CS 314 Principles of Programming Languages

Lecture 20: Lambda Calculus

Zheng (Eddy) Zhang

Rutgers University

April 18, 2018

Lambda Calculus

λ-terms are inductively defined.

A **λ-term** is:

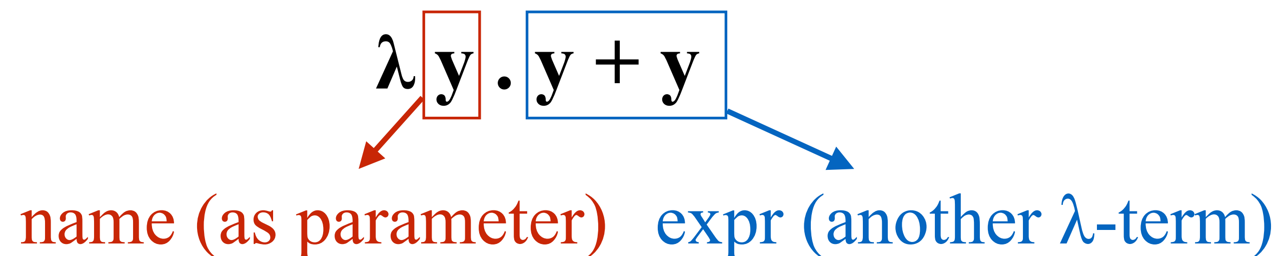
- a variable x
- (λx. M) ⇒ where x is a variable and λ is a λ-term (abstraction)
- (M N) ⇒ where M and N are both λ-terms (application)

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:

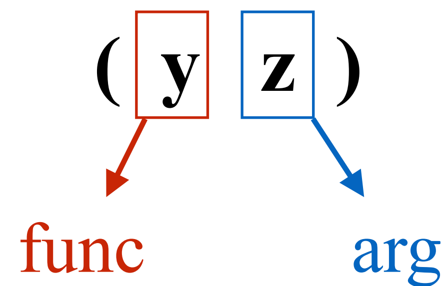


λ-terms

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:



Lambda Calculus

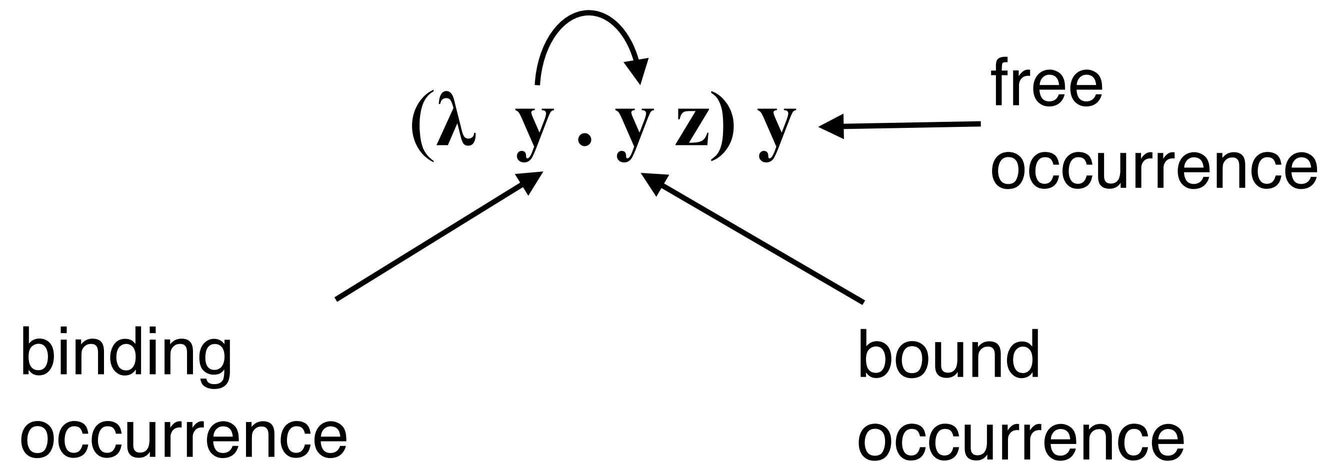
Associativity and Precedence

- 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))

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.

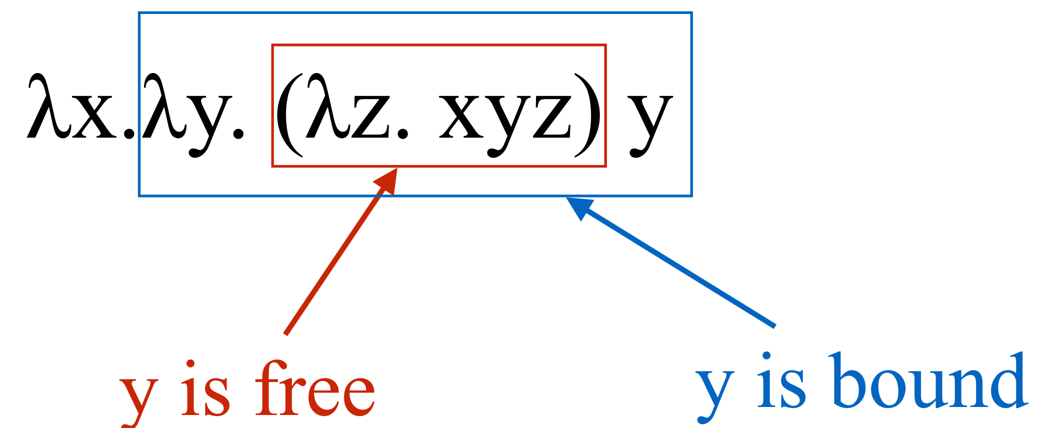


Review: Free and Bound Variables

Note:

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

Example:



"free" is relative to a λ-sub-term.

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)

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 bound occurrence of x in E with y

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 bound occurrence of x in E with y

$$\lambda \mathbf{m}.(\underline{\mathbf{m}} \text{ true}) \; \underline{\lambda n.(n\ M\ N)} \rightarrow_\beta \lambda n.(\underline{n}\ M\ N)\ \underline{\text{true}}$$

$$\rightarrow_\beta \text{true } M\ N$$

$$\equiv \lambda a.\lambda b.\underline{a}\ \ \underline{M}\ \ N$$

$$\rightarrow_\beta \lambda b.M\ N$$

$$\rightarrow_\beta M$$

function body E

true $\equiv \lambda a.\ \lambda b.\ a$

Function Application

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

α–reduction:

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

Example 1: $\lambda a.\lambda b.a+b\ 2\ x \quad \rightarrow_{\beta} \lambda b.2+b\ x$

$\rightarrow_{\beta} 2+x$

Function Application

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

α–reduction:

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

Example 2: $\lambda a.\lambda b.a+b\ b\ 2$ $\rightarrow_{\beta}$ $\lambda b.b+b\ 2$ → incorrect

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

Function Application

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

α–reduction:

$$(\lambda x.\mathbf{E}) \quad \to_{\alpha} \quad \lambda y.\mathbf{E[y/x]} \text{ if } y \notin \text{free}(E)$$

Example 2: $\lambda a.\underline{\lambda b.a+b}\ b\ 2 \quad \to_{\alpha} \lambda a.\underline{\lambda x.a+x}\ b\ 2$

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

$\to_{\beta}$ b+2

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):

≡ *abbreviated as*

true ≡ λa. λb. a *select-first*

false ≡ λa. λb. b *select-second*

if ≡ λp. λm. λn.(p m n)

if T 3 4

When T is true, return 3

if true 3 4

≡ λp.λm.λn.(p m n) true 3 4

≡ λp.λm.λn.(p m n) λa.λb.a 3 4

$\rightarrow_\beta$ λa.λb.a 3 4

$\rightarrow_\beta$ 3

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*

not ≡ λx. (x false true)

if x is **true** return **false**

not true
≡ λx. (x false true) true
$\rightarrow_\beta$ true false true
≡ λa.λb.a false true
$\rightarrow_\beta$ false

if x is **false** return **true**

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*

and ≡ λx. λy. (x y false)

if x is **true** return **y**		and true y	if x is **false** return **false**		and false y
	$\rightarrow_\beta$	true y false		$\rightarrow_\beta$	false y false
	≡	λa.λb.a y false		≡	λa.λb.b y false
	$\rightarrow_\beta$	y		$\rightarrow_\beta$	false

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

Programming in Lambda Calculus

What about data structures?

Data structures:

pairs can be represented as:

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

first $\equiv \lambda x.\ (x\ \text{true})$ (car)

$$\begin{aligned}\text{first}\ [M.N] &\equiv \lambda x.(x\ \text{true})\ \lambda z.(z\ M\ N)\\ &\rightarrow_\beta \lambda z.(z\ M\ N)\ \text{true}\\ &\rightarrow_\beta \text{true}\ M\ N\\ &\rightarrow_\beta M\end{aligned}$$

Programming in Lambda Calculus

What about data structures?

Data structures:

pairs can be represented as:

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

second $\equiv$ $\lambda x.\ (x\ \text{false})$ (cdr)

$$\begin{aligned} \text{second } [M.N] &\equiv \lambda x.(x\ \text{false})\ \lambda z.(z\ M\ N) \\ &\rightarrow_\beta \lambda z.(z\ M\ N)\ \text{fase} \\ &\rightarrow_\beta \text{false}\ M\ N \\ &\rightarrow_\beta N \end{aligned}$$

Programming in Lambda Calculus

What about data structures?

Data structures:

pairs can be represented as:

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

first $\equiv$ $\lambda x. (x\ \text{true})$ (car)
second $\equiv$ $\lambda x. (x\ \text{false})$ (cdr)
build $\equiv$ $\lambda x.\lambda y.\lambda z. (z\ x\ y)$ (cons)

Programming in Lambda Calculus

What about the encoding of arithmetic constants?

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? $\equiv \lambda m.\ (m\ \text{false not false})$

Programming in Lambda Calculus

Example:

(mult 2 3)

≡ ((λmn.(λfx.((m (n f)) x))) 2 3)

m = 2

n = 3

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta,}$ $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

m = 2

n = 3

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta}$, $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta}$, $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0{}^3 x))\ )\ x_0)$

Programming in Lambda Calculus

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

Example:

$(\text{mult}\ 2\ 3)$

$\equiv\quad ((\lambda mn.(\ \lambda fx.((m\ (n\ f))\ x)\ ))\ 2\ 3)$

$\rightarrow_{\beta},\ \lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv\quad \lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}\ \lambda f_0 x_0.((2\ (\lambda x.(f_0{}^3 x))\ \ )\ x_0)$

$\rightarrow_{\alpha}\ \lambda f_0 x_0.(\ (\boxed{2}\ (\lambda x_1.(f_0{}^3\ x_1))\ )\ \ x_0)$

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta,}$ $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0{}^3 x))\ \ )\ x_0)$

$\rightarrow_{\alpha}$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0{}^3\ x_1))\ )\ \ x_0)$

f in 2 ≡ λ**f**x.(**f** (**f** x))

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta}$, $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0^3 x))\ \ )\ x_0)$

$\rightarrow_{\alpha}$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0^3\ x_1))\ )\ \ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.(\ (\lambda x_1.(f_0^3\ x_1))\ (\ (\lambda x_1.(f_0^3\ x_1)\ )\ x)\ \ \ ))\ x_0) =$

f in 2 $\equiv$ λ**f**x.(**f** (**f** x))

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta,}$ $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0^3 x))\ \ )\ x_0)$

$\rightarrow_{\alpha}$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0^3\ x_1))\ )\ \ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.(\ (\lambda x_1.(f_0^3\ x_1))\ (\ (\lambda x_1.(f_0^3\ x_1)\ )\ x)\ \ \ ))\ x_0) =$

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta,}$ $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0^3 x))\ \ )\ x_0)$

$\rightarrow_{\alpha}$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0^3\ x_1))\ )\ \ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.(\ (\lambda x_1.(f_0^3\ x_1))\ (\ (\lambda x_1.(f_0^3\ x_1)\ )\ x)\ \ ))\ x_0) =$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.\ ((\lambda x_1.(f_0^3\ x_1))\ (f_0^3\ x)\ )\ \ )\ x_0)$

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\to_{\beta,}$ $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\to_\beta$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0^3 x))\ \ )\ x_0)$

$\to_\alpha$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0^3\ x_1))\ )\ \ x_0)$

$\to_\beta$ $\lambda f_0 x_0.((\lambda x.(\ (\lambda x_1.(f_0^3\ x_1))\ (\ (\lambda x_1.(f_0^3\ x_1)\ )\ x)\ \ \ ))\ x_0) =$

$\to_\beta$ $\lambda f_0 x_0.((\lambda x.\ ((\lambda x_1.(f_0^3\ x_1))\ (f_0^3\ x)\ )\ \ )\ x_0)$

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta,}$ $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0^3 x))\ \ )\ x_0)$

$\rightarrow_{\alpha}$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0^3\ x_1))\ )\ \ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.(\ (\lambda x_1.(f_0^3\ x_1))\ (\ (\lambda x_1.(f_0^3\ x_1)\ )\ x)\ \ \ ))\ x_0) =$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.\ ((\lambda x_1.(f_0^3\ x_1))\ (f_0^3\ x)\ )\ \ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.\ ((\ \lambda x.(\ f_0^3\ (f_0^3\ x)\ )\ )\ x_0)$

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ $((\lambda mn.(\ \lambda fx.((m\ (n\ f))\ x)\ ))\ 2\ 3)$

$\rightarrow_{\beta}$, $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0{}^3 x))\ \ )\ x_0)$

$\rightarrow_{\alpha}$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0{}^3\ x_1))\ )\ \ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.(\ (\lambda x_1.(f_0{}^3\ x_1))\ (\ (\lambda x_1.(f_0{}^3\ x_1)\ )\ x)\ \ \ ))\ x_0) =$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.\ ((\lambda x_1.(f_0{}^3\ x_1))\ (f_0{}^3\ x)\ )\ \ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.\ ((\ \lambda x.(\ f_0{}^3\ (f_0{}^3\ x)\ )\ )\ x_0)$

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta,}$ $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0^3 x))\ \ )\ x_0)$

$\rightarrow_{\alpha}$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0^3\ x_1))\ )\ \ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.(\ (\lambda x_1.(f_0^{\ 3}\ x_1))\ (\ (\lambda x_1.(f_0^3\ x_1)\ )\ x)\ \ \ ))\ x_0) =$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.\ ((\lambda x_1.(f_0^3\ x_1))\ (f_0^3\ x)\ )\ \ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.\ ((\ \lambda x.(\ f_0^3\ (f_0^3\ x)\ )\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.(f_0^{\ 3}\ (f_0^{\ 3}\ x_0))$

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\rightarrow_{\beta,}$ $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0^3 x))\ \ )\ x_0)$

$\rightarrow_{\alpha}$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0^3\ x_1))\ )\ \ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.(\ (\lambda x_1.(f_0^3\ x_1))\ (\ (\lambda x_1.(f_0^3\ x_1)\ )\ x)\ \ \ ))\ x_0) =$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.((\lambda x.\ ((\lambda x_1.(f_0^3\ x_1))\ (f_0^3\ x)\ )\ \ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.\ ((\ \lambda x.(\ f_0^3\ (f_0^3\ x)\ )\ )\ x_0)$

$\rightarrow_{\beta}$ $\lambda f_0 x_0.(f_0^3\ (f_0^3\ x_0))$

Programming in Lambda Calculus

Example:

(mult 2 3)

$\equiv$ ((λmn.(λfx.((m (n f)) x))) 2 3)

$\to_{\beta,}$ $\lambda f_0 x_0.((2\ (\mathbf{3}\ f_0)\ )\ x_0)$

$\equiv$ $\lambda f_0 x_0.((2\ (\lambda fx.(f^3 x)\ f_0)\ )\ x_0)$

$\to_\beta$ $\lambda f_0 x_0.((2\ (\lambda x.(f_0^3 x))\ )\ x_0)$

$\to_\alpha$ $\lambda f_0 x_0.(\ (2\ (\lambda x_1.(f_0^3\ x_1))\ )\ x_0)$

$\to_\beta$ $\lambda f_0 x_0.((\lambda x.(\ (\lambda x_1.(f_0^3\ x_1))\ (\ (\lambda x_1.(f_0^3\ x_1)\ )\ x)\ ))\ x_0) =$

$\to_\beta$ $\lambda f_0 x_0.((\lambda x.\ ((\lambda x_1.(f_0^3\ x_1))\ (f_0^3\ x)\ )\ )\ x_0)$

$\to_\beta$ $\lambda f_0 x_0.\ ((\ \lambda x.(\ f_0^3\ (f_0^3\ x)\ )\ )\ x_0)$

$\to_\beta$ $\lambda f_0 x_0.(f_0^3\ (f_0^3\ x_0))$

$\to_\alpha$ $\lambda fx.(f^6\ x) = 6$

Recursion in Lambda Calculus

Does this make sense?

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

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

Recursion in Lambda Calculus

Does this make sense?

$$f \equiv \ldots f \ldots$$

In lambda calculus, ≡ is "abbreviated as", **but not an assignment.**

add ≡ λmn. (if (isZero?n) m (**add** (m+1) (n-1)))

Incorrect!

How about

ADD ≡ λf.(λmn.(if (isZero?n) m (**f** (m+1) (n-1)))) such that

ADD f m n = (if (isZero?n) m (**f** (m+1) (n-1))

f = **ADD f**

ADD f m n = (if (isZero?n) m (**ADD f** (m+1) (n-1))

Function Fixed Points

The fixed point of a function g is the set of values $fix(g) = \{x \mid x = g(x)\}$.

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 λ–term Y that "computes" a fixed point of any function F

$$YF = F(YF)$$

YES. 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 λ–term Y that "computes" a fixed point of any function F

$$YF = F(YF)$$

YES. 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 λ–term Y that "computes" a fixed point of any function F

$$YF = F(YF)$$

YES. 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 λ–term Y that "computes" a fixed point of any function F

$$YF = F(YF)$$

YES. 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 λ–term Y that "computes" a fixed point of any function F

$$YF = F(YF)$$

YES. Y is called the fixed point combinator.

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

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

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

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

Function Fixed Points

Is there a λ–term Y that "computes" a fixed point of any function F

$$YF = F(YF)$$

YES. Y is called the fixed point combinator.

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

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

Function Fixed Points

Is there a λ–term Y that “computes” a fixed point of any function F

$$YF = F(YF)$$

YES. Y is called the fixed point combinator.

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

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

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

<u>Example:</u>

add ≡ λ**f**. (λmn. if (isZero? n) then m else (**f** (succ m) (pred n)))

((YF)3 2) =
(((λf.((λx.f(x x)) (λx.f(x x)))) F) 3 2) =

((F((λx.F(x x)) (λx.F(x x)))) 3 2) =

((λmn.if (isZero? n) then m else
(((λx.F(x x)) (λx.F(x x))) (succ m) (pred n))) 3 2) =

if (isZero? 2) then 3 else
(((λx.F(x x)) (λx.F(x x))) (succ 3) (pred 2)) =

(((λx.F(x x)) (λx.F(x x))) 4 1) =

((F((λx.F(x x)) (λx.F(x x)))) 4 1) = if (isZero? 1) then 4 else

(((λx.F(x x)) (λx.F(x x))) (succ 4) (pred 1)) =

(((λx.F(x x)) (λx.F(x x))) 5 0) =

((F((λx.F(x x)) (λx.F(x x)))) 5 0) = if (isZero? 0) then 5 else

(((λx.F(x x)) (λx.F(x x))) (succ 5) (pred 0)) = **5**

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 ≡ λxy.x
  - S ≡ λxyz.xz(yz)

Combinatory logic is as powerful as Turing Machines.

Next Lecture

Reading:

- Scott, Chapter 11.7 (4th Edition supplementary chapters)