COSE212: Programming Languages

Lecture 17 — Lambda Calculus (Origin of Programming Languages)

Hakjoo Oh
2019 Fall

A Fundamental Question

Programming languages look very different.

- C, C++, Java, OCaml, Haskell, Scala, JavaScript, etc

Example: QuickSort in C

```
void swap(int* a, int* b) { int t = *a; *a = *b; *b = t; }

int partition (int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high- 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Example: QuickSort in Haskell

```
quicksort [] = []
quicksort (x:xs) = quicksort ys ++ [x] ++ quicksort zs
                where
                    ys = [a | a <- xs, a <=x]
                    zs = [b | b <- xs, b > x]
```

A Fundamental Question

Are they different fundamentally? or Is there a core mechanism underlying all programming languages?

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:

$$\texttt{let}\ x = E_1\ \texttt{in}\ E_2 \overset{desugar}{\Longrightarrow} (\texttt{proc}\ x\ E_2)\ E_1$$

- Exercise) Desugar the program:

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

Syntactic Sugar

Q) Identify all syntactic sugars of the language:

$$\begin{array}{rcl} E & \to & n \\ & \mid & x \\ & \mid & E + E \\ & \mid & E - E \\ & \mid & \texttt{iszero}\ E \\ & \mid & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\ & \mid & \texttt{let}\ x = E\ \texttt{in}\ E \\ & \mid & \texttt{letrec}\ f(x) = E\ \texttt{in}\ E \\ & \mid & \texttt{proc}\ x\ E \\ & \mid & E\ E \end{array}$$

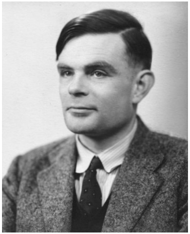
Lambda Calculus ($\boldsymbol{\lambda}$-Calculus)

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

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

$$\boxed{\text{Programming language} = \text{Lambda calculus} + \text{Syntactic sugars}}$$

Origins of Programming Languages and Computer

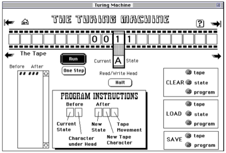


- In 1935, Church developed $\boldsymbol{\lambda}$-calculus as a formal system for mathematical logic and argued that any computable function on natural numbers can be computed with $\boldsymbol{\lambda}$-calculus. Since then, $\boldsymbol{\lambda}$-calculus became the model of programming languages.
- 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 $\boldsymbol{\lambda}$-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 $\boldsymbol{\lambda}$-calculus and Turing machine are equivalent.

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



A function is $\boldsymbol{\lambda}$-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 $\boldsymbol{\lambda}$-computable if and only if Turing computable.

λ-Calculus is Everywhere

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

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

- 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 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:

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

   This rewriting is called β-reduction

Repeat the above two steps until there are no redexes.

Evaluation

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

Substitution

The definition of $[x \mapsto e_1]e_2$:

$$\begin{array}{rcl} [x \mapsto e_1]x & = & e_1 \\ [x \mapsto e_1]y & = & y \\ [x \mapsto e_1](\lambda y.e_2) & = & \lambda z.[x \mapsto e_1]([y \mapsto z]e_2) \;\; (\text{new } z) \\ [x \mapsto e_1](e_2\ e_3) & = & ([x \mapsto e_1]e_2\ [x \mapsto e_1]e_3) \end{array}$$

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:
  - Normal order
  - Call-by-name
  - Call-by-value

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:

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

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):

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

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

Compiling to Lambda Calculus

Consider the source language:

$$
\begin{array}{rcl}
E & \rightarrow & \mathit{true} \\
  & \mid & \mathit{false} \\
  & \mid & n \\
  & \mid & x \\
  & \mid & E + E \\
  & \mid & \texttt{iszero}\ E \\
  & \mid & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
  & \mid & \texttt{let}\ x = E\ \texttt{in}\ E \\
  & \mid & \texttt{letrec}\ f(x) = E\ \texttt{in}\ E \\
  & \mid & \texttt{proc}\ x\ E \\
  & \mid & E\ E
\end{array}
$$

Define the translation procedure from E to λ-calculus.

Compiling to Lambda Calculus

$\underline{E}$: the translation result of E in λ-calculus

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

Correctness of Compilation

Theorem

For any expression $\boldsymbol{E}$,

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

where $\llbracket \boldsymbol{E} \rrbracket$ *denotes the value that results from evaluating* $\boldsymbol{E}$.

Examples: Booleans

$$\begin{array}{rcl}\underline{\texttt{if } \boldsymbol{true} \texttt{ then } \mathbf{0} \texttt{ else } \mathbf{1}} & = & \underline{\boldsymbol{true}}\ \underline{\mathbf{0}}\ \underline{\mathbf{1}} \\ & = & (\boldsymbol{\lambda t.\lambda f.t})\ \underline{\mathbf{0}}\ \underline{\mathbf{1}} \\ & = & \underline{\mathbf{0}} \\ & = & \boldsymbol{\lambda s.\lambda z.z}\end{array}$$

Note that

$$[\![\underline{\texttt{if } \boldsymbol{true} \texttt{ then } \mathbf{0} \texttt{ else } \mathbf{1}}]\!] = \underline{[\![\texttt{if } \boldsymbol{true} \texttt{ then } \mathbf{0} \texttt{ else } \mathbf{1}]\!]}$$

Exercises

Define the translation for the boolean operations:

- $\underline{\boldsymbol{E_1} \text{ and } \boldsymbol{E_2}} =$
- $\underline{\boldsymbol{E_1} \text{ or } \boldsymbol{E_2}} =$
- $\underline{\text{not } \boldsymbol{E}} =$

Example: Numerals

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

Exercises

Define the translation for the boolean operations:

- $\underline{\text{succ } \boldsymbol{E}} =$
- $\underline{\text{pred } \boldsymbol{E}} =$
- $\underline{\boldsymbol{E_1 * E_2}} =$
- $\underline{\boldsymbol{E_1^{E_2}}} =$

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, 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

$$\boxed{\text{Programming language} = \text{Lambda calculus} + \text{Syntactic sugars}}$$

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

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