Instructors: Kazutaka Matsuda and Eijiro Sumii

Why We Learn Untyped and Typed λ-Calculus?

- a model of computation,
- the simplest programming language with a type system, and
- a formal proof system for the intuitionistic propositional logic.

Untyped λ-Calculus

Definition (λ-terms)**.** The set of λ-*terms* is defined by the following BNF.

$$M, N ::= x \mid M\ N \mid \lambda x.M$$ □

A λ-term is sometimes called a λ-expression. An expression of the form of $M\ N$ is called *(function) application*, and an expression of the form of $\lambda x.M$ is called λ-*abstraction*.

Example(s). $\lambda x.x$, $\lambda x.y$, $\lambda x.(x\ x)$. $(\lambda x.(x\ x))(\lambda x.(x\ x))$ are examples of λ-terms. □

Intuitively, $\lambda x.M$ represents a function. For example, the function f defined by $f(x) = x + 3$ is represented as $\lambda x.x + 3$, where $+$ and 3 are corresponding λ-terms.

Convention

Function application is left-associative, and binds tighter than abstractions. For example, $M_1\ M_2\ M_2$ means $(M_1\ M_2)\ M_3$, and $\lambda x.x\ x$ means $\lambda x.(x\ x)$. It is simplest to follow the convention that N in $(M\ N)$ must be parenthesized unless N is a variable. A term of the form of $\lambda x_1.\lambda x_2.\dots\lambda x_n.M$ is sometimes written as $\lambda x_1 x_2 \dots x_n.M$.

Occurrence and Subterms

A λ-term may contain multiple occurrences of the same term that have different roles. For example, a term $x\ (\lambda x.x\ (\lambda x.x))$ contains three occurrences of the same variable x, which we want to distinguish as we will show later. A way to formalize occurrences is to use paths in the tree representation of a λ-term.

For a set S, we write by S^* the set of sequences of S elements. That is, an element of S^* is a sequence $s_1 s_2 \dots s_n$ for some n where $s_i \in S$ for all $1 \le i \le n$. The empty sequence is written by ϵ.

Definition. For a λ-term M, the set of *positions* $\mathcal{POS}(M) \subseteq \{1,2\}^*$ is defined inductively as follows.

$$\begin{array}{lcl}
\mathcal{POS}(x) & = & \{\epsilon\} \\
\mathcal{POS}(M\ N) & = & \{\epsilon\} \cup \{1p \mid p \in \mathcal{POS}(M)\} \cup \{2p \mid p \in \mathcal{POS}(N)\} \\
\mathcal{POS}(\lambda x.M) & = & \{\epsilon\} \cup \{1p \mid p \in \mathcal{POS}(M)\}
\end{array}$$ □

Definition. For a λ-term M and a position $p \in \mathcal{POS}(M)$, a *subterm* $M|_p$ at p is defined inductively as follows.

$$\begin{array}{lcl} M|_\epsilon & = & M \\ (M_1\ M_2)|_{ip} & = & M_i|_p \quad (i = 1, 2) \\ (\lambda x.M)|_{1p} & = & M|_p \end{array} \qquad \square$$

We also say that M *occurs at* p *in* N if M is a subterm of N at p. We merely say that M is a subterm of N or M occurs in N if $M = N|_p$ for some p. Note that x does not occur in $\lambda x.y$. Formally, an *occurrence* of a term M in N is a pair (M, p) such that $M = N|_p$, but we do not explicitly use such pairs in what follows.

Free and Bound Variables

Definition. For a λ-term M, a variable x that occurs at p in M is called *bound* if there is a subterm $\lambda x.M'$ in M at p' and $p = p'p''$ for some p'' (i.e., $\lambda x.M'$ contains the occurrence of x). Otherwise, the variable occurrence is called *free*. $\square$

Example(s). The λ-term $(\lambda x.x)\ x$ has two occurrences of x: the left one at 11 is bound and the other at 2 is free. $\square$

Exercise. Underline the bound occurrences of variables in $(\lambda x.x\ (\lambda y.x\ y)\ y)\ (\lambda z.z)\ z$.

Definition. For a λ-term M, the set of *free variables* $\mathrm{FV}(M)$ of M is the set of variables that occur free in M. The set can be defined inductively as follows.

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\} \\ \mathrm{FV}(\lambda x.M) & = & \mathrm{FV}(M) \setminus \{x\} \\ \mathrm{FV}(M\ N) & = & \mathrm{FV}(M) \cup \mathrm{FV}(N) \end{array} \qquad \square$$

A term M is called *closed* if M has no free variables, i.e., $\mathrm{FV}(M) = \emptyset$. Closed λ-terms are sometimes called combinators. Famous combinators include $I \stackrel{\mathrm{def}}{=} \lambda x.x$, $K \stackrel{\mathrm{def}}{=} \lambda x.\lambda y.x$, $S \stackrel{\mathrm{def}}{=} \lambda x.\lambda y.\lambda z.x\ z\ (y\ z)$, $\Delta \stackrel{\mathrm{def}}{=} \lambda x.x\ x$, and Y that will be introduced later.

Substitution and α-equivalence

Intuitively, a substitution $M[N/x]$ replaces all the free occurrences of x in M with N. However, naively doing so is problematic when N contains free variables. Let us consider two λ-terms $\lambda x.z\ x$ and $\lambda y.z\ y$. We do not want to distinguish two terms as $f(x) = z + x$ and $f(y) = z + y$ represent the same function. However, naively replacing z with y makes the two function different. Thus, we define substitution so that it renames bound variables if necessary, as follows.

Definition. For a variable x and λ-terms M and N, we define a *(capture-avoiding) substitution* of x in M to N, $M[N/x]$, inductively as follows.

$$\begin{array}{lcl} y[N/x] & = & \begin{cases} N & (x = y) \\ y & (x \neq y) \end{cases} \\ (\lambda y.M)[N/x] & = & \begin{cases} \lambda y.M & (x = y) \\ \lambda y.M[N/x] & (x \neq y \wedge y \notin \mathrm{FV}(N)) \\ (\lambda z.M[z/y])[N/x] & (x \neq y \wedge y \in \mathrm{FV}(N) \wedge z \notin \mathrm{FV}(N)) \end{cases} \\ (M\ M')[N/x] & = & (M[N/x])\ (M'[N/x]) \end{array} \qquad \square$$

Note. There is another common way to write substitution: $M[x := N]$ to mean $M[N/x]$. Some people use a prefix notation to write $[x := N]M$ instead. Some people represent a (simultaneous) substitution itself as a function θ from variables to terms such that $\{x \mid \theta(x) \neq x\}$ is finite, and then define its application $M\theta$ to a term M. □

The notion of α-equivalence formalizes the equality of terms up to remaining of bound variables.

Definition (α-equivalence)**.** the relation $\equiv_\alpha$ is the smallest reflexive and transitive relation satisfying the following conditions.

- $\lambda x.M \equiv_\alpha \lambda y.M[y/x]$ for all λ-terms M, variables x, and variables $y \notin \mathrm{FV}(M)$.
- $M \equiv_\alpha M'$ implies $\lambda x.M \equiv_\alpha \lambda x.M'$ for all λ-terms M and M'.
- $M \equiv_\alpha M'$ implies $M\ N \equiv_\alpha M'\ N$ for all λ-terms M, M' and N.
- $N \equiv_\alpha N'$ implies $M\ N \equiv_\alpha M\ N'$ for all λ-terms M, N and N'. □

Example(s). The pairs $\lambda x.x$ and $\lambda y.y$, $\lambda x.z\ x$ and $\lambda y.z\ y$, and $(\lambda x.x\ x)\ (\lambda x.x\ x)$ and $(\lambda y.y\ y)\ (\lambda z.z\ z)$ are all α-equivalent terms. In contrast, $\lambda x.z\ x$ and $\lambda x.w\ x$ are not α-equivalent. □

Replacement of a λ-term with an α-equivalent one is called *α-conversion* or *α-renaming*.

Convention

We identify two α-equivalent λ-terms. In other words, $\lambda x.x$ and $\lambda y.y$ are treated as the same term. In this sense, the third clause of the definition $(\lambda y.M)[N/x]$ is superfluous because we can choose the name of bound variables so that the conditions in the second clause are fulfilled.

β-Reduction

Now we are ready to define the all and only computing mechanism of λ-terms, β-reduction.

Definition (β-reduction)**.** We define the relation $\longrightarrow_\beta$ by the following rules.

$$\frac{}{(\lambda x.M)N \longrightarrow_\beta M[N/x]} \qquad \frac{M \longrightarrow_\beta M'}{M\ N \longrightarrow_\beta M'\ N} \qquad \frac{N \longrightarrow_\beta N'}{M\ N \longrightarrow_\beta M\ N'} \qquad \frac{M \longrightarrow_\beta M'}{\lambda x.M \longrightarrow_\beta \lambda x.M'}$$ □

We sometimes omit β to write $\longrightarrow$. Intuitively, β-reduction replaces an occurrence of $(\lambda x.M)\ N$ with $M[N/x]$. A term M is in a (β-) *normal form* if there is no N such that $M \longrightarrow_\beta N$. We say M is a normal form of N if M is in a normal form and $N \longrightarrow_\beta^* M$. Some λ-terms do not have normal forms, such as $(\lambda x.x\ x)\ (\lambda x.x\ x)$. A subterm of the form of $(\lambda x.M)\ N$ is sometimes called (β-) *redex*. A term can contain multiple redexes as $(\lambda x.(\lambda y.y)\ x)\ ((\lambda z.z)\ (\lambda w.w))$; in such a situation, the result of a β-reduction depends on the choice of the redex. It is known that those terms will coincide after further β-reductions if we choose redexes appropriately. This property is called Church-Rosser property.

Theorem (Church-Rosser)**.** Let $\equiv_\beta$ be the smallest reflexive, symmetric and transitive relation that contains $\longrightarrow_\beta$. Then, for all λ-terms M and M' such that $M \equiv_\beta M'$, there exists a term N such that $M \longrightarrow_\beta^* N$ and $M' \longrightarrow_\beta^* N$. □

It follows that, if a term has a normal form, the normal form is unique. Even if a term has a normal form, not all sequence of reduction lead to it (some may never terminate), as $(\lambda x.y)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$. It is known that, if we reduce the leftmost outermost redex, the reduction sequence always ends in the normal form if it exists.

We may consider another reduction called η.

$$\frac{x \notin \mathrm{FV}(M)}{\lambda x.Mx \longrightarrow_\eta M} \qquad \frac{M \longrightarrow_\eta M'}{M\ N \longrightarrow_\eta M'\ N} \qquad \frac{N \longrightarrow_\eta N'}{M\ N \longrightarrow_\eta M\ N'} \qquad \frac{M \longrightarrow_\eta M'}{\lambda x.M \longrightarrow_\eta \lambda x.M'}$$

Church Encoding

We now introduce how to represent computations in λ-calculus.

Church Booleans. First, we represent computation with Boolean values in λ-calculus. We represent a thing by what it can do. For Booleans, what they can do is branching, so we define *true* and *false* as follows.

$$\begin{array}{rcl} \mathit{true} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.x \\ \mathit{false} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.y \end{array}$$

Branching then is merely an application.

$$\textbf{if } M_1 \textbf{ then } M_2 \textbf{ else } M_3 \stackrel{\text{def}}{=} M_1\ M_2\ M_3$$

It is easy to see that $\mathit{true}\ M\ N \longrightarrow^* M$ and $\mathit{false}\ M\ N \longrightarrow^* N$.

Boolean functions can be defined on the representation. For example, the negation operator *not* can be defined as:

$$\mathit{not} \stackrel{\text{def}}{=} \lambda b.\lambda x.\lambda y.b\ y\ x.$$

Check how terms $(\mathit{not}\ \mathit{true})\ M\ N$ and $(\mathit{not}\ \mathit{false})\ M\ N$ will be reduced. As another example, we define the function *and* that does conjunction:

$$\mathit{and} \stackrel{\text{def}}{=} \lambda b_1.\lambda b_2.b_1\ b_2\ \mathit{false}.$$

The subterm $b_1\ b_2\ \mathit{false}$ essentially represents $\textbf{if } b_1 \textbf{ then } b_2 \textbf{ else } \mathit{false}$. Check how terms $(\mathit{and}\ \mathit{true}\ b)\ M\ N$ and $(\mathit{and}\ \mathit{false}\ b)\ M\ N$ will be reduced.

<u>Exercise</u>. Define a λ-term "*or*" that corresponds to disjunction. □

Church Pairs. We now define the representation of pairs. Since a pair encapsulates two pieces of data, what a pair can do is to pass the data to the rest of computation. Thus, if we write *pair* for the pair constructor, we can define it as follows.

$$\mathit{pair} \stackrel{\text{def}}{=} \lambda x.\lambda y.\lambda f.f\ x\ y$$

We extract the first and the second components of a pair by the following functions *fst* and *snd*, respectively.

$$\begin{array}{rcl} \mathit{fst} & \stackrel{\text{def}}{=} & \lambda p.p\ \mathit{true} \\ \mathit{snd} & \stackrel{\text{def}}{=} & \lambda p.p\ \mathit{false} \end{array}$$

Check how $\mathit{fst}\ (\mathit{pair}\ M\ N)$ will be reduced.

Church Numerals. Now, we discuss how to perform computations on natural numbers. In Church encoding, a natural number n is represented by the nth iteration.

$$
\begin{array}{lcl}
0 & \stackrel{\text{def}}{=} & \lambda s.\lambda z.z \\
1 & \stackrel{\text{def}}{=} & \lambda s.\lambda z.s\ z \\
2 & \stackrel{\text{def}}{=} & \lambda s.\lambda z.s\ (s\ z)
\end{array}
\qquad
\begin{array}{lcl}
3 & \stackrel{\text{def}}{=} & \lambda s.\lambda z.s\ (s\ (s\ z)) \\
 & \vdots & \\
n & = & \lambda s.\lambda z.\underbrace{s\ (\ldots (s}_{n}\ z) \ldots)
\end{array}
$$

In other words, the encoding of a natural number n represents the same computation as the following JavsScript-like code.

```
var r = z;
for (var i = 0; i < n; i++) {
   r = s(r);
}
```

In advance to defining the addition of Church numerals, we define the function *succ* to compute the successor.

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

Addition *add* is then as follows.

$$\mathit{add} \stackrel{\text{def}}{=} \lambda n.\lambda m.n\ \mathit{succ}\ m$$

For example, *add* 1 1 is reduced as follows.

$$
\begin{array}{l}
\mathit{add}\ 1\ 1 = (\lambda n.\lambda m.n\ \mathit{succ}\ m)\ (\lambda s.\lambda z.s\ z)\ (\lambda s.\lambda z.s\ z) \\
\longrightarrow (\lambda m.(\lambda s.\lambda z.s\ z)\ \mathit{succ}\ m)\ (\lambda s.\lambda z.s\ z) \\
\longrightarrow (\lambda s.\lambda z.s\ z)\ \mathit{succ}\ (\lambda s.\lambda z.s\ z) \\
\longrightarrow (\lambda z.\mathit{succ}\ z)\ (\lambda s.\lambda z.s\ z) \\
\longrightarrow \mathit{succ}\ (\lambda s.\lambda z.s\ z) = (\lambda n.\lambda s.\lambda z.s\ (n\ s\ z))\ (\lambda s.\lambda z.s\ z) \\
\longrightarrow \lambda s'.\lambda z'.s'\ ((\lambda s.\lambda z.s\ z)\ s'\ z') \\
\longrightarrow \lambda s'.\lambda z'.s'\ ((\lambda z.s'\ z)\ z') \longrightarrow \lambda s'.\lambda z'.s'\ (s'\ z') = 2
\end{array}
$$

Exercise. Another definition of *succ* is

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

How *add* 1 1 will be reduced with this definition of *succ*? Also, one can define *add* without using *succ* as follows.

$$\mathit{add} \stackrel{\text{def}}{=} \lambda n.\lambda m.\lambda s.\lambda z.n\ s\ (m\ s\ z)$$

Compute *add* 1 1 with this definition. □

Exercise. Give λ-terms *mult* and *pow* that compute multiplication and exponentiation. □

We need a small trick to define a predecessor function.

$$\mathit{pred} \stackrel{\text{def}}{=} \lambda n.\mathit{fst}\ (n\ (\lambda p.\mathit{pair}\ (\mathit{snd}\ p)\ (\mathit{succ}\ (\mathit{snd}\ p))\ (\mathit{pair}\ 0\ 0))$$

The trick is to keep the result of the previous iteration by using a pair. Notice that *pred* 0 evaluates to 0 in this definition.

By using *pred*, we can define subtraction.

$$\mathit{sub} \stackrel{\text{def}}{=} \lambda n.\lambda m.m\ \mathit{pred}\ n$$

Notice that $\mathit{sub}\ n\ m$ evaluates to 0 if $n \leq m$.

It is sometimes useful to check whether a number is 0 or not.

$$\mathit{isZero} \stackrel{\text{def}}{=} \lambda n\ (\lambda x.\mathit{false})\ \mathit{true}$$

Exercise. Give λ-terms *le*, *lt*, *ge*, *gt* and *eq* that correspond to $(\leq)$, $(<)$, $(\geq)$, $(>)$ and $(=)$ on natural numbers, respectively. $\square$

General Recursion

Assume that we have a λ-term Y that can be reduced as follows.

$$Y\ M \longrightarrow^* M\ (Y\ M)$$

With Y, we can realize recursive functions:

$$\mathit{sum} \stackrel{\text{def}}{=} Y\ (\lambda f.\lambda n.\textbf{if}\ \mathit{isZero}\ n\ \textbf{then}\ 0\ \textbf{else}\ \mathit{add}\ n\ (f\ (\mathit{pred}\ n))).$$

For example, *sum* 2 evaluates as follows.

$$\begin{aligned}
\mathit{sum}\ 2 &\longrightarrow^* \textbf{if}\ \mathit{isZero}\ 2\ \textbf{then}\ 0\ \textbf{else}\ \mathit{add}\ 2\ (\mathit{sum}\ (\mathit{pred}\ 2)) \\
&\longrightarrow^* \mathit{add}\ 2\ (\mathit{sum}\ 1) \\
&\longrightarrow^* \mathit{add}\ 2\ (\textbf{if}\ \mathit{isZero}\ 1\ \textbf{then}\ 0\ \textbf{else}\ \mathit{add}\ 1\ (\mathit{sum}\ (\mathit{pred}\ 1))) \\
&\longrightarrow^* \mathit{add}\ 2\ (\mathit{add}\ 1\ (\mathit{sum}\ 0)) \\
&\longrightarrow^* \mathit{add}\ 2\ (\mathit{add}\ 1\ (\textbf{if}\ \mathit{isZero}\ 0\ \textbf{then}\ 0\ \textbf{else}\ \mathit{add}\ 0\ (\mathit{sum}\ (\mathit{pred}\ 0)))) \\
&\longrightarrow^* \mathit{add}\ 2\ (\mathit{add}\ 1\ 0) \longrightarrow^* 3
\end{aligned}$$

How do we define such Y? A hint is the λ-term $\Delta = \lambda x.x\ x$; we have $\Delta\ (\lambda x.\Delta\ x) \longrightarrow (\lambda x.\Delta\ x)\ (\lambda x.\Delta\ x) \longrightarrow \Delta\ (\lambda x.\Delta\ x)$. Then, consider a slightly different version $\Delta\ (\lambda x.f\ (\Delta\ x)))$ that produces f after copying by the first Δ. Then, we have $\Delta\ (\lambda x.f\ (\Delta\ x))) \longrightarrow (\lambda x.f\ (\Delta\ x))\ (\lambda x.f\ (\Delta\ x)) \longrightarrow f\ (\Delta\ (\lambda x.f\ (\Delta\ x)))$. Thus, we can define Y as follows.

$$Y \stackrel{\text{def}}{=} \lambda f.\Delta\ (\lambda x.f\ (\Delta\ x))$$

This Y is known as Curry's fixed-point combinator.

Exercise. Give a λ-term that computes factorials with or without Y. Give a λ-term that computes the Ackermann function a defined below with Y.

$$a(m, n) = \begin{cases} n + 1 & \text{if } m = 0, \\ a(m-1, 1) & \text{if } n = 0, \\ a(m-1, a(m, n-1)) & \text{otherwise.} \end{cases}$$ $\square$