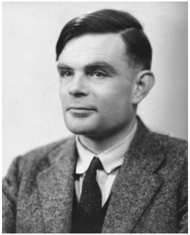
COSE212: Programming Languages

Lecture 13 — Lambda Calculus (1)

Hakjoo Oh
2015 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

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

- Examples:

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

- 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 λ-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 until there are no redexes.

Evaluation

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

Formal Definition of Substitution

The substitution $[x \mapsto e_1]e_2$ is inductively defined on the structure of e_2:

$$
\begin{array}{lcll}
[x \mapsto e_1]x & = & & \\
[x \mapsto e_1]y & = & & \text{if } x \neq y \\
[x \mapsto e_1](\lambda x.e) & = & & \\
[x \mapsto e_1](\lambda y.e) & = & & \text{if } x \neq y \\
[x \mapsto e_1](e_2\ e_3) & = & &
\end{array}
$$

Examples:

$$
\begin{array}{lcl}
[x \mapsto y]\lambda x.x & \neq & \lambda x.y \\
[x \mapsto y]\lambda x.x & = & \lambda z.[x \mapsto y][x \mapsto z]x \\
 & = & \lambda z.z
\end{array}
$$

$$
\begin{array}{lcl}
[y \mapsto x]\lambda x.y & \neq & \lambda x.x \\
[y \mapsto x]\lambda x.y & = & \lambda z.[y \mapsto x][x \mapsto z]x \\
 & = & \lambda z.x
\end{array}
$$

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:

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

Normal order strategy

Reduce the leftmost, outermost redex first:

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

Call-by-name strategy

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

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

Call-by-value strategy

Reduce the outermost redex whose right-hand side has a *value*:

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

Normal Terms

- A lambda expression is said to have *normal term* if evaluating the expression terminates under an evaluation strategy.
- Does every lambda expression have normal term? e.g.,

$$(\lambda x.x\ x)(\lambda x.x\ x)$$

- The normal order strategy guarantees to reach the normal terms (if exists): e.g.,

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

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:

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