lambda x\,.\,e) \ldots\}$$

$$\{\ldots f \mapsto (\mathscr{E}, \lambda x\,.\,e) \ldots\}$$

$$\{\ldots f \mapsto (\mathscr{E}, \lambda x\,.\,e) \ldots\}$$

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

The Problem

$$\{\ldots f \mapsto (\mathscr{E}, \lambda x\,.\,e) \ldots\}$$

$$\{\ldots f \mapsto (\mathscr{E}, \lambda x\,.\,e) \ldots\}$$

$$\{\ldots f \mapsto (\mathscr{E}, \lambda x\,.\,e) \ldots\}$$

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

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

The Problem

$$\{\ldots f \mapsto (\mathscr{E}, \lambda x \,.\, e) \ldots\}$$

$$\{\ldots f \mapsto (\mathscr{E}, \lambda x \,.\, e) \ldots\}$$

$$\{\ldots f \mapsto (\mathscr{E}, \lambda x \,.\, e) \ldots\}$$

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

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

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

The Problem

$$\{\ldots f \mapsto (\mathcal{E}, \lambda x \,.\, e) \ldots\}$$

$$\{\ldots f \mapsto (\mathcal{E}, \lambda x \,.\, e) \ldots\}$$

$$\{\ldots f \mapsto (\mathcal{E}, \lambda x \,.\, e) \ldots\}$$

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

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

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

We need to be able to *name* closures

<u>The idea.</u> 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`

$$\frac{}{\langle \mathscr{E}, \lambda x \,.\, e \rangle \Downarrow (\mathscr{E}, \lambda x \,.\, e)} \qquad \frac{}{\langle \mathscr{E}, n \rangle \Downarrow n} \qquad \frac{\mathscr{E}(x) \neq \bot}{\langle \mathscr{E}, x \rangle \Downarrow \mathscr{E}(x)}$$

Lambda Calculus^{++} (Semantics)

values and variables

$$\frac{}{\langle \mathscr{E}, \lambda x \,.\, e \rangle \Downarrow (\mathscr{E}, \lambda x \,.\, e)} \qquad \frac{}{\langle \mathscr{E}, n \rangle \Downarrow n} \qquad \frac{\mathscr{E}(x) \neq \bot}{\langle \mathscr{E}, x \rangle \Downarrow \mathscr{E}(x)}$$

application (unnamed closure)

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

Lambda Calculus^{++} (Semantics)

values and variables

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

application (unnamed closure)

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

application (named closure)

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

Lambda Calculus^{++} (Semantics)

values and variables

$$\frac{}{\langle \mathscr{E}, \lambda x \,.\, e \rangle \Downarrow (\mathscr{E}, \lambda x \,.\, e)} \qquad \frac{}{\langle \mathscr{E}, n \rangle \Downarrow n} \qquad \frac{\mathscr{E}(x) \neq \bot}{\langle \mathscr{E}, x \rangle \Downarrow \mathscr{E}(x)}$$

application (unnamed closure)

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

application (named closure)

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

let expressions

$$\frac{\langle \mathscr{E}, e_1 \rangle \Downarrow v_1 \qquad \langle \mathscr{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathscr{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2} \qquad \frac{\langle \mathscr{E}[f \mapsto (f, \mathscr{E}, \lambda x \,.\, e_1)], e_2 \rangle \Downarrow v_2}{\langle \mathscr{E}, \text{let rec } f\ x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

Closer Look (Application)

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

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)

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

`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

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(!)