Unification of the Lambda-Calculus and Combinatory Logic

Masahiko Sato
Graduate School of Informatics, Kyoto University

IFIP WG 2.2 Meeting
TU Wien
September 25, 2019

What are the Lambda-Calculus and Combinatory Logic?

What are the Lambda-Calculus and Combinatory Logic?

The Preface of "Lambda-Calculus and Combinators, an Introduction" by J.R. Hindley J.P. Seldin says:

The $\boldsymbol{\lambda}$-calculus and combinatory logic are two systems of logic which can also serve as abstract programming languages. They both aim to describe some very general properties of programs that can modify other programs, in an abstract setting not cluttered by details. In some ways they are rivals, in others they support each other.

Plan of the talk

In this talk, I will argue that they are, in fact, one and the same calculus. To show this we unify these two systems into a single system whose syntax naturally contains the syntax of the two systems.

The unification is carried out in three steps:

1. We start from Church's syntax $\mathbf{\Lambda}$ (sometimes called raw terms), but will provide a new way of looking at these terms modulo α-equivalence.
2. We formalize Combinatory Logic by giving a completely new syntax $\mathbf{\Delta}$ for Cobinatory Logic.
3. We obtain the ultimate system by simply taking the union of $\mathbf{\Lambda}$ and $\mathbf{\Delta}$.

History of the calculi

History of the calculi

Again from the Preface of “Lambda-Calculus and Combinators, an Introduction”.

The λ-calculus was invented around 1930 by an American logician Alonzo Church, as part of a comprehensive logical system which included higher-order operators (operators which act on other operators)...

History of the calculi

Again from the Preface of “Lambda-Calculus and Combinators, an Introduction”.

The $\boldsymbol{\lambda}$-calculus was invented around 1930 by an American logician Alonzo Church, as part of a comprehensive logical system which included higher-order operators (operators which act on other operators)...

Combinatory logic has the same aims as $\boldsymbol{\lambda}$-calculus, and can express the same computational concepts, but its grammar is much simpler. Its basic idea is due to two people: Moses Shönfinkel, who first thought of it in 1920, and Haskell Curry, who independently re-discovered it seven years later and turned it into a workable technique.

The syntax of the Lambda Calculus and Combinatory Logic

$$
\begin{aligned}
\mathbb{X} &::= x, y, z, \cdots \\
M, N \in \Lambda &::= x \mid \lambda_x M \mid (M\ N) \\
M, N \in \mathsf{CL} &::= x \mid \mathsf{I} \mid \mathsf{K} \mid \mathsf{S} \mid (M\ N)
\end{aligned}
$$

$(M\ N)$ stands for the application of the function M to its argument N. It is often written simply MN, but we will always use the notation $(M\ N)$ for the application.

The Lambda Calculus

$$M, N \in \Lambda \ ::= \ x \mid \lambda_x M \mid (M\ N)$$

$\lambda_x M$ stands for the function obtained from M by abstracting x in M. We will write $\lambda_{x_0 \cdots x_{n-1}} M$ for $\lambda_{x_0} \cdots \lambda_{x_{n-1}} M$.

β-conversion rule

$$(\lambda_x M\ N) \to [x := N]M$$

Example

If $x \neq y$, and y is not free in M, then

$$\begin{aligned}((\lambda_{xy} x\ M)\ N) &\to ([x := M]\lambda_y x\ N)\\ &= (\lambda_y [x := M]x\ N)\\ &= (\lambda_y M\ N)\\ &\to [y := N]M\\ &= M\end{aligned}$$

Combinatory Logic

$$M, N \in \mathsf{CL} ::= x \mid \mathsf{I} \mid \mathsf{K} \mid \mathsf{S} \mid (M\ N)$$

Weak reduction rules

$$(\mathsf{I}\ M) \to M$$
$$((\mathsf{K}\ M)\ N) \to M$$
$$(((\mathsf{S}\ M)\ N)\ P) \to ((M\ P)\ (N\ P))$$

These rules suggest the following identities.

$$\mathsf{I} = \lambda_x x$$
$$\mathsf{K} = \lambda_{xy} x$$
$$\mathsf{S} = \lambda_{xyz}((x\ z)\ (y\ z))$$

By this identification, every combinatory term becomes a lambda term. Moreover, the above rewriting rules all hold in the lambda calculus.

Combinatory Logic (cont.)

What about the converse direction? We can translate every lambda term to a combinatory term as follows.

$$\begin{aligned} x^* &= x \\ (\lambda_x M)^* &= [x]M^* \\ (M\ N)^* &= (M^*\ N^*) \end{aligned}$$

We used $[-]- : \mathbb{X} \times \mathsf{CL} \to \mathsf{CL}$ above, which we define by:

$$\begin{aligned} [x]x &:= \mathsf{I} \\ [x]y &:= (\mathsf{K}\ y) \quad \text{if } x \neq y \\ [x]M &:= (\mathsf{K}\ M) \quad \text{if } M = \mathsf{I}, \mathsf{K}, \mathsf{S} \\ [x](M\ N) &:= ((\mathsf{S}\ [x]M)\ [x]N) \end{aligned}$$

Combinatory Logic (cont.)

The abstraction operator $[-]-$ enjoys the following property.

$$([x]M\ N) \to [x := N]M$$

So, CL can simulate the β-reduction rule of the λ-calculus. However, the simulation does not provide β-conversion preserving isomorphism. Therefore, for example, the Church-Rosser property for CL does not imply the CR property for the λ-calculus.

Still, the simulated β-reduction has the nice property that substitution is always variable capture-avoiding since CL does not have bound variables.

We will reformulate CL, keeping this nice proerty and at the same time the simulated β-conversion will provide an isomorphism between Λ (modulo α-equivalence) and reformulated CL.

The set $\mathbb{X}$ of variables

We write $\mathbb{X}$ for the set of variables we use in this talk, and use $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ etc. as metavariables ranging over variables.

Moreover we assume that variables in $\mathbb{X}$ are enumertated as:

$$\mathtt{v}_{\mathbf{0}}\ \mathtt{v}_{\mathbf{1}}\ \cdots\ \mathtt{v}_{\boldsymbol{i}}\ \cdots$$

so that any variable $\boldsymbol{x}$ can be written as $\boldsymbol{x} = \mathtt{v}_{\boldsymbol{i}}$ for some uniquely determined natural number $\boldsymbol{i}$.

This enumeration naturally defines a well-ordering on $\mathbb{X}$ definied by: $\mathtt{v}_{\boldsymbol{i}} \leq \mathtt{v}_{\boldsymbol{j}} \Longleftrightarrow \boldsymbol{i} \leq \boldsymbol{j}$.

Height and Thickness of Λ-terms

Definition (Height (Ht), thickness (Th))

$$\begin{aligned} \mathsf{Ht}(x) &:= 0 \\ \mathsf{Ht}(\lambda_x M) &:= \mathsf{Ht}(M) + 1 \\ \mathsf{Ht}((M\ N)) &:= 0 \end{aligned}$$

$$\begin{aligned} \mathsf{Th}(x) &:= 0 \\ \mathsf{Th}(\lambda_x M) &:= \mathsf{Th}(M) \\ \mathsf{Th}((M\ N)) &:= \mathsf{Th}(M) + \mathsf{Th}(N) + 1 \end{aligned}$$

$\mathsf{Ht}(M)$ counts the number of initial sequence of λ-binders, and $\mathsf{Th}(M)$ counts the number of applications in M.

Free variables and Freeness of Λ-terms

Definition (Free variables (FV), freeness (Fn))

$$\begin{aligned} \mathsf{FV}(x) &:= \{x\} \\ \mathsf{FV}(\lambda_x M) &:= \mathsf{FV}(M) - \{x\} \\ \mathsf{FV}((M\ N)) &:= \mathsf{FV}(M) \cup \mathsf{FV}(N) \end{aligned}$$

Given a natural number n and a finite set V of variables, we say that n covers V if $n > i$ for any $\mathtt{v}_i \in V$. Then, the freeness of M, $\mathsf{Fn}(M)$, is the smallest n which covers $\mathsf{FV}(M)$.

Note that $\mathsf{Fn}(M) = 0$ if and only if $\mathsf{FV}(M) = \{\}$.

Height, thickness and freeness are 3 key invariants on α-equivalent terms.

Thread

We will call a term M a thread if $\mathrm{Th}(M) = 0$, namely, if it is constructed from a variable only by abstraction. So, a thread M can be written as

$$M = \lambda_{x_0 \cdots x_{n-1}} y$$

where $n = \mathrm{Ht}(M)$, and if $n = 0$, then $M = y$.

A thread $\lambda_{x_0 \cdots x_{n-1}} y$ is closed if y occurs in $x_0 \cdots x_{n-1}$, and it is open otherwise.

We note that an open thread is characterized up to α-equivalence by n and y, since the choice of x_i are irrelevant as long as they are chosen avoiding y.

Similarly, a closed thread is characterized by a pair of natural numbers i and k such that $y = x_i$, $k = n - 1 - i$ and y is not in $x_{i+1} \cdots x_{n-1}$. The number k is equal to de Bruijn index of the thread.

Standard substitution

Definition (Standard substitution of N for x in M)

$$\begin{aligned}[N/x]x &:= N\\ [N/x]y &:= y \;\text{ if } x \neq y\\ [N/x]\lambda_x M &:= \lambda_x M\\ [N/x]\lambda_y M &:= \lambda_y [N/x]M \;\text{ if } x \neq y\\ [N/x](M_1\ M_2) &:= ([N/x]M_1\ [N/x]M_2)\end{aligned}$$

Standard substitution is a total function on $\Lambda \times \mathbb{X} \times \Lambda$, but in the fourth case, if N has a free occurrence of y, then the standard substitution gives an unwanted result.
Capture-avoiding substitution add a condition that N may not contain free occurreces of y in case four. But, then it is not total on $\Lambda \times \mathbb{X} \times \Lambda$.

Standard term and standard form

Definition (n-standard term and n-standard form)

A Λ-term M is n-standard if $n = \mathsf{Fn}(M)$, $i < n$ for any free variable $\mathtt{v}_i$ in M, and $n \leq i$ for any bound variable $\mathtt{v}_i$ in M, We define the n-standard form of M $(n \geq 0)$ as follows.

$$
\begin{aligned}
[x]^n &:= x \\
[\lambda_x M]^n &:= \lambda_{\mathtt{v}_n}[\mathtt{v}_n/x][M]^{n+1} \\
[(M\ N)]^n &:= ([M]^n\ [N]^n)
\end{aligned}
$$

Proposition

1. If $n \geq \mathsf{Fn}(M)$, then $[M]^n$ is an n-standard term and $[[M]^n]^n = [M]^n$.
2. If $P = (\lambda_x M\ N)$, $n = \mathsf{Fn}(P)$ and P is an n-standard term, then $[N/x]M$ is computed in a capture-avoiding way.

Canonical form of Λ-terms and α-equivalence

Definition (Canonical form)

Given $M \in \Lambda$, we define the α-canonical form of M by putting:

$$M_\alpha := [M]^{\mathsf{Fn}(M)}.$$

It is easy to see that $(M_\alpha)_\alpha = M_\alpha$.

Definition (α-equivalence)

Given two terms M and N, they are α-equivalent, written $M =_\alpha N$, if $M_\alpha = N_\alpha$.

Remark

1. That this is indeed an equivalence relation is obvious.
2. If $n \geq \mathsf{Fn}(M)$, then $[M]^n =_\alpha M$.

Substitution on Λ-terms

Definition (Substitution on Λ-terms)

Given Λ-terms x, M and N, we put $n = \mathsf{Fn}((\lambda_x M\ N))$ and define the result of substituting N for x in M as follows.

$$[x := N]M := ([[N]^n/\mathtt{v}_n][M]^{n+1})_\alpha$$

Substitution is a total function $\mathbb{X} \times \Lambda \times \Lambda$.

Proposition

1. $[x := N]M = [x := N_\alpha]M_\alpha$.
2. If $M_1 =_\alpha M_2$ and $N_1 =_\alpha N_2$, then $[x := N_1]M_1 = [x := N_2]M_2$.

The λ_β-calculus (classical version)

$$\frac{x \in \mathbb{X} \quad M \in \Lambda \quad N \in \Lambda}{(\lambda_x M\ N) \to_\beta [x := N]M}\ \beta$$

$$\frac{M \to_\beta M'}{\lambda_x M \to_\beta \lambda_x M'}\ \xi_{x}$$

$$\frac{M \to_\beta M' \quad N \to_\beta N'}{(M\ N) \to_\beta (M'\ N')}\ \mathsf{A}$$

$$\frac{}{M \to_\beta M}\ \mathsf{I}_M \qquad \frac{M_1 \to_\beta M_2 \quad M_2 \to_\beta M_3}{M_1 \to_\beta M_3}\ \mathsf{C}$$

A different view of Λ-terms

We will provide a different view of $\mathbf{\Lambda}$-terms. This view is obtained by introducing a systematic way of using any $\mathbf{\Lambda}$-term M as an abreviation of M_α. Namely, we will think of α-canotical terms as 'real' λ-terms and other non-canonical terms as 'names' of the corresponding canonical terms.

Given a subset $\mathbf{X}$ of $\mathbf{\Lambda}$, we put

$$[\mathbf{X}] := \{M_\alpha \mid M \in \mathbf{X}\}$$

and introduce the following convention:

$$M : \mathbf{X} \iff M_\alpha \in [\mathbf{X}]$$

Proposition

$$M : \mathbf{X} \iff M \in \bar{\mathbf{X}} := \{M \mid M =_\alpha M \in [\mathbf{X}]\}$$

Classification of Λ-terms by height

We classify $\boldsymbol{\Lambda}$-terms according to their height.
We put:

$$\begin{aligned} \boldsymbol{\Lambda}^{n} &:= \{M \mid \mathsf{Ht}(M) \geq n\} \\ \boldsymbol{\Lambda}^{=n} &:= \boldsymbol{\Lambda}^{n} - \boldsymbol{\Lambda}^{n+1} \end{aligned}$$

We have:

$$\boldsymbol{\Lambda} = \boldsymbol{\Lambda}^{0} = \bigcup_{n=0}^{\infty} \boldsymbol{\Lambda}^{=n} \quad (\text{disjoint union})$$

All the sets defined above commute with the operation $[-]$. For example: $[\boldsymbol{\Lambda}] = \bigcup_{n=0}^{\infty} [\boldsymbol{\Lambda}^{=n}]$.

Application at height i

We generalize traditional application term $(M\ N)$ to terms of the form $(M\ N)^i$ $(i \geq 0)$ (application of M to N at height i) by means of notational convention.

Suppose that $M, N \in \Lambda^i$ and $n = \mathrm{Fn}((M\ N))$. Then we define $(M\ N)^i \in \Lambda^{=i}$ by the rule:

$$\frac{[M]^n = \lambda_{\mathrm{v}_n \cdots \mathrm{v}_{n+i-1}} M' \in \Lambda^i \quad [N]^n = \lambda_{\mathrm{v}_n \cdots \mathrm{v}_{n+i-1}} N' \in \Lambda^i}{(M\ N)^i := (\lambda_{\mathrm{v}_n \cdots \mathrm{v}_{n+i-1}} (M'\ N'))_\alpha \in \Lambda^i}$$

We note that $(-\ -)^i$ is a total function on $\Lambda^i \times \Lambda^i$, and in particular when $i = 0$, then it is total on $\Lambda \times \Lambda$ and $(M\ N)^0 = (M\ N)_\alpha$.

A different view of Λ-terms

We can now check that, for each $n \geq 0$, $[\Lambda^{=n}]$ can inductively generated by the following rules.

$$\frac{x_0, \ldots, x_{n-1}, y \in \mathbb{X}}{\lambda_{x_0 \cdots x_{n-1}} y : \Lambda^{=n}} \qquad \frac{M : \Lambda^n \quad N : \Lambda^n}{(M\ N)^n : \Lambda^{=n}}$$

These rles provide us with simpler induction principle than the traditional induction principle involving variable binding for the case of abstraction.

A different view of Λ-terms (cont.)

We can also understand the above rules as a new form of inducution principle on $\mathbf{\Lambda}$-terms.

The first rule covers threads, namely, those terms whose thickness is 0. Thus, as a base case of new induction priciple, we must first settle this base case (with no IH).

The second rule covers terms with positive thickness, namely, applications. Using the abbreviation just introduced, an application can be written as $(M\ N)^{i}$. The second case is the induction step case, and our induction priciple allows us to use two IHs which correscond to the cases for M and N.

Also while the traditional induction priciple has three cases for induction, one for base case (variale) and two (abstaction and application) cases for step cases, in our case we have one (thread) for base case and one (application) for step case.

Instantiation on Λ-terms

Definition (Instatiation on Λ-terms)

Given $M \in \Lambda^1$ and $N \in \Lambda$, we put $n = \mathsf{Fn}((M\ N))$ and define the result of instantiating M by N as follows.

$$\langle M\ N \rangle := ([[N]^n / \mathtt{v}_n][M]^{n+1})_\alpha$$

Instantiation is a total function $\Lambda^1 \times \Lambda$.

Proposition

If $M = \lambda_x M'$, then we have

$$\langle M\ N \rangle = [x := N]M'.$$

Instatiation on Λ-terms at height i

We can naturally generalize the instatiation operation defined in the previous slides and had the functionality:

$$\langle - \ - \rangle : \mathbf{\Lambda}^1 \times \mathbf{\Lambda}^0 \to \mathbf{\Lambda}^0$$

to instantiation operation at height i so that it will have the functionality:

$$\langle - \ - \rangle^i : \mathbf{\Lambda}^{i+1} \times \mathbf{\Lambda}^i \to \mathbf{\Lambda}^i$$

and satisfies the equation:

$$\langle \boldsymbol{\lambda}_{x_0 \cdots x_{i-1}} \boldsymbol{\lambda}_y M \ \boldsymbol{\lambda}_{x_0 \cdots x_{i-1}} N \rangle^i =_\alpha \boldsymbol{\lambda}_{x_0 \cdots x_{i-1}} \langle \boldsymbol{\lambda}_y M \ N \rangle$$

Instatiation on Λ-terms at height i (cont.)

This generalized instantiation operation enables us to reformulate the classical $\boldsymbol{\lambda}_{\boldsymbol{\beta}}$-calculus in such a way that we can apply $\boldsymbol{\beta}$-conversion to a redex inside several abstractions without appealing to the $\boldsymbol{\xi}$-rule.

The λ_β-calculus (reformulated version)

$$\frac{M \in \Lambda^{i+1} \quad N \in \Lambda^{i}}{(M\ N)^i \to_\beta \langle M\ N\rangle^i}\ \beta$$

$$\frac{M, N \in \Lambda^{i} \quad M \to_\beta M' \quad N \to_\beta N'}{(M\ N)^i \to_\beta (M'\ N')^i}\ \mathsf{A}$$

$$\frac{}{M \to_\beta M}\ \mathsf{I}_M \qquad \frac{M_1 \to_\beta M_2 \quad M_2 \to_\beta M_3}{M_1 \to_\beta M_3}\ \mathsf{C}$$

For comparison, we show the classical version again in the next slide.

The λ_β-calculus (classical version)

$$\frac{}{(\lambda_x M\ N) \to_\beta [x := N]M}\ \beta$$

$$\frac{M \to_\beta M'}{\lambda_x M \to_\beta \lambda_x M'}\ \xi_x$$

$$\frac{M \to_\beta M' \quad N \to_\beta N'}{(M\ N) \to_\beta (M'\ N')}\ \mathsf{A}$$

$$\frac{}{M \to_\beta M}\ \mathsf{I}_M \qquad \frac{M_1 \to_\beta M_2 \quad M_2 \to_\beta M_3}{M_1 \to_\beta M_3}\ \mathsf{C}$$

The datatype Δ of derivations

In order to study the intrinsic structure of Λ we introduce the datatype Δ of derivations.

Definition (The datatype Δ of derivations)

$$\Lambda \ni M, N ::= \ x \mid \lambda_x M \mid (M\ N)$$
$$\Delta \ni d, e ::= \ \mathsf{V}_x^i \mid \mathsf{P}_k^i \mid (d\ e)^i$$

V_x^i are called lifted variables and P_k^i are called projections. Their computational behaviors are characterized by the following β-equalityies.

$$(\mathsf{V}_x^i\ e_1 \cdots e_i)^0 =_\beta \mathsf{V}_x^0$$
$$(\mathsf{P}_k^i\ e_0 \cdots e_{i+k})^0 =_\beta e_i$$

The datatype Δ of derivations (cont.)

We may think of $\mathbf{\Delta}$-terms as a variant of CL-terms. For example, combinators I, K and S are definable in $\mathbf{\Delta}$ as abbreviations:

$$
\begin{aligned}
\mathsf{I} &:= \mathsf{P}^{\mathbf{0}}_{\mathbf{0}} \\
\mathsf{K} &:= \mathsf{P}^{\mathbf{0}}_{\mathbf{1}} \\
\mathsf{S} &:= \left(\left(\mathsf{P}^{\mathbf{0}}_{\mathbf{2}}\ \mathsf{P}^{\mathbf{2}}_{\mathbf{0}}\right)^{\mathbf{3}}\ \left(\mathsf{P}^{\mathbf{1}}_{\mathbf{1}}\ \mathsf{P}^{\mathbf{2}}_{\mathbf{0}}\right)^{\mathbf{3}}\right)^{\mathbf{3}}.
\end{aligned}
$$

Abstraction operation in Δ

In $\boldsymbol{\Delta}$, we can mimic $\boldsymbol{\lambda}$-abstraction in $\boldsymbol{\Lambda}$ by introducing the following notational convention. Given a variable x and a $\boldsymbol{\Delta}$-term d, $[x]d$ stands for the following $\boldsymbol{\Delta}$-term.

$$
\begin{aligned}
[x]\mathsf{V}_x^i &:= \mathsf{P}_i^0 \\
[x]\mathsf{V}_y^i &:= \mathsf{V}_y^{i+1} \quad \text{if } x \neq y \\
[x]\mathsf{P}_k^i &:= \mathsf{P}_k^{i+1} \\
[x](d\ e)^i &:= ([x]d\ [x]e)^{i+1}
\end{aligned}
$$

Recall that, for CL, it was defined by:

$$
\begin{aligned}
[x]x &:= \mathsf{I} \\
[x]y &:= (\mathsf{K}\ y) \quad \text{if } x \neq y \\
[x]M &:= (\mathsf{K}\ M) \quad \text{if } M = \mathsf{I}, \mathsf{K}, \mathsf{S} \\
[x](M\ N) &:= ((\mathsf{S}\ [x]M)\ [x]N)
\end{aligned}
$$

Translation from Λ to Δ

We translate each $\mathbf{\Lambda}$-term M into a $\mathbf{\Delta}$-term M^* as follows.

$$\begin{aligned} x^* &:= \mathsf{V}_x^0 \\ (\lambda_x M)^* &:= [x]M^* \\ (M\ N)^* &:= (M^*\ N^*)^0 \end{aligned}$$

This translation naturally induces an instantiation preserving isomorphism $[\mathbf{\Lambda}] \simeq \mathbf{\Delta}$.

Unification of Λ and Δ

Definition (The unified syntax $\boldsymbol{\Lambda\Delta}$)

$$\begin{aligned}
\boldsymbol{\Lambda} \ni M, N &::= \ x \mid \lambda_x M \mid (M\ N) \\
\boldsymbol{\Delta} \ni d, e &::= \ \mathsf{V}_x^i \mid \mathsf{P}_k^i \mid (d\ e)^i \\
\boldsymbol{\Lambda\Delta} \ni d, e &::= \ x \mid \lambda_x d \mid (d\ e) \mid \mathsf{V}_x^i \mid \mathsf{P}_k^i \mid (d\ e)^i
\end{aligned}$$

We may think of $\boldsymbol{\Lambda}$ and $\boldsymbol{\Delta}$ as two sides of the same coin $\boldsymbol{\Lambda\Delta}$.

In $\boldsymbol{\Lambda\Delta}$, we can freely mix syntax from two languages $\boldsymbol{\Lambda}$ and $\boldsymbol{\Delta}$.