

CVO103: Programming Languages

Lecture 15 — Lambda Calculus (Origin of Programming Languages)

Hakjoo Oh
2018 Spring

Syntactic Sugar

- Syntactic sugar is syntax that makes a language “sweet”: it does not add expressiveness but makes programs easier to read and write.
- For example, we can “desugar” the `let` expression:

$$\text{let } x = E_1 \text{ in } E_2 \xrightarrow{\text{desugar}} (\text{proc } x E_2) E_1$$

- Exercise) Desugar the program:

```
let x = 1 in
  let y = 2 in
    x + y
```

Syntactic Sugar

Q) Identify all syntactic sugars of the language:

$$\begin{array}{l} E \rightarrow n \\ | \\ | x \\ | \\ | E + E \\ | \\ | E - E \\ | \\ | \text{iszero } E \\ | \\ | \text{if } E \text{ then } E \text{ else } E \\ | \\ | \text{let } x = E \text{ in } E \\ | \\ | \text{letrec } f(x) = E \text{ in } E \\ | \\ | \text{proc } x E \\ | \\ | E E \end{array}$$

Lambda Calculus (λ -Calculus)

- By removing all syntactic sugars from a language, we obtain a minimal language, called *lambda calculus*:

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

Origins of Programming Languages and Computer

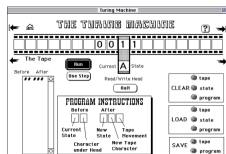


- In 1935, Church developed λ -calculus as a formal system for mathematical logic and argued that any computable function on natural numbers can be computed with λ -calculus. Since then, λ -calculus became the model of programming languages.
- In 1936, Turing independently developed Turing machine and argued that any computable function on natural numbers can be computed with the machine. Since then, Turing machine became the model of computers.

Church-Turing Thesis

- A surprising fact is that the classes of λ -calculus and Turing machines can compute coincide even though they were developed independently.
- Church and Turing proved that the classes of computable functions defined by λ -calculus and Turing machine are equivalent.

$$\begin{array}{l} e \rightarrow x \\ | \lambda x.e \\ | e e \end{array} =$$



A function is λ -computable if and only if Turing computable.

- This equivalence has led mathematicians and computer scientists to believe that these models are “universal”: A function is computable if and only if λ -computable if and only if Turing computable.

Impact of λ -Calculus

λ -calculus had immense impacts on programming languages.

- It has been the core of functional programming languages (e.g., Lisp, ML, Haskell, Scala, etc).
- Lambdas in other languages:

- ▶ Java8

```
(int n, int m) -> n + m
```

- ▶ C++11

```
[](int x, int y) { return x + y; }
```

- ▶ Python

```
(lambda x, y: x + y)
```

- ▶ JavaScript

```
function (a, b) { return a + b }
```

Syntax of Lambda Calculus

e	\rightarrow	x	variables
		$\lambda x.e$	abstraction
		$e e$	application

- Examples:

$$\begin{array}{ccccc} & & x & y & z \\ & & \lambda x.x & \lambda x.y & \lambda x.\lambda y.x \\ x y & (\lambda x.x) z & x \lambda y.z & ((\lambda x.x) \lambda x.x) \end{array}$$

- 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 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:

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

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

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

- Evaluation strategies:
 - ▶ Normal order
 - ▶ Call-by-name
 - ▶ Call-by-value

Normal order strategy

Reduce the leftmost, outermost redex first:

$$\begin{aligned} & id (id (\lambda z.id z)) \\ \rightarrow & \frac{id (id (\lambda z.id z))}{id (\lambda z.id z)} \\ \rightarrow & \lambda z.id z \\ \rightarrow & \lambda z.z \\ \not\rightarrow & \end{aligned}$$

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

Call-by-name strategy

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

$$\begin{aligned} & id (id (\lambda z.id z)) \\ \rightarrow & \frac{id (\lambda z.id z)}{id (\lambda z.id z)} \\ \rightarrow & \lambda z.id z \\ \not\rightarrow & \end{aligned}$$

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):

$$\begin{aligned} & id (id (\lambda z.id z)) \\ \rightarrow & \frac{id (id (\lambda z.id z))}{id (\lambda z.id z)} \\ \rightarrow & \lambda z.id z \\ \not\rightarrow & \end{aligned}$$

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

Compiling to Lambda Calculus

Consider the source language:

$$\begin{array}{l} E \rightarrow \text{true} \\ | \text{false} \\ | n \\ | x \\ | E + E \\ | \text{iszero } E \\ | \text{if } E \text{ then } E \text{ else } E \\ | \text{let } x = E \text{ in } E \\ | \text{letrec } f(x) = E \text{ in } E \\ | \text{proc } x E \\ | E E \end{array}$$

Define the translation procedure from E to λ -calculus.

Compiling to Lambda Calculus

E : the translation result of E in λ -calculus

$$\begin{aligned}\underline{true} &= \lambda t. \lambda f. t \\ \underline{false} &= \lambda t. \lambda f. f \\ \underline{0} &= \lambda s. \lambda z. z \\ \underline{1} &= \lambda s. \lambda z. (s z) \\ \underline{n} &= \lambda s. \lambda z. (s^n z) \\ \underline{x} &= x \\ \underline{E_1 + E_2} &= (\lambda n. \lambda m. \lambda s. \lambda z. m s (n s z)) \underline{E_1} \underline{E_2} \\ \underline{\text{iszero } E} &= (\lambda m. m (\lambda x. \underline{false}) \underline{true}) \underline{E} \\ \underline{\text{if } E_1 \text{ then } E_2 \text{ else } E_3} &= \underline{E_1} \underline{E_2} \underline{E_3} \\ \underline{\text{let } x = E_1 \text{ in } E_2} &= (\lambda x. \underline{E_2}) \underline{E_1} \\ \underline{\text{letrec } f(x) = E_1 \text{ in } E_2} &= \underline{\text{let } f = Y (\lambda f. \lambda x. E_1) \text{ in } E_2} \\ \underline{\text{proc } x E} &= \lambda x. \underline{E} \\ \underline{E_1 E_2} &= \underline{E_1} \underline{E_2}\end{aligned}$$

Correctness of Compilation

Theorem

For any expression E ,

$$\llbracket \underline{E} \rrbracket = \llbracket \underline{E} \rrbracket$$

where $\llbracket E \rrbracket$ denotes the value that results from evaluating E .

Examples: Booleans

$$\begin{aligned}\underline{\text{if } true \text{ then } 0 \text{ else } 1} &= \underline{true} \underline{0} \underline{1} \\ &= (\lambda t. \lambda f. t) \underline{0} \underline{1} \\ &= \underline{0} \\ &= \lambda s. \lambda z. z\end{aligned}$$

Note that

$$\llbracket \underline{\text{if } true \text{ then } 0 \text{ else } 1} \rrbracket = \llbracket \underline{\text{if } true \text{ then } 0 \text{ else } 1} \rrbracket$$

Example: Numerals

$$\begin{aligned}\underline{1 + 2} &= (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ \underline{1}\ \underline{2} \\ &= \lambda s.\lambda z.\underline{2}\ s\ (\underline{1}\ s\ z) \\ &= \lambda s.\lambda z.\underline{2}\ s\ (\lambda s.\lambda z.(s\ z)\ s\ z) \\ &= \lambda s.\lambda z.\underline{2}\ s\ (s\ z) \\ &= \lambda s.\lambda z.(\lambda s.\lambda z.(s\ (s\ z)))\ s\ (s\ z) \\ &= \lambda s.\lambda z.s\ (s\ (s\ z)) \\ &= \underline{3}\end{aligned}$$

Recursion

- For example, the factorial function

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

is encoded by

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

where Y is the Y-combinator (or fixed point combinator):

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

- Then, $\text{fact } n$ 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
 $G = \lambda x.F(x x)$.

fact 1

$$= (Y F) 1$$

$$= (\lambda f.((\lambda x.f(x x))(\lambda x.f(x x)))) F) 1$$

$$= ((\lambda x.F(x x))(\lambda x.F(x x))) 1$$

$$= (G G) 1$$

$$= (F (G G)) 1$$

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

$$= \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * (G G)(1 - 1)$$

$$= \text{if false then } 1 \text{ else } 1 * (G G)(1 - 1)$$

$$= 1 * (G G)(1 - 1)$$

$$= 1 * (F (G G))(1 - 1)$$

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

$$= 1 * \text{if } (1 - 1) = 0 \text{ then } 1 \text{ else } (1 - 1) * (G G)((1 - 1) - 1)$$

$$= 1 * 1$$

Summary

- λ -calculus is a minimal programming language.
 - ▶ Syntax: $e \rightarrow x \mid \lambda x.e \mid e e$
 - ▶ Semantics: β -reduction
- Yet, λ -calculus is Turing-complete.

$$e \rightarrow \begin{array}{l} x \\ \lambda x.e \\ e e \end{array} =$$

