Closures and the Environment Model

Principles of Programming Languages

CAS CS 320

Lecture 18

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

Learning Objectives

- Determine if a set of semantic rules implements lexical or dynamic scoping
- Give an example of a program which behaves differently depending on whether closures are used
- Determine what closure a function evaluates to in a given OCaml program
- Implement the environment model for the lambda calculus

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

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)
» A block defines the scope of a variable (python functions)

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. We have a special data structure called `env` for implementing environments.

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`
$\mathcal{E}(x) = \perp$	`find_opt x env = None`

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

Important: Adding mappings shadows existing mappings:

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

Recall: Why are we doing this?

```
let x = v in
[very large expression]
```

The substitution model is inefficient

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

(As usual, this is a simplification)

Recall: The Environment Model

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

Rather than *eagerly* substituting variables, we'll keep track of their values in an environment

We'll then evaluate *relative* to the environment, *lazily* filling in variable values along the way

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

(This might feel a bit more natural than substitution from the perspective of imperative languages)

Recall: 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

Challenge Problem. Rewrite this grammar so that it is not ambiguous

Recall: Lambda Calculus$^+$ (Semantics)

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

"values evaluate to values"

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

"variables evaluate to their values in the environment"

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

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

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

Important. These rules are incorrect!

Recall: What went wrong?

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

What is the value of this expression?

On the next slide, we'll see that **we accidentally implemented dynamic scoping**

Practice Problem

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

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

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

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

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

Derive the above judgment in the given system.

Answer

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

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

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

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

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

$$\vdots$$

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

Closures

Definition/Notation

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

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

The environment **captures** bindings which a function depends on

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

The **values** of our language will include **closures** instead of functions *(this is where we see that it's often useful to have values which aren't the same as expressions)*

Values

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

A value (a member of the set Val) in our toy language 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)

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

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

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

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

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

The Derivation (Again)

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

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

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

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

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

Practice Problem

```
let x = 0 in
let g = fun x -> x + 1 in
let f = fun y -> g x in
let x = 1 in
f
```

What (closure) does the following expression evaluate to? You don't need to give the derivation.

Answer

```
let x = 0 in
let g = fun x -> x + 1 in
let f = fun y -> g x in
let x = 1 in
f
```

Recursion

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 <u>non-recursive</u> functions (recursive is difficult...)

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

There are many ways to deal with this in the environment model. We will look at one

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

Named Closures

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

First, we need to be able to *name* closures. If a closure is named, it is intended to be recursive

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>
```

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

Our values now include *both* named and unnamed closures

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

Lambda Calculus^{++} (Semantics)

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

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

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

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

Closer Look

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

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

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

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

Example

```
let rec f x =
  if x = 0
  then x
  else f 0
in f 1
```

Demo