CS 314 Principles of Programming Languages

Lecture 19: Lambda Calculus

Zheng (Eddy) Zhang

Rutgers University

April 11, 2018

Lambda Calculus: Historical Origin

- The **imperative** and **functional** models grew out of work undertaken by Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, and etc in 1930s.
  - Different formalizations of the notion of an algorithm, or "effective procedure", based on *automata*, *symbolic manipulation*, *recursive function definitions*, and *combinatorics*.
- These results led Church to conjecture that:

"Any intuitively appealing model of computing would be equally powerful as well."

— *Church's thesis*

Lambda Calculus: Historical Origin

- **Turing's model of computing was the *Turing machine* a sort of pushdown automaton using an unbounded storage "tape"**

The Turing machine computes in an imperative way, by changing the values in cells of its tape – like variables just as a high level imperative program computes by changing the values of variables.

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*.

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

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
expo → name | number | λ name . expr | func arg
func → name | (λ name . expr) | func arg
arg → name | number | (λ name . expr) | (func arg)

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

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

λ-terms

The context-free grammar for λ-terms:

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

$$(\ \underset{\text{func}}{\boxed{\mathbf{y}}}\ \ \underset{\text{arg}}{\boxed{\mathbf{z}}}\ )$$

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.

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

free occurrence

binding occurrence

bound occurrence

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)

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.

Function Application as Substitution

The result of applying an abstraction (λx.M) to an argument M is formalized by a special form of textual substitution.

$$\mathbf{(\lambda x.\ M)\ N} \quad \cong \quad [N/x]M$$

Informally, N replaces all free occurrences of x in M.

Function Application as Substitution

The result of applying an abstraction ($\lambda x.M$) to an argument M is formalized by a special form of textual substitution.

$$\mathbf{(\lambda x.\ M)\ N} \quad \cong \quad [N/x]M$$

Informally, N replaces all free occurrences of x in M.

Example:

$$(\lambda a.\ \lambda b.\ a+b)2\ x \cong (\lambda b.2+b)x$$

$$\cong 2 + x$$

Function Application

Computation in the lambda calculus is based on the concept or reduction (rewriting rules). The goal is to "simplify" an expression until it can no longer be further simplified.

$(\lambda x.M)N \Rightarrow_\beta [N/x]M$ (β–reduction)

$(\lambda x.M) \Rightarrow_\alpha \lambda y.[y/x]M$ (α–reduction), if $y \notin free(M)$

Note:

- An equivalence relation can be defined based on $\cong$-convertible λ-terms. "Reduction" rules really work both ways, but we are interested in reducing the complexity of λ-term (forward direction)
- α–reduction does not reduce the complexity of λ-term
- β–reduction: corresponds to application, models computation

Reduction

- A subterm of the form (λx.M)N is called a *redex* (reduction expression)
- A reduction is any sequence of **α–reductions** and **β–reductions**
- A term that cannot be **β–reduced** are said to be in **β normal form**
- A subterm that is an abstraction or a variable is said to be in **head normal form** .

Question: Does a normal form always exist?

Example:

((λx.xx)) (λx.xx)))

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 $\equiv \lambda a.\ \lambda b.\ a$ *select-first*

false $\equiv \lambda a.\ \lambda b.\ b$ *select-second*

cond $\equiv \lambda m.\ \lambda n.\ \lambda p.(p\ m\ n)$

if p is **true** return **m**

cond p m n
$\cong$ p m n
$\cong \lambda a.\lambda b.a\ m\ n$
$\cong$ m

if p is **false** return **n**

cond p m n
$\cong$ p m n
$\cong \lambda a.\lambda b.b\ m\ n$
$\cong$ n

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 $\equiv \lambda a.\ \lambda b.\ a$ *select-first*

false $\equiv \lambda a.\ \lambda b.\ b$ *select-second*

not $\equiv \lambda x.\ (x \text{ false true})$

if x is **true** return **false**	**not** x
	$\cong \lambda x.\ (x \text{ false true})\ x$
	$\cong x \text{ false true}$
	$\cong \lambda a.\lambda b.a \text{ false true}$
	$\cong \text{false}$

if x is **false** return **true**	**not** x
	$\cong \lambda x.\ (x \text{ false true})\ x$
	$\cong x \text{ false true}$
	$\cong \lambda a.\lambda b.b \text{ false true}$
	$\cong \text{true}$

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** x y ≅ x y false ≅ λa.λb.a y false ≅ y	if x is **false** return **false**	**and** x y ≅ x y false ≅ λa.λb.b y false ≅ 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 ≡ λm. λn. λp.(p m n)

not ≡ λx. (x false true)

and ≡ λx. λy. (x y false)

or ≡ homework

Programming in Lambda Calculus

What about data structures?

$\boxed{\textit{Data structures:}}$

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:

$0 \equiv \lambda fx.\ x$

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

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

…

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

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

Next Lecture

Reading:

- Scott, Chapter 11.1 - 11.3 (4th Edition) or Chapter 10.1 - 10.3 (3rd Edition)
- Scott, Chapter 11.7 (4th Edition supplementary chapters)