FLOLAC 2026

λ-Calculus

Untyped λ-Calculus, Part II

陳亮廷 Chen, Liang-Ting

Formosan Summer School on Logic, Language, and Computation 2026

Institute of Information Science
Academia Sinica

Recap

We have covered so far

- Terms: variables, abstractions, and applications.
- α-equivalence avoids dependence on bound-variable names.
- β-reduction: $(\lambda x.\, t)\ u \longrightarrow_\beta t[u/x]$.

Today, we will use λ-calculus as a programming language, then study two aspects of reduction: evaluation strategies and confluence.

Defining data in λ-Calculus

Church encoding: data by behaviour

A Church-encoded value is represented by what it does to its eliminators.

Boolean	chooses between two branches
Natural number	iterates a function a fixed number of times

Boolean values and conditional expressions can be encoded as combinators.

Boolean values

$$\texttt{True} := \lambda x\, y.\, x$$
$$\texttt{False} := \lambda x\, y.\, y$$

Conditional

$$\texttt{if} := \lambda b\ x\ y.\ b\, x\, y$$
$$\texttt{if True}\ t\ u \twoheadrightarrow_\beta t$$
$$\texttt{if False}\ t\ u \twoheadrightarrow_\beta u$$

for any two λ-terms t and u.

Thus a Boolean value is represented by its eliminator: it receives the ‘then’ branch and the ‘else’ branch, and selects one of them.

Example

$$\begin{aligned}\texttt{if True}\ t\ u &= (\lambda b\, x\, y.\, b\, x\, y)\ (\lambda x\, y.\, x)\ t\ u \\ &\longrightarrow_\beta (\lambda x\, y.\, x)\ t\ u \\ &\longrightarrow\!\!\!\!\!\rightarrow_\beta t\end{aligned}$$

Exercise

Define the following terms directly without using `if`. Hint: a Boolean value is already a conditional.

- Negation: `not` such that `not True` $\longrightarrow\!\!\!\!\!\rightarrow_\beta$ `False` and `not False` $\longrightarrow\!\!\!\!\!\rightarrow_\beta$ `True`.
- Conjunction `and`.
- Disjunction `or`.

Natural numbers and arithmetic operations can also be encoded.

Church numerals

$$\begin{aligned}\mathbf{c}_0 &:= \lambda f\, x.\, x \\ \mathbf{c}_1 &:= \lambda f\, x.\, f\, x \\ \mathbf{c}_2 &:= \lambda f\, x.\, f\, (f\, x) \\ \mathbf{c}_{n+1} &:= \lambda f\, x.\, f^{n+1}(x)\end{aligned}$$

where $f^1(x) := f\ x$ and $f^{n+1}(x) := f\ (f^n(x))$.

A Church numeral receives a function f and an initial value x. The numeral $\mathbf{c}_n$ applies f to x exactly n times.

Successor

$$\mathsf{succ} := \lambda n.\, \lambda f\, x.\, f\, (n\, f\, x)$$

$$\mathsf{succ}\ \mathbf{c}_n \longrightarrow\!\!\!\!\!\rightarrow_\beta \mathbf{c}_{n+1}$$

Addition

$$\mathsf{add} := \lambda n\, m.\, \lambda f\, x.\ n\ f\ (m\ f\ x)$$

$$\mathsf{add}\ \mathbf{c}_n\ \mathbf{c}_m \longrightarrow\!\!\!\!\!\rightarrow_\beta \mathbf{c}_{n+m}$$

Zero test

$$\mathtt{ifz} := \lambda n\ x\ y.\ n\ (\lambda z.\ y)\ x$$

$$\mathtt{ifz}\ \mathbf{c}_0\ t\ u \longrightarrow\!\!\!\!\!\rightarrow_\beta t$$

$$\mathtt{ifz}\ \mathbf{c}_{n+1}\ t\ u \longrightarrow\!\!\!\!\!\rightarrow_\beta u$$

Example

$$
\begin{aligned}
\mathsf{succ}\ \mathbf{c}_1 &= (\lambda n.\, \lambda f\, x.\, f\, (n\, f\, x))\ \mathbf{c}_1 \\
&\longrightarrow_\beta \lambda f\, x.\, f\, (\mathbf{c}_1\, f\, x) \\
&\longrightarrow\!\!\!\!\rightarrow_\beta \lambda f\, x.\, f\, (f\, x) \\
&= \mathbf{c}_2.
\end{aligned}
$$

Exercise

1. Evaluate `succ` $\mathbf{c}_0$ and `add` $\mathbf{c}_1$ $\mathbf{c}_2$.
2. Define the multiplication `mul` over Church numerals.
3. (Hard) Define the predecessor `pred`, i.e. `pred` $\mathbf{c}_{1+n} =_\beta \mathbf{c}_n$.
   Hint: use pairs to keep track of both the current and previous numeral.

Recursion in λ-Calculus

The summation $\sum_{i=0}^{n} i$, for $n \in \mathbb{N}$, is typically described by self-reference as

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

In the pure λ-calculus, this cannot be expressed by directly naming sum inside its own definition. That is, the equation

$$sum = \lambda n.\, \mathtt{ifz}\ n\ \mathbf{c}_0\ (\mathtt{add}\ n\ (sum\ (\mathtt{pred}\ n)))$$

is a recursive equation at the meta-level, not a λ-term.[1]

A term cannot refer to its own name, but self-reference can still be encoded.

[1] Here $=$ denotes meta-level equality, whereas $=_\beta$ denotes object-level equality of λ-terms.

Instead of referring to sum by name, define a functional G. The argument f of $G\ f$ is the recursive call that the body is allowed to use:

$$G := \lambda f.\, \lambda n.\, \mathtt{ifz}\ n\ \mathbf{c}_0\ (\mathtt{add}\ n\ (f\ (\mathtt{pred}\ n))).$$

Thus $G\ f$ computes one step of summation, assuming that f computes the smaller recursive calls. A recursive summation function should then satisfy

$$\mathtt{sum} =_\beta G\ \mathtt{sum}.$$

If we have a fixed-point combinator $\mathbf{Y}$ producing a fixed point of G, then

$$\mathtt{sum} := \mathbf{Y}G, \qquad \mathbf{Y}G =_\beta G\ (\mathbf{Y}G).$$

Therefore, in the body of G, the variable f is instantiated by $\mathbf{Y}G$ itself:

$$\mathbf{Y}G =_\beta \lambda n.\, \mathtt{ifz}\ n\ \mathbf{c}_0\ (\mathtt{add}\ n\ ((\mathbf{Y}G)\ (\mathtt{pred}\ n))).$$

Proposition 1

The Y combinator defined as

$$\mathbf{Y} := \lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)).$$

is a fixed-point operator; that is, for every λ-term F, $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$.

Intuitively, $\mathbf{Y}F$ defines recursion, while F describes *one-step* unfolding of the recursive definition.

Exercise

Verify that $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$ for any term F.

Using the functional G from the previous slide, define

$$\mathtt{sum} := \mathbf{Y}G.$$

Also let

$$G' := (\lambda x.\, G\ (x\ x))\ (\lambda x.\, G\ (x\ x)).$$

Then $\mathbf{Y}G \longrightarrow_\beta G'$, and hence

$$\begin{aligned}
\mathtt{sum}\ \mathbf{c}_1 &\longrightarrow_\beta G'\ \mathbf{c}_1 \\
&\longrightarrow_\beta (G\ G')\ \mathbf{c}_1 \\
&\longrightarrow_\beta (\lambda n.\, \mathtt{ifz}\ n\ \mathbf{c}_0\ (\mathtt{add}\ n\ (G'\ (\mathtt{pred}\ n))))\ \mathbf{c}_1 \\
&\longrightarrow_\beta \mathtt{ifz}\ \mathbf{c}_1\ \mathbf{c}_0\ (\mathtt{add}\ \mathbf{c}_1\ (G'\ (\mathtt{pred}\ \mathbf{c}_1))) \\
&\longrightarrow_\beta \cdots
\end{aligned}$$

Exercise: Identifying the Recursive Call

Consider

$$G := \lambda f\, n.\, \texttt{ifz}\ n\ \mathbf{c}_0\ (\texttt{add}\ n\ (f\ (\texttt{pred}\ n))).$$

1. What role does the variable f play in G?
2. Write out $G\ (\mathbf{Y}G)$.
3. Explain why $\mathbf{Y}G$ behaves like the summation function.

Exercise

Reduce

$$\texttt{sum}\ \mathbf{c}_2$$

far enough to see the recursive calls

$$\texttt{add}\ \mathbf{c}_2\ (\texttt{sum}\ \mathbf{c}_1), \qquad \texttt{add}\ \mathbf{c}_1\ (\texttt{sum}\ \mathbf{c}_0).$$

You do not need to expand the definitions of `add` or `pred`.

Curry's combinator satisfies

$$\mathbf{Y}F =_\beta F(\mathbf{Y}F),$$

but the term $\mathbf{Y}F$ does not reduce literally to $F(\mathbf{Y}F)$.

Turing's fixed-point combinator has the stronger reduction behaviour:

$$\Theta F \longrightarrow\!\!\!\!\!\rightarrow_\beta F(\Theta F).$$

Proposition 2

Define

$$\Theta := (\lambda x\, f.\, f\,(x\,x\,f))\;(\lambda x\, f.\, f\,(x\,x\,f)).$$

Then $\Theta F \longrightarrow\!\!\!\!\!\rightarrow_\beta F(\Theta F)$.

Exercise

Show that $\Theta F \longrightarrow\!\!\!\!\!\rightarrow_\beta F(\Theta F)$.

Exercise: factorial

Define factorial using a fixed-point combinator, for example

$$\texttt{fact} := \mathbf{Y} G_{fact} \quad \text{or} \quad \texttt{fact} := \Theta G_{fact}.$$

That is, find a functional G_{fact} such that

$$\texttt{fact}\ \mathbf{c}_0 \twoheadrightarrow_\beta \mathbf{c}_1, \qquad \texttt{fact}\ \mathbf{c}_{n+1} \twoheadrightarrow_\beta \texttt{mul}\ \mathbf{c}_{n+1}\ (\texttt{fact}\ \mathbf{c}_n).$$

Assume that `ifz`, `pred`, and `mul` have already been defined.

Exercise: Different Reduction Choices

Let G be the summation functional defined above, and let

$$T := (\Theta G)\ ((\lambda x.\, x)\ \mathbf{c}_1).$$

1. Identify two different β-redexes in T.
2. Reduce the argument first:

$$(\Theta G)\ ((\lambda x.\, x)\ \mathbf{c}_1) \longrightarrow_\beta (\Theta G)\ \mathbf{c}_1.$$

3. Unfold the fixed point first:

$$(\Theta G)\ ((\lambda x.\, x)\ \mathbf{c}_1) \twoheadrightarrow_\beta G(\Theta G)\ ((\lambda x.\, x)\ \mathbf{c}_1).$$

4. Continue both paths far enough to see that they are computing the same recursive call.

Exercise: divergence from fixed points

Let

$$G := \lambda f.\, f.$$

Study the behaviour of

$$\mathbf{Y}G.$$

1. Show that $\mathbf{Y}G =_\beta G(\mathbf{Y}G)$.
2. Reduce $G(\mathbf{Y}G)$.
3. Does this term terminate?

Let

$$G_0 := \lambda f\, n.\, \mathbf{c}_0$$

and define

$$\texttt{zero} := \mathbf{Y} G_0.$$

Intuitively, `zero` should ignore its argument and return $\mathbf{c}_0$.

1. Show that, by unfolding the fixed point once,

$$\texttt{zero}\ \mathbf{c}_3 =_\beta G_0(\mathbf{Y}G_0)\ \mathbf{c}_3 \longrightarrow\!\!\!\!\!\rightarrow_\beta \mathbf{c}_0.$$

2. Now expand $\mathbf{Y}G_0$:

$$\mathbf{Y}G_0 \longrightarrow_\beta (\lambda x.\, G_0(x\,x))\ (\lambda x.\, G_0(x\,x)).$$

   What happens under a call-by-value-like strategy that tries to evaluate the argument of G_0 before reducing the redex $G_0(\cdots)$?

Fixed-point combinators make recursion possible in the pure λ-calculus, but they also expose two fundamental issues with unrestricted reduction.

Non-determinacy A term may contain several β-redexes. Reducing different redexes first may produce different intermediate terms:

$$t \twoheadrightarrow_\beta t_1, \qquad t \twoheadrightarrow_\beta t_2.$$

Divergence Some reduction choices may unfold a recursive call forever, even when another choice would quickly reach a result.

The next two topics address these issues.

Evaluation strategies How should a programming language choose which redex to reduce?

Confluence If two reduction paths start from the same term, can they be joined again?

Evaluation strategy

An evaluation strategy is a procedure for selecting which β-redex to reduce at each step of computation.

Formally, an evaluation strategy is a subrelation $\longrightarrow_{\text{ev}}$ of the full one-step β-reduction relation $\longrightarrow_{\beta}$.

Innermost β-redex A β-redex that contains no other β-redex.

Outermost β-redex A β-redex that is not contained in any other β-redex.

Leftmost-outermost strategy (normal order) **Reduce the leftmost outermost β-redex first. For example,**

$$
\begin{aligned}
& \underline{(\lambda x.\,(\lambda y.\,y)\,x)} \quad \underline{(\lambda x.\,(\lambda y.\,y\,y)\,x)} \\
\longrightarrow_\beta\ & \underline{(\lambda y.\,y)} \quad \underline{(\lambda x.\,(\lambda y.\,y\,y)\,x)} \\
\longrightarrow_\beta\ & \lambda x.\,\underline{(\lambda y.\,y\,y)} \quad \underline{x} \\
\longrightarrow_\beta\ & (\lambda x.\,x\,x) \\
& \not\longrightarrow_\beta
\end{aligned}
$$

Leftmost-innermost strategy **Reduce the leftmost innermost β-redex first. For example,**

$$
\begin{aligned}
& (\lambda x.\, \underline{(\lambda y.\, y)}\ \underline{x})\ (\lambda x.\, (\lambda y.\, y\, y)\, x) \\
\longrightarrow_\beta\ & (\lambda x.\, x)\ (\lambda x.\, \underline{(\lambda y.\, y\, y)}\ \underline{x}) \\
\longrightarrow_\beta\ & \underline{(\lambda x.\, x)}\ \underline{(\lambda x.\, x\, x)} \\
\longrightarrow_\beta\ & (\lambda x.\, x\, x) \\
\not\longrightarrow_\beta\ &
\end{aligned}
$$

Rightmost-innermost and rightmost-outermost strategies **These are defined similarly, but choose the rightmost candidate redex instead.**

For terms with free variables, take values to be variables and abstractions:

$$V ::= x \mid \lambda x.\, t.$$

Call-by-value Reduce a redex $(\lambda x.\, t)\, V$ only when the argument is already a value V; do not reduce under abstractions.

Call-by-name Reduce the leftmost outermost redex without first evaluating its argument; do not reduce under abstractions.

Proposition 3 (Determinacy)

Each fixed evaluation strategy is deterministic. That is, if $t \longrightarrow_{\text{ev}} u_1$ *and* $t \longrightarrow_{\text{ev}} u_2$, *then* $u_1 = u_2$.

Exercise

Let

$$\Omega := (\lambda x.\, x\, x)\, (\lambda x.\, x\, x), \qquad \mathbf{K}_1 := \lambda x\, y.\, x.$$

Compare the behaviour of

$$\mathbf{K}_1\ z\ \Omega$$

under call-by-name and call-by-value.

Definition 4

1. M is in *normal form* if $M \not\longrightarrow_\beta N$ for every term N.
2. M is *weakly normalising* if $M \longrightarrow\!\!\!\!\!\rightarrow_\beta N$ for some term N in normal form.

$$\Omega := (\lambda x.\, x\, x)\, (\lambda x.\, x\, x), \qquad \mathbf{K}_1 := \lambda x\, y.\, x.$$

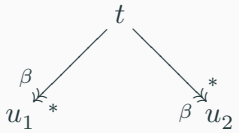
1. Ω is not weakly normalising.
2. $\mathbf{K}_1$ is normal, and hence weakly normalising.
3. $\mathbf{K}_1\ z\ \Omega$ is weakly normalising.

Theorem 5

The normal-order strategy reduces every weakly normalising term to a normal form.

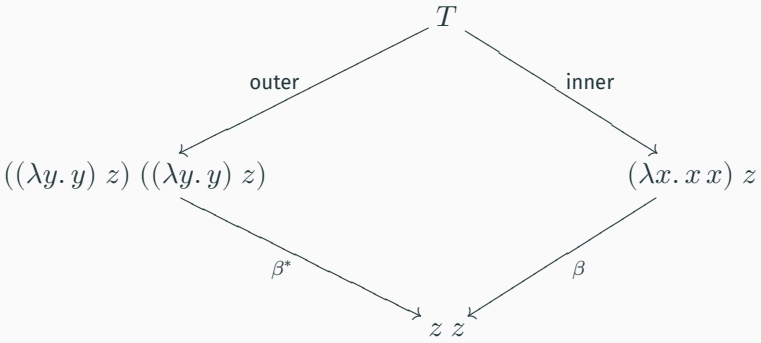
Confluence

Multi-step β-reduction is nondeterministic: a term may contain several β-redexes, and reducing different redexes first may produce different intermediate terms.



Confluence does not say that every strategy terminates. It says that different successful reduction paths are compatible: if two reducts are reached, they can be joined by further reductions.

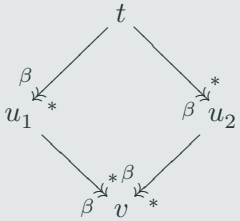
Let $T := (\lambda x.\, x\, x)\ ((\lambda y.\, y)\ z)$. There are two different first reduction steps:



The first two reducts are different, but confluence says that further reduction can join them.

Theorem 6 (Confluence)

If $t \longrightarrow\!\!\!\!\!\rightarrow_\beta u_1$ and $t \longrightarrow\!\!\!\!\!\rightarrow_\beta u_2$, then there exists a term v such that $u_1 \longrightarrow\!\!\!\!\!\rightarrow_\beta v$ and $u_2 \longrightarrow\!\!\!\!\!\rightarrow_\beta v$.



The term v is called a common reduct of u_1 and u_2.

Confluence lets us reason about computational equality by reducing both sides to a common reduct. Its proof is a bit involved, so we ignore it for now.

Proposition 7

If $t =_\beta u$, then there exists a term v such that $t \longrightarrow\!\!\!\!\!\rightarrow_\beta v$ and $u \longrightarrow\!\!\!\!\!\rightarrow_\beta v$.

Idea.

The relation $=_\beta$ is generated by $\longrightarrow_\beta$, reflexivity, symmetry, and transitivity. Confluence handles the cases where reduction paths go in different directions. □

Corollary 8

If t and u are both normal forms and $t =_\beta u$, then $t =_\alpha u$.

Exercise

Show that if $t =_\beta u$, then there is a common reduct v of t and u; that is, $t \longrightarrow\!\!\!\!\!\rightarrow_\beta v$ and $u \longrightarrow\!\!\!\!\!\rightarrow_\beta v$ by induction on the derivation of $t =_\beta u$.

Proof of Confluence

Proving confluence can be tricky. This section presents a direct strategy based on the notion of complete development, which contracts as many β-redexes as possible *statically*.

The *complete development* M^* of a λ-term M is defined by

$$
\begin{aligned}
x^* &= x \\
(\lambda x.\, M)^* &= \lambda x.\, M^* \\
((\lambda x.\, M)\ N)^* &= M^*[N^*/x] \\
(M\ N)^* &= M^*\ N^* && \text{if } M \not\equiv \lambda x.\, M'.
\end{aligned}
$$

Let $M \Longrightarrow_\beta N$ denote the *parallel reduction* relation defined by the following rules.

$$\frac{}{x \Longrightarrow_\beta x} \qquad \frac{M \Longrightarrow_\beta M' \qquad N \Longrightarrow_\beta N'}{M\ N \Longrightarrow_\beta M'\ N'}$$

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

For example,

$$\underline{(\lambda x.\, (\lambda y.\, y)\ x)}\ \underline{((\lambda x.\, x)\ \texttt{false})} \Longrightarrow_\beta \texttt{false},$$

because $(\lambda y.\, y)\ x \Longrightarrow_\beta x$ and $(\lambda x.\, x)\ \texttt{false} \Longrightarrow_\beta \texttt{false}$.

Lemma 9

1. $M \Longrightarrow_\beta M$ *holds for every term* M,
2. $M \longrightarrow_\beta N$ *implies* $M \Longrightarrow_\beta N$, *and*
3. $M \Longrightarrow_\beta N$ *implies* $M \longrightarrow\!\!\!\!\!\rightarrow_\beta N$.

In particular, $M \Longrightarrow_\beta^* N$ *if and only if* $M \longrightarrow\!\!\!\!\!\rightarrow_\beta N$.

Lemma 10 (Substitution respects parallel reduction)

If $M \Longrightarrow_\beta M'$ *and* $N \Longrightarrow_\beta N'$, *then* $M[N/x] \Longrightarrow_\beta M'[N'/x]$.

Theorem 11 (Triangle property)

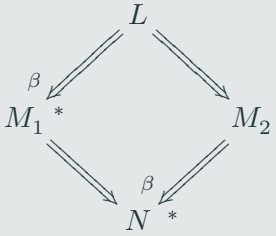
If $M \Longrightarrow_\beta N$, *then* $N \Longrightarrow_\beta M^*$.

Proof sketch.

By induction on the derivation of $M \Longrightarrow_\beta N$. □

Lemma 12 (Strip lemma)

If $L \Longrightarrow_\beta^* M_1$ and $L \Longrightarrow_\beta M_2$, then there exists a term N such that $M_1 \Longrightarrow_\beta N$ and $M_2 \Longrightarrow_\beta^* N$; that is,

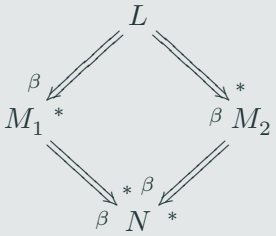


Proof sketch.

By induction on the derivation of $L \Longrightarrow_\beta^* M_1$. □

Theorem 13

If $L \Longrightarrow_\beta^* M_1$ and $L \Longrightarrow_\beta^* M_2$, then there exists a term N such that $M_1 \Longrightarrow_\beta^* N$ and $M_2 \Longrightarrow_\beta^* N$.



Corollary 14

The relation $\longrightarrow\!\!\!\!\!\rightarrow_\beta$ is confluent.

This follows because $M \Longrightarrow_\beta^* N$ is equivalent to $M \longrightarrow\!\!\!\!\!\rightarrow_\beta N$.