Variables and Environments

Principles of Programming Languages

CAS CS 320

Lecture 16

Practice Problem

$$[\boxed{(\lambda x \,.\, xy)}/z](\lambda x \,.\, \lambda y \,.\, yz)$$

$\lambda x . \lambda \boxed{y} . y(\lambda x . x \boxed{y})$

Perform the following substitution, avoiding variable capture.

$$[v/y]x = \begin{cases} v & x = y \\ x & \texttt{else} \end{cases}$$

$$[v/y](\lambda x \,.\, e) = \begin{cases} \lambda x \,.\, e & x = y \\ \lambda z \,.\, [w/z][z/x]e & x \in FV(v), z \notin FV(e) \\ \lambda x \,.\, [v/y]e & \texttt{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

Answer

$$[(\lambda x\,.\,xy)/z](\lambda x\,.\,\lambda y\,.\,yz)$$

$[\lambda x.\, xy\ /z](\lambda x.\, \lambda y.\, yz) =$

$FV(\lambda x.xy) = \{y\}$

$\lambda x.\ [\lambda x.xy/z](\lambda y.yz)$

$\lambda x.\ \lambda w.\ [\lambda x.xy/z][w/y]\,yz =$

$\lambda x.\lambda w.\ [\lambda x.xy/z]\ wz =$

$\lambda x.\lambda w.\ w(\lambda x.xy)$

Outline

Demo an implementation of the lambda calculus

Discuss the difference between lexical and dynamic scoping

Look at the semantics of variable binding, with examples using environments

Begin to look at how closures are used to implement lexical scoping with environments

Learning Objectives

- Describe the difference between dynamic and lexical scoping

- Give a sequence of reductions given a toy language which implements dynamic scoping

- Give a semantic derivation in the lambda calculus with let-expressions

Demo (The Lambda Calculus)

Variables

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?

<u>OCaml variables are:</u>

» immutable
» binding defined
» lexically scoped

Mutability

x shadows the other x.

```
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.

<u>We think of variables as:</u>

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

OCaml:

0

python:

1

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.

<u>There are two standard paradigms:</u>

» dynamic scoping
» lexical scoping (static scoping)

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

» the scope of a variable
» the scope of a binding
» the scope of a function
» scopes in general (like *global* scope)
» "this variables is not in scope..."

Dynamic Scoping

assign x to 23

calling g

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

assign y to val of x

call f

Bash

print value of y

print:

23

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

scope of binding of x

```
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)
```

scope of binding of x

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)

Environments

High-Level

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

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

$\{x \mapsto 2 \text{ , } y \mapsto 3\}$

want to eval: $x+y$

let x = 2 in
let y = 3 in
x + y

Operations

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

?? None

x is not bound in $\mathcal{E}$

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

Important: Adding mappings shadows existing mappings:

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

$\{x \mapsto 2\}[x \mapsto 3] =$

$\{x \mapsto 3\}$

Dynamic Scoping

Toy Language (Syntax)

```
<prog>  ::= { <stmt>; }
<stmt>  ::= <var>() { { <stmt1>; } }
          | <stmt1>
<stmt1> ::= <var> | <var> = $<var>
          | <var> = <num>
<var>   ::= ...
<num>   ::= ...
```

This is a small grammar for a language with, numbers, subroutines, and variable assignments

It's like a tiny fragment of Bash

example

var

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

Toy Language (Semantics)

```
<prog>  ::= { <stmt>; }
<stmt>  ::= <var>() { { <stmt1>; } }
          | <stmt1>
<stmt1> ::= <var> | <var> = $<var>
          | <var> = <num>
<var>   ::= ...
<num>   ::= ...
```

config

$$\frac{}{\underbrace{\langle\ \overbrace{\mathcal{E}}^{\text{state}},\ \underbrace{f\texttt{()}\ \texttt{\{}\ P\ \texttt{\};}\ Q}_{\text{prog}}\ \rangle}_{\text{config}} \longrightarrow \langle\ \mathcal{E}[f \mapsto P],\ Q\ \rangle}\ \textbf{(func)}$$

"put f into env"

$$\frac{\mathcal{E}(f) = P \in \mathbb{F}}{\langle\ \mathcal{E},\ f\texttt{;}\ Q\ \rangle \longrightarrow \langle\ \mathcal{E},\ \boxed{P\ Q}\ \rangle}\ \textbf{(call)}$$

"prepend body of f to curr. prog."

$$\frac{\mathcal{E}(y) = n \in \mathbb{Z}}{\langle\ \mathrm{E},\ x\ \texttt{=}\ \texttt{\$}y\texttt{;}\ Q\ \rangle \longrightarrow \langle\ \mathcal{E}[x \mapsto n],\ Q\ \rangle}\ \textbf{(aVar)}$$

"put x → val of y in env"

$$\frac{}{\langle\ \mathrm{E},\ x\ \texttt{=}\ n\texttt{;}\ Q\ \rangle \longrightarrow \langle\ \mathcal{E}[x \mapsto n],\ Q\ \rangle}\ \textbf{(aNum)}$$

Note. The environment contains both functions ($\mathbb{F}$) and numbers ($\mathbb{Z}$)

Toy Language (Example)

$$\overline{\langle\ \mathcal{E},\ f()\ \{\ P\ \};\ Q\ \rangle \longrightarrow \langle\ \mathcal{E}[f \mapsto P],\ Q\ \rangle}\ \texttt{(func)}$$

$$\frac{\mathcal{E}(f) = P \in \mathbb{F}}{\langle\ \mathcal{E},\ f;\ Q\ \rangle \longrightarrow \langle\ \mathcal{E},\ P\ Q\ \rangle}\ \texttt{(call)}$$

$$\frac{\mathcal{E}(y) = n \in \mathbb{Z}}{\langle\ \mathsf{E},\ x\ =\ \$y;\ Q\ \rangle \longrightarrow \langle\ \mathcal{E}[x \mapsto n],\ Q\ \rangle}\ \texttt{(aVar)}$$

$$\overline{\langle\ \mathsf{E},\ x\ =\ n;\ Q\ \rangle \longrightarrow \langle\ \mathcal{E}[x \mapsto n],\ Q\ \rangle}\ \texttt{(aNum)}$$

$\langle \emptyset,$ `f() { x=23; g; }; g() { y=$x; }; f;` $\rangle \rightarrow$

$\langle \{f \mapsto x=23; g; \},$ `g() { y=$x }; f;` $\rangle \rightarrow$

$\langle \{f \mapsto x=23; g;,\ g \mapsto y=\$x\},\ f; \rangle \rightarrow$

$\langle \{f \mapsto \cdots,\ g \mapsto \cdots\},\ x=23;\ g; \rangle \rightarrow$

$\langle \{x \mapsto 23,\ f \mapsto \cdots,\ g \mapsto \cdots\},\ g; \rangle \rightarrow$

$\langle \{\boxed{x \mapsto 23},\ f \mapsto \cdots,\ g \mapsto \cdots\},\ y=\$x; \rangle \rightarrow$

$\langle \{x \mapsto 23,\ y \mapsto 23,\ f \mapsto \cdots,\ g \mapsto \cdots\},\ \varepsilon \rangle$

empty program

The Takeaway

Defining and implementing dynamic scoping is very easy

It's like maintaining *only* a global scope

But it's behavior is harder to track, and arguably more error prone, you have to *remember* if there is some computation which will put a variable into scope.

Most modern programming languages implement lexical scoping

Lexical Scoping

Didn't we do this?

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

really big expression

may only have 1 or 2 x's.

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*

The Environment Model (High-Level)

config

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

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)

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

let x = 0 in
let f = λy.x in
let x = 1 in
f 10

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 eval to values"

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

"variables eval. to val. in env."

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

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

same as before except we keep track bindings in env.

Important. These rules are incorrect!

What went wrong?

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

shadow

What is the value of this expression?

We'll see on the next slide that **we accidentally implemented dynamic scoping**

The Derivation

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

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

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

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

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

Closures (Looking Ahead)

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

What do we do about this?

The problem was that f saw a different value of x than what it was when $f\ 0$ was evaluated

So functions will need to <u>remember</u> the environments they were defined in

Summary

To deal with variables in an imperative setting, we need <u>environments</u>

To deal with variables in a function setting, we don't *need* environments, but they will make our implementation more efficient

<u>Dynamic</u> scoping says a variable binding is determined by the computational context whereas <u>lexical</u> scoping says its determine by its lexical (textual) context