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

```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:

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

- Exercise) Desugar the program:

```plaintext
let x = 1 in
  let y = 2 in
    x + y
```
Q) Identify all syntactic sugars of the language:

\[ E \rightarrow n \]

\[ x \]

\[ E + E \]

\[ E - E \]

\[ \text{iszero } E \]

\[ \text{if } E \text{ then } E \text{ else } E \]

\[ \text{let } x = E \text{ in } E \]

\[ \text{letrec } f(x) = E \text{ in } E \]

\[ \text{proc } x \ E \]

\[ E \ E \]
Lambda Calculus (\(\lambda\)-Calculus)

- By removing all syntactic sugars from the language, we obtain a minimal language, called \textit{lambda calculus}:

\[
e \rightarrow x \quad \text{variables} \\
| \lambda x.e \quad \text{abstraction} \\
| ee \quad \text{application}
\]

Programming language = Lambda calculus + Syntactic sugars
In 1935, Church developed $\lambda$-calculus as a formal system for mathematical logic and argued that any computable function on natural numbers can be computed with $\lambda$-calculus. Since then, $\lambda$-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 \( \lambda \)-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 \( \lambda \)-calculus and Turing machine are equivalent.

A function is \( \lambda \)-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 \( \lambda \)-computable if and only if Turing computable.
\textbf{\textit{\lambda}-Calculus is Everywhere}

\textit{\lambda}-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
    \[(\text{int } n, \text{int } m) \rightarrow n + m\]
  - C++11
    \[
    \text{[](int } x, \text{int } y) \{ \text{return } x + y; \}\]
  - Python
    \[
    (\text{lambda } x, y: x + y)
    \]
  - JavaScript
    \[
    \text{function (a, b) \{ return a + b \}}
    \]
Syntax of Lambda Calculus

\[ e \rightarrow x \quad \text{variables} \]
\[ \quad | \quad \lambda x. e \quad \text{abstraction} \]
\[ \quad | \quad e e \quad \text{application} \]

- **Examples:**

\[
x \quad y \quad z
\]
\[
\lambda x. x \quad \lambda x. y \quad \lambda x. \lambda y. x
\]
\[
x \quad y \quad (\lambda x. x) \quad z \quad x \quad \lambda y. z \quad ((\lambda x. x) \quad \lambda x. x)
\]

- **Conventions when writing \( \lambda \)-expressions:**
  1. Application associates to the left, e.g., \( s \quad t \quad u = (s \quad t) \quad u \)
  2. The body of an abstraction extends as far to the right as possible, e.g.,
     \[ \lambda x. \lambda y. x \quad y \quad x = \lambda x. (\lambda y. ((x \quad y) \quad x)) \]
Bound and Free Variables

- An occurrence of variable $x$ is said to be *bound* when it occurs inside $\lambda x$, 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 $\lambda$-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 $\beta$-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 $[x \mapsto e_1]e_2$:

$\begin{align*}
[x \mapsto e_1]x &= e_1 \\
[x \mapsto e_1]y &= y \\
[x \mapsto e_1](\lambda y.e_2) &= \lambda z.[x \mapsto e_1]([y \mapsto z]e_2) \quad \text{(new } z) \\
[x \mapsto e_1](e_2 e_3) &= ([x \mapsto e_1]e_2 \ [x \mapsto e_1]e_3)
\end{align*}$
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:

\[
\begin{align*}
& id \ (id \ (\lambda z.id \ z)) \\
& id \ (id \ (\lambda z.id \ z)) \\
& id \ (id \ (\lambda z.id \ z))
\end{align*}
\]

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

Reduce the leftmost, outermost redex first:

\[
\begin{align*}
id (id (\lambda z. id z)) &\rightarrow id (\lambda z. id z) \\
&\rightarrow \lambda z. id z \\
&\rightarrow \lambda z. z
\end{align*}
\]

The evaluation is deterministic (i.e., partial function).
Call-by-name strategy

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

\[
\begin{align*}
\text{id} \ (\text{id} \ (\lambda z. \text{id} \ z)) \\
\rightarrow \text{id} \ (\lambda z. \text{id} \ z) \\
\rightarrow \lambda z. \text{id} \ z \\
\n\end{align*}
\]


The call-by-name strategy is \textit{non-strict} (or \textit{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):

\[
\begin{align*}
\text{id} \ (\text{id} \ (\lambda z. \text{id} \ z)) & \\
\rightarrow & \text{id} \ (\lambda z. \text{id} \ z) \\
\rightarrow & \lambda z. \text{id} \ z \\
\not\rightarrow & \\
\end{align*}
\]

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:

\[
E \rightarrow \begin{align*}
& \text{true} \\
& \text{false} \\
& n \\
& x \\
& E + E \\
& \text{iszero } E \\
& \text{if } E \text{ then } E \text{ else } E \\
& \text{let } x = E \text{ in } E \\
& \text{letrec } f(x) = E \text{ in } E \\
& \text{proc } x E \\
& E E
\end{align*}
\]

Define the translation procedure from \( E \) to \( \lambda \)-calculus.
### Compiling to Lambda Calculus

**E**: the translation result of $E$ in $\lambda$-calculus

<table>
<thead>
<tr>
<th>$true$</th>
<th>$\lambda t.\lambda f.t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$false$</td>
<td>$\lambda t.\lambda f.f$</td>
</tr>
<tr>
<td>$0$</td>
<td>$\lambda s.\lambda z.z$</td>
</tr>
<tr>
<td>$1$</td>
<td>$\lambda s.\lambda z.(s\ z)$</td>
</tr>
<tr>
<td>$n$</td>
<td>$\lambda s.\lambda z.(s^n z)$</td>
</tr>
<tr>
<td>$x$</td>
<td>$x$</td>
</tr>
</tbody>
</table>

$$E_1 + E_2 = (\lambda n.\lambda m.\lambda s.\lambda z.m\ s\ (n\ s\ z))\ E_1\ E_2$$

$$\text{iszero } E = (\lambda m.m\ (\lambda x.\text{false})\ true)\ E$$

$$\text{if } E_1 \text{ then } E_2 \text{ else } E_3 = E_1\ E_2\ E_3$$

$$\text{let } x = E_1 \text{ in } E_2 = (\lambda x.E_2)\ E_1$$

$$\text{letrec } f(x) = E_1 \text{ in } E_2 = \text{let } f = Y\ (\lambda f.\lambda x.E_1) \text{ in } E_2$$

$$\text{proc } x\ E = \lambda x.E$$

$$E_1\ E_2 = E_1\ E_2$$
Correctness of Compilation

Theorem

For any expression $E$, 

$$[E] = [E]$$

where $[E]$ denotes the value that results from evaluating $E$. 
Examples: Booleans

\[
\begin{align*}
\text{if } true \text{ then } 0 \text{ else } 1 & \quad = \quad true \ 0 \ 1 \\
& \quad = \quad (\lambda t.\lambda f.t) \ 0 \ 1 \\
& \quad = \quad 0 \\
& \quad = \quad \lambda s.\lambda z.z
\end{align*}
\]

Note that

\[
[\text{if } true \text{ then } 0 \text{ else } 1] = [\text{if } true \text{ then } 0 \text{ else } 1]
\]
Exercises

Define the translation for the boolean operations:

- $E_1$ and $E_2 =$
- $E_1$ or $E_2 =$
- not $E =$
Example: Numerals

\[
\begin{align*}
1 + 2 & = (\lambda n. \lambda m. \lambda s. \lambda z. m \ s \ (n \ s \ z)) \ 1 \ 2 \\
& = \lambda s. \lambda z. 2 \ s \ (1 \ s \ z) \\
& = \lambda s. \lambda z. 2 \ s \ (\lambda s. \lambda z. (s \ z) \ s \ z) \\
& = \lambda s. \lambda z. 2 \ s \ (s \ z) \\
& = \lambda s. \lambda z. (\lambda s. \lambda z. (s \ (s \ z))) \ s \ (s \ z) \\
& = \lambda s. \lambda z. s \ (s \ (s \ z)) \\
& = 3
\end{align*}
\]
Exercises

Define the translation for the boolean operations:

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

- For example, the factorial function

\[ f(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n - 1) \]

is encoded by

\[ \text{fact} = Y(\lambda f.\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times f(n - 1)) \]

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

\[\text{fact } 1 \]

\[= (Y F) 1 \]
\[= (\lambda f. ((\lambda x. f(x x)) (\lambda x. f(x x))) F) 1 \]
\[= ((\lambda x. F(x x)) (\lambda x. F(x x))) 1 \]
\[= (G G) 1 \]
\[= (F (G G)) 1 \]
\[= (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (G G)(n - 1)) 1 \]
\[= \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * (G G)(1 - 1) \]
\[= \text{if } \text{false} \text{ then } 1 \text{ else } 1 * (G G)(1 - 1) \]
\[= 1 * (G G)(1 - 1) \]
\[= 1 * (F (G G))(1 - 1) \]
\[= 1 * (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (G G)(n - 1))(1 - 1) \]
\[= 1 * \text{if } (1 - 1) = 0 \text{ then } 1 \text{ else } (1 - 1) * (G G)((1 - 1) - 1) \]
\[= 1 * 1 \]
Summary

Programming language = Lambda calculus + Syntactic sugars

- \( \lambda \)-calculus is a minimal programming language.
  - Syntax: \( e \rightarrow x \mid \lambda x.e \mid e \ e \)
  - Semantics: \( \beta \)-reduction

- Yet, \( \lambda \)-calculus is Turing-complete.

\[
\begin{align*}
e & \rightarrow x \\
\mid & \lambda x.e \\
\mid & e \ e
\end{align*}
\]