CS 314 Principles of Programming Languages

Lecture 17: Lambda Calculus

Prof. Zheng Zhang

Rutgers University

October 31, 2018

Class Information

- Midterm exam 11/7 Wednesday 10:20am - 11:40am
- Extended hours are Posted
- No classes on 11/2 this Friday

Review: Lambda Calculus - Historical Origin

- **Church's model of computing is called the *lambda calculus***

  It is based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter λ — hence the notation's name). Lambda calculus was the inspiration for functional programming: one uses it to compute by *substituting parameters into expressions,* just as one computes in a high level functional program by *passing arguments to functions*.

Review: Functional Programming

- Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are an attempt to realize Church's lambda calculus in practical form as a programming language

- **The key idea: do everything by composing functions**
  - No mutable state
  - No side effects
  - Function as first-class values

Review: Lambda Calculus

λ-terms are inductively defined.

A **λ-term** is:

- a variable x
- (λx. M) ⇒ where x is a variable and λ is a λ-term (abstraction)
- (M N) ⇒ where M and N are both λ-terms (application)

Review: λ-terms

The context-free grammar for λ-terms:

λ-term	→ expr
expr	→ name | number | λ name . expr | func arg
func	→ name | (λ name . expr) | func arg
arg	→ name | number | (λ name . expr) | (func arg)

Example 1:

$$\boldsymbol{\lambda}\, \boxed{\mathbf{y}}\ .\ \boxed{\mathbf{y}\ \ \mathbf{x}}$$

name (as parameter) expr (another λ-term)

Review: λ-terms

The context-free grammar for λ-terms:

λ-term	→ expr
expr	→ name | number | λ name . expr | func arg
func	→ name | (λ name . expr) | func arg
arg	→ name | number | (λ name . expr) | (func arg)

Example 2:

$\boxed{\mathbf{y}}$ $\boxed{\mathbf{z}}$

func arg

Review: Lambda Calculus

Associativity and Precedence

- Function application is left associative: (f g z) is ((f g) z)
- Function application has precedence over function abstraction. "function body" extends as far to the right as possible:

  (λx.yz) is (λx.(yz))
- Multiple arguments: (λxy.z) is (λx(λy.z))

Review: Free and Bound Variables

Abstraction (λx. M**)** "binds" variable x in "body" M. You can think of this as a declaration of variable x with scope M.

$$(\lambda\ \mathbf{y}\ .\ \mathbf{y}\ \mathbf{z})\ \mathbf{y}$$

free occurrence

binding occurrence

bound occurrence

Review: Free and Bound Variables

Note:

A variable can occur **free** and **bound** in a λ-term.

Example:

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

y is free

y is bound

"free" is relative to a λ-sub-term.

Review: Free and Bound Variables

Let M, N be λ-terms and x is a variable. The set of *free variable* of M, free(M), is defined inductively as follows:

- free(x) = {x}
- free(M N) = free(M) ∪ free(N)
- free (λx.M) = free(M) - free(x)

Review: Function Application

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

β–reduction:

$$(\lambda x.\mathbf{E})y \;\to_\beta\; \mathbf{E}[y/x]$$

1. Return function body E
2. Replace every free occurrence of x in E with y

Review: Function Application

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

β–reduction:

$$(\lambda x.\mathbf{E})y \ \rightarrow_\beta \ \mathbf{E}[y/x]$$

1. Return function body E
2. Replace every free occurrence of x in E with y

Example:

$$(\lambda a.\lambda b.a+b)\ 2\ x \ \rightarrow_\beta \ (\lambda b.2+b)\ x$$

$$\rightarrow_\beta \ 2+x$$

Function Application

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

β–reduction:

$$(\lambda x.\mathbf{E})y \ \rightarrow_\beta \ \mathbf{E}[y/x]$$

We should not perform β–reduction if y is a bound variable within E

Example:

$$\underline{(\lambda a.\lambda b.a+b)}\ b\ 2 \ \rightarrow_\beta \ \boxed{(\lambda b.b+b)\ 2} \rightarrow \text{Incorrect}$$

$$\rightarrow_\beta \ 2+2$$

b is a bound variable within λa.λb.a+b

This is called capturing

Review: Function Application

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

α–reduction:

$$(\lambda x.\mathbf{E}) \quad \rightarrow_{\alpha} \quad \lambda y.\mathbf{E}[\mathbf{y}/\mathbf{x}]$$

Review: Function Application

Computation in lambda calculus is based on the concept or reduction. Simplify an expression until it can no longer be simplified.

α–reduction:

$$(\lambda x.\mathbf{E}) \quad \rightarrow_{\alpha} \quad \lambda y.\mathbf{E[y/x]}$$

Perform α–reduction first

Example: $(\lambda a.\underline{\lambda b.a+b})\ b\ 2 \rightarrow_{\alpha} \boxed{(\lambda a.\underline{\lambda x.a+x})}\ b\ 2$

$\rightarrow_{\beta}\ \lambda x.b+x\ 2$

$\rightarrow_{\beta}\ b+2$

Review: Programming in Lambda Calculus

Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly β–reductions).

Logical constants and operations (incomplete list):

true ≡ λa. λb. a *select-first*

false ≡ λa. λb. b *select-second*

cond ≡ λp. λm. λn.(p m n)

not ≡ λx. (x false true)

and ≡ λx. λy. (x y false)

or ≡ homework

Review: Programming in Lambda Calculus

What about data structures?

Data structures:

pairs can be represented as:

$$[M.N] \;\equiv\; \lambda z.\ (z\ M\ N)$$

first $\equiv$ λx. (x true) (car)
second $\equiv$ λx. (x false) (cdr)
build $\equiv$ λx.λy.λz. (z x y) (cons)

Programming in Lambda Calculus

What about the encoding of arithmetic constants?

$\boxed{\text{Church Numerals:}}$

$0 \equiv \lambda fx.\ x$

$1 \equiv \lambda fx.\ (f\ x)$

$2 \equiv \lambda fx.\ (f\ (f\ x))$

…

$n \equiv \lambda fx.(\ f\ (\ f\ (\ldots\ (\ f\ x)\ \ldots\ ))\ \equiv\ \lambda fx.\ (f^{\,n}x)$

The natural number n is represented as a function that applies a function *f* *n*-times to x.

succ $\equiv \lambda m.\ (\lambda fx.(f\ (m\ f\ x)))$
add $\equiv \lambda mn.\ (\lambda fx.((m\ f)\ (n\ f\ x)))$
mult $\equiv \lambda mn.\ (\lambda fx.((m\ (n\ f))\ x))$
isZero? $\equiv \lambda m.\ (m\ \lambda x.\text{false true})$

Recursion in Lambda Calculus

Does this make sense?

$$\mathbf{f} \equiv \ldots \mathbf{f} \ldots$$

In lambda calculus, $\equiv$ is "abbreviated as", **but not an assignment.**

Recursion in Lambda Calculus

Does this make sense?

$$\mathbf{f \equiv \ldots\, f \ldots}$$

In lambda calculus, $\equiv$ is “abbreviated as”, **not an assignment.**

$$\boxed{\mathbf{add} \equiv \lambda mn.\ (\text{ if } (\text{isZero?}n)\ m\ (\mathbf{add}\ (m+1)\ (n-1))\ )}$$

Incorrect! “add” is not defined

How about

$$\underbrace{\boxed{\lambda \mathbf{f}.(\lambda mn.(\text{ if } (\text{isZero?}n)\ m\ (\mathbf{f}\ (m+1)\ (n-1)))\ )}}_{F}\ \mathbf{add}$$

$$\lambda mn.(\text{ if } (\text{isZero?}n)\ m\ (\mathbf{add}\ (m+1)\ (n-1))) \quad = \quad \mathbf{add}$$

$$F\ \mathbf{add} \quad = \quad \mathbf{add}$$

“add” is a fixed point to function F

Function Fixed Points

The fixed point of a function *g* is the set of values

$$\{ x \mid x = g(x) \}$$

Examples:

function **g**	fix(g)
λx.6	{6}
λx.(6 - x)	{3}
λx.((x * x) + (x - 4))	{-2, 2}
λx.x	entire domain of function f
λx.(x + 1)	{ }

Function Fixed Points

Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

YES. $x = YF$, and Y is called the fixed point combinator.

$$Y \equiv \lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))$$

$$YF = ((\lambda f.((\lambda x.f(\ x\ x))\ (\lambda x.f(\ x\ x\ ))))\ F)$$

Function Fixed Points

Is there a way to “compute” the fixed point of any function F

$$x = F(x)$$

YES. $x = YF$, and Y is called the fixed point combinator.

$$Y \equiv \lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))$$

$$YF = ((\lambda f.((\lambda x.\underline{f}(\ x\ x))\ (\lambda x.\underline{f}(\ x\ x\ ))))\ \underline{F})$$

Function Fixed Points

Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

YES. x = YF, and Y is called the fixed point combinator.

$$Y \equiv \lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))$$

$$YF = ((\lambda f.((\lambda x.\underline{f}(\ x\ x))\ (\lambda x.\underline{f}(\ x\ x\ ))))\ \underline{F})$$

$$= (\lambda x.F(\underline{x}\ x))\ \underline{(\lambda x.F(x\ x)})$$

Function Fixed Points

Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

YES. x = YF, and Y is called the fixed point combinator.

$$Y \equiv \lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))$$

$$YF = ((\lambda f.((\lambda x.f(\ x\ x))\ (\lambda x.\underline{f}(\ x\ x\ ))))\ F)$$

$$= (\lambda x.F(\underline{x}\ \underline{x}))\ \underline{(\lambda x.F(x\ x)}))$$

Function Fixed Points

Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

YES. $x = YF$, and Y is called the fixed point combinator.

$$Y \equiv \lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))$$

$$\begin{aligned} YF &= ((\lambda f.((\lambda x.f(\ x\ x))\ (\lambda x.\underline{f}(\ x\ x\ ))))\ F) \\ &= (\lambda x.F(\underline{x}\ \underline{x}))\ \underline{(\lambda x.F(x\ x)})\big) \\ &= F(\ (\lambda x.F(x\ x))\ (\lambda x.F(x\ x))) \end{aligned}$$

Function Fixed Points

Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

YES. x = YF, and Y is called the fixed point combinator.

$$Y \equiv \lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))$$

$$YF = ((\lambda f.((\lambda x.f(\ x\ x))\ (\lambda x.\underline{f}(\ x\ x\ ))))\ F)$$

$$YF = (\lambda x.F(x\ x))\ (\lambda x.F(x\ x))$$

$$YF = F(\ (\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))$$

Function Fixed Points

Is there a way to "compute" the fixed point of any function F

$$x = F(x)$$

YES. x = YF, and Y is called the fixed point combinator.

$$Y \equiv \lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))$$

$$YF = ((\lambda f.((\lambda x.f(\ x\ x))\ (\lambda x.\underline{f}(\ x\ x\ ))))\ F)$$

$$YF = (\lambda x.F(x\ x))\ (\lambda x.F(x\ x))$$

$$YF = F(\ (\lambda x.F(x\ x))\ (\lambda x.F(x\ x)))$$

$$YF = F(YF)$$

The Y - Combinator Example (Cont.)

- Informally, the Y-Combinator allows us to get as many copies of the recursive procedure body as we need. The computation "unrolls" recursive procedure calls one at a time

$$Y \equiv \lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x)))$$

The Y - Combinator

Example:

$\mathbf{F} \equiv \lambda \mathbf{f}.\ (\lambda mn.\ \text{if (isZero? n) then m else } (\mathbf{f}\ (\text{succ m})\ (\text{pred n})))$

$$
\begin{aligned}
(YF\ 3\ 2) &= (((\lambda f.((\lambda x.f(x\ x))\ (\lambda x.f(x\ x))))\ F)\ 3\ 2) \\
&= (\ (F\ (YF)\ )\ 3\ 2) \\
&= ((\lambda mn.\text{if (isZero? n) then m else } YF\ (\text{succ m})\ (\text{pred n})))\ 3\ 2) \\
&= \text{if (isZero? 2) then 3 else } YF\ (\text{succ 3})\ (\text{pred 2})) \\
&= (\ YF\ 4\ 1) \\
&= ((F\ (YF))\ 4\ 1) \\
&= \text{if (isZero? 1) then 4 else } YF\ (\text{succ 4})\ (\text{pred 1})) \\
&= (\ YF\ 5\ 0\ ) \\
&= (\ F\ (YF)\ 5\ 0\ ) \\
&= \text{if (isZero? 0) then 5 else } (\ YF\ (\text{succ 5})\ (\text{pred 0})) \\
&= \mathbf{5}
\end{aligned}
$$

Lambda Calculus - Final Remarks

- We can express all computable functions in our λ-calculus.
- All computable functions can be expressed by the following two combinators, referred to as **S** and **K**.
  - $K \equiv \lambda xy.x$
  - $S \equiv \lambda xyz.xz(yz)$

Combinatoric logic is as powerful as Turing Machines.

Next Lecture

Reading:

- Scott, Chapter 11.1 - 11.3
- Scott, Chapter 11.7
- ALSU, Chapter 11.1 - 11.3