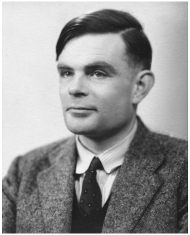
COSE212: Programming Languages

Lecture 13 — Untyped Lambda Calculus

Hakjoo Oh
2016 Fall

Origins of Computers and Programming Languages



- What is the original model of computers?
- What is the original model of programming languages?
- Which one came first?

cf) Church-Turing thesis:

$$\text{Lambda calculus} = \text{Turing machine}$$

Lambda Calculus

- The first, yet turing-complete, programming language
- Developed by Alonzo Church in 1936
- The core of functional programming languages (e.g., Lisp, ML, Haskell, Scala, etc)

Syntax of Lambda Calculus

$$\begin{array}{rcll} e & \rightarrow & x & \text{variables} \\ & | & \lambda x.e & \text{abstraction} \\ & | & e\ e & \text{application} \end{array}$$

- Examples:

$$\begin{array}{c} x \qquad y \qquad z \\ \lambda x.x \qquad \lambda x.y \qquad \lambda x.\lambda y.x \\ x\ y \qquad (\lambda x.x)\ z \qquad x\ \lambda y.z \qquad ((\lambda x.x)\ \lambda x.x) \end{array}$$

- Conventions when writing λ-expressions:
  1. Application associates to the left, e.g., $s\ t\ u = (s\ t)\ u$
  2. The body of an abstraction extends as far to the right as possible, e.g., $\lambda x.\lambda y.x\ y\ x = \lambda x.(\lambda y.((x\ y)\ x))$

Bound and Free Variables

- An occurrence of variable x is said to be *bound* when it occurs inside λx, otherwise said to be *free*.
  - $\lambda y.(x\ y)$
  - $\lambda x.x$
  - $\lambda z.\lambda x.\lambda x.(y\ z)$
  - $(\lambda x.x)\ x$
- Expressions without free variables is said to be *closed expressions* or *combinators*.

Evaluation

To evaluate $\boldsymbol{\lambda}$-expression e,

1. Find a sub-expression of the form:

$$(\lambda x.e_1)\ e_2$$

   Expressions of this form are called “redex” (reducible expression).

2. Rewrite the expression by substituting the e_2 for every free occurrence of x in e_1:

$$(\lambda x.e_1)\ e_2 \rightarrow [x \mapsto e_2]e_1$$

   This rewriting is called β-reduction

Repeat the above two steps until there are no redexes.

Evaluation

- $\lambda x.x$
- $(\lambda x.x)\ y$
- $(\lambda x.x\ y)$
- $(\lambda x.x\ y)\ z$
- $(\lambda x.(\lambda y.x))\ z$
- $(\lambda x.(\lambda x.x))\ z$
- $(\lambda x.(\lambda y.x))\ y$
- $(\lambda x.(\lambda y.x\ y))\ (\lambda x.x)\ z$

Evaluation Strategy

- In a lambda expression, multiple redexes may exist. Which redex to reduce next?

$$\lambda x.x\ (\lambda x.x\ (\lambda z.(\lambda x.x)\ z)) = id\ (id\ (\lambda z.id\ z))$$

redexes:

$$\underline{id\ (id\ (\lambda z.id\ z))}$$
$$id\ \underline{(id\ (\lambda z.id\ z))}$$
$$id\ (id\ (\lambda z.\underline{id\ z}))$$

- Evaluation strategies:
  - Full beta-reduction
  - Normal order
  - Call-by-name
  - Call-by-value

Full beta-reduction strategy

Any redex may be reduced at any time:

$$\begin{array}{ll} & id\ (id\ (\lambda z.\underline{id\ z})) \\ \rightarrow & id\ \underline{(id\ (\lambda z.z))} \\ \rightarrow & \underline{id\ (\lambda z.z)} \\ \rightarrow & \lambda z.z \\ \not\rightarrow & \end{array}$$

or,

$$\begin{array}{ll} & id\ (\underline{id\ (\lambda z.id\ z)}) \\ \rightarrow & \underline{id\ (\lambda z.id\ z)} \\ \rightarrow & \lambda z.\underline{id\ z} \\ \rightarrow & \lambda z.z \\ \not\rightarrow & \end{array}$$

The evaluation is non-deterministic.

Normal order strategy

Reduce the leftmost, outermost redex first:

$$\begin{array}{ll} & \underline{id\ (id\ (\lambda z.id\ z))} \\ \to & \underline{id\ (\lambda z.id\ z))} \\ \to & \lambda z.\underline{id\ z} \\ \to & \lambda z.z \\ \not\to & \end{array}$$

The evaluation is deterministic (i.e., partial function).

Call-by-name strategy

Follow the normal order reduction, not allowing reductions inside abstractions:

$$\begin{array}{rl} & \underline{id\ (id\ (\lambda z.id\ z))} \\ \to & \underline{id\ (\lambda z.id\ z))} \\ \to & \lambda z.id\ z \\ \not\to & \end{array}$$

The call-by-name strategy is *non-strict* (or *lazy*) in that it evaluates arguments that are actually used.

Call-by-value strategy

Reduce the outermost redex whose right-hand side has a *value* (a term that cannot be reduced any further):

$$\begin{array}{ll} & \boldsymbol{id}\ (\underline{\boldsymbol{id}\ (\boldsymbol{\lambda z.id\ z})}) \\ \rightarrow & \underline{\boldsymbol{id}\ (\boldsymbol{\lambda z.id\ z})} \\ \rightarrow & \boldsymbol{\lambda z.id\ z} \\ \not\rightarrow & \end{array}$$

The call-by-name strategy is *strict* in that it always evaluates arguments, whether or not they are used in the body.

Programming in the Lambda Calculus

- boolean values
- natural numbers
- pairs
- recursion
- ...

Church Booleans

- Boolean values:

$$\begin{array}{rcl} \text{true} & = & \lambda t.\lambda f.t \\ \text{false} & = & \lambda t.\lambda f.f \end{array}$$

- Conditional test:

$$\text{test} = \lambda l.\lambda m.\lambda n.l\ m\ n$$

- Then,

$$\text{test}\ b\ v\ w = \begin{cases} v & \text{if } b = \text{true} \\ w & \text{if } b = \text{false} \end{cases}$$

- Example:

$$\begin{array}{rcl} \text{test true}\ v\ w & = & (\lambda l.\lambda m.\lambda n.l\ m\ n)\ \text{true}\ v\ w \\ & = & (\lambda m.\lambda n.\text{true}\ m\ n)\ v\ w \\ & = & \text{true}\ v\ w \\ & = & (\lambda t.\lambda f.t)\ v\ w \\ & = & (\lambda f.v)\ w \\ & = & v \end{array}$$

Church Booleans

Logical operators:

- Logical "and":

$$\text{and} = \lambda b.\lambda c.(b\ c\ \text{false})$$

$$\begin{array}{rcl} \text{and true true} & = & \text{true} \\ \text{and true false} & = & \text{false} \\ \text{and false true} & = & \text{false} \\ \text{and false false} & = & \text{false} \end{array}$$

- (exercise) Logical "or" and "not"?

$$\begin{array}{rcl} \text{or true true} & = & \text{true} \\ \text{or true false} & = & \text{true} \\ \text{or false true} & = & \text{true} \\ \text{or false false} & = & \text{false} \end{array}$$

$$\begin{array}{rcl} \text{not true} & = & \text{false} \\ \text{not false} & = & \text{true} \end{array}$$

Pairs

Using booleans, we can encode pairs of values.

$$\begin{array}{rcl} \text{pair } v\ w & : & \text{create a pair of } v \text{ and } w \\ \text{fst } p & : & \text{select the first component of } p \\ \text{snd } p & : & \text{select the second component of } p \end{array}$$

- Definition:

$$\begin{array}{rcl} \text{pair} & = & \lambda f.\lambda s.\lambda b.b\ f\ s \\ \text{fst} & = & \lambda p.p \text{ true} \\ \text{snd} & = & \lambda p.p \text{ false} \end{array}$$

- Example:

$$\begin{array}{rcl} \text{fst (pair } v\ w) & = & \text{fst } ((\lambda f.\lambda s.\lambda b.b\ f\ s)\ v\ w) \\ & = & \text{fst } (\lambda b.b\ v\ w) \\ & = & (\lambda p.p \text{ true})\ (\lambda b.b\ v\ w) \\ & = & (\lambda b.b\ v\ w) \text{ true} \\ & = & \text{true } v\ w \\ & = & v \end{array}$$

Church Numerals

$$
\begin{aligned}
c_0 &= \lambda s.\lambda z.z \\
c_1 &= \lambda s.\lambda z.(s\ z) \\
c_2 &= \lambda s.\lambda z.s\ (s\ z) \\
&\vdots \\
c_n &= \lambda s.\lambda z.s^n\ z
\end{aligned}
$$

Church Numerals

- Successor:

$$\text{succ } c_i = c_{i+1}$$

Definition:

$$\text{succ} = \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$$

Example:

$$\begin{aligned} \text{succ } c_0 &= \lambda n.\lambda s.\lambda z.(s\ (n\ s\ z))\ c_0 \\ &= \lambda s.\lambda z.(s\ (c_0\ s\ z)) \\ &= \lambda s.\lambda z.(s\ z) \\ &= c_1 \end{aligned}$$

Church Numeral

- Addition:

$$\text{plus } c_n \ c_m = c_{n+m}$$

Definition:

$$\text{plus} = \lambda n.\lambda m.\lambda s.\lambda z.m \ s \ (n \ s \ z)$$

Example:

$$\begin{array}{rcl} \text{plus } c_1 \ c_2 & = & \lambda s.\lambda z.c_2 \ s \ (c_1 \ s \ z) \\ & = & \lambda s.\lambda z.c_2 \ s \ (s \ z) \\ & = & \lambda s.\lambda z.s \ (s \ (s \ z)) \\ & = & c_3 \end{array}$$

Church Numerals

- Multiplication:

$$\text{mult}\ c_n\ c_m = c_{n*m}$$

Definition:

$$\text{mult} =$$

Church Numerals

- Multiplication:

$$\text{mult } c_n\ c_m = c_{n*m}$$

Definition:

$$\text{mult} =$$

Example:

$$\begin{array}{rcl} \text{mult } c_1\ c_2 & = & (\lambda m.\lambda n.m\ (\text{plus } n)\ c_0)\ c_1\ c_2 \\ & = & c_1\ (\text{plus } c_2)\ c_0 \\ & = & (\text{plus } c_2)\ c_0 \\ & = & (\lambda m.\lambda s.\lambda z.m\ s\ (c_2\ s\ z))\ c_0 \\ & = & \lambda s.\lambda z.c_0\ s\ (c_2\ s\ z) \\ & = & \lambda s.\lambda z.c_2\ s\ z \\ & = & \lambda s.\lambda z.s\ (s\ z) \end{array}$$

- Other definition:

$$\text{mult}\mathbf{2} = \lambda m.\lambda n.\lambda s.\lambda z.m\ (n\ s)\ z$$

- Power (n^m):

$$\text{power} = \lambda m.\lambda n.m\ (\text{mult } n)\ c_1$$

Church Numerals

- Testing zero:

$$\begin{aligned} \text{zero? } c_0 &= \text{true} \\ \text{zero? } c_1 &= \text{false} \end{aligned}$$

Definition:

$$\text{zero? } =$$

Example:

$$\text{zero? } c_0 =$$

Recursion

- In lambda calculus, recursion is magically realized via Y-combinator:

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

- For example, the factorial function

$$\mathsf{f}(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n * \mathsf{f}(n-1)$$

  is encoded by

$$\text{fact} = Y(\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1))$$

  Then, fact n computes $n!$.

- Recursive functions can be encoded by composing non-recursive functions!

Recursion

Let $F = \lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$ and $G = \lambda x.F(x\ x)$.

$$
\begin{aligned}
&\text{fact } 1\\
&= (Y\ F)\ 1\\
&= (\lambda f.((\lambda x.f(x\ x))(\lambda x.f(x\ x)))\ F)\ 1\\
&= ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ 1\\
&= (G\ G)\ 1\\
&= (F\ (G\ G))\ 1\\
&= (\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (G\ G)(n-1))\ 1\\
&= \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * (G\ G)(1-1))\\
&= \text{if false then } 1 \text{ else } 1 * (G\ G)(1-1))\\
&= 1 * (G\ G)(1-1)\\
&= 1 * (F\ (G\ G))(1-1)\\
&= 1 * (\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n * (G\ G)(n-1))(1-1)\\
&= 1 * \text{if } (1-1) = 0 \text{ then } 1 \text{ else } (1-1) * (G\ G)((1-1)-1)\\
&= 1 * 1
\end{aligned}
$$

Summary

- $\boldsymbol{\lambda}$-calculus is a simple and minimal language.
  - Syntax: $e \rightarrow x \mid \lambda x.e \mid e\ e$
  - Semantics: β-reduction
- Yet, $\boldsymbol{\lambda}$-calculus is Turing-complete.
  - E.g., ordinary values (e.g., boolean, numbers, pairs, etc) can be encoded in $\boldsymbol{\lambda}$-calculus (see in the next class).
- Church-Turing thesis:

$$\begin{array}{rcl} e & \rightarrow & x \\ & \mid & \lambda x.e \\ & \mid & e\ e \end{array} \quad = $$