Lambda Calculus and Types

Untyped Lambda Calculus

陳亮廷 Chen, Liang-Ting

2018 邏輯、語言與計算暑期研習營
Formosan Summer School on
Logic, Language, and Computation

Swansea University, UK

Introduction

λ-calculus ...

1. was developed by Alonzo Church (Turing's PhD supervisor)
2. is a model of computation
3. is a backbone of programming languages

As a programming language, it only supports 3 constructs

1. variable
2. function definition
3. function application

Syntax of Lambda Calculus

Terms of λ-calculus

Definition 1 (Syntax of λ-calculus)

Let $V := \{x, y, z, \ldots\}$ be a countably infinite set of *variables*.
The set Λ of λ-terms is defined by

$$\frac{x \in V}{x \in \Lambda}\ (\text{var})$$

$$\frac{M \in \Lambda \qquad N \in \Lambda}{(M\,N) \in \Lambda}\ (\text{app})$$

$$\frac{M \in \Lambda \qquad x \in V}{\lambda x.\, M \in \Lambda}\ (\text{abs})$$

Application is left associative; abstraction is right associative.

Meta-Language and Object-Language

- *Meta-language* is the language we use to describe the object of study. E.g., naive set theory.
- *Meta-variable* is a placeholder *in* the meta-language.
- *Object-language* is the object of study. E.g., arithmetic expressions, λ-terms, etc.
- *Variable* refers to some variable of λ-calculus.

Example 2

As *naming* a function is not supported in λ-calculus, we do so in the meta-language:

$$\texttt{id} := \lambda x.\, x$$

`id` is a synonym of the λ-term on RHS in the meta-language.

Example 3 (Projections)

The first and the second projections are made of abstractions and variables only:

$$\texttt{fst} := \lambda x.\, \lambda y.\, x \quad \text{and} \quad \texttt{snd} := \lambda x.\, \lambda y.\, y$$

For brevity $\lambda x_1\, x_2\, \ldots x_n.\, M := \lambda x_1.\, (\lambda x_2.\, (\ldots (\lambda x_n.\, M) \ldots))$.

Hence, projections are equal to

$$\lambda x\, y.\, x \quad \text{and} \quad \lambda x\, y.\, y$$

In Haskell: $\backslash\texttt{x y} \rightarrow \texttt{x}$ and $\backslash\texttt{x y} \rightarrow \texttt{y}$. However, $\texttt{fst} = \backslash\texttt{x y} \rightarrow \texttt{x}$ is a proper term in Haskell (object-language).

α-equivalence, informally

Definition 4

Two λ-terms are α-equivalent if variables *bound* by abstractions can be renamed to derive the same term.

Example 5

1. $\lambda x.\, x \neq \lambda y.\, y$ but $\lambda x.\, x \equiv_\alpha \lambda y.\, y$.
2. $\lambda x.\, \lambda y.\, y \equiv_\alpha \lambda z.\, \lambda y.\, y$.
3. $\lambda x.\, \lambda y.\, x \not\equiv_\alpha \lambda x.\, \lambda y.\, y$.

α-equivalent terms are considered 'programs of the same structure'. Renaming variables do not change program behaviour but readability.

Concrete and Abstract Syntax

Concrete Syntax

A string possibly annotated with brackets and other delimiters.

Abstract Binding Tree

A tree structure with pointers where each node is an operator with arguments as its sub-trees.

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

Operational Semantics

Evaluation, informally

- The term $(M\ N)$ is understood as a function application where N is the argument for the 'function' M.
- The term $\lambda x.\, L$ is understood as a function with a parameter x and the function body L.
- The only 'computation' in λ-calculus is function application:
$$(\lambda x.\, M)\, N \longrightarrow M[x \mapsto N]$$
where $M[x \mapsto N]$ means that x in M is substituted for N.

How to evaluate the following terms?

1. $(\lambda x.\, \lambda y.\, x)\, M\, N$
2. $(\lambda y.\, \lambda y.\, y)\, M$
3. $(\lambda x.\, \lambda y.\, x)\, y$

Definition 6

For $x \in V$ and $M \in \Lambda$, the substitution of x for M is defined by

$$
\begin{aligned}
x[x \mapsto M] &= M \\
y[x \mapsto M] &= M && \text{if } x \neq y \\
(M\,N)[x \mapsto L] &= (M[x \mapsto L]\,N[x \mapsto L]) \\
(\lambda x.\,M)[y \mapsto N] &= \lambda x.\,M[y \mapsto N]
\end{aligned}
$$

A bound variable may become free.

$$(\lambda x.\,x)[x \mapsto y] = \lambda x.\,y$$

Naive Substitution ii

Definition 7

For $x \in V$ and $M \in \Lambda$, the substitution of x for M is defined by

$$
\begin{aligned}
x[x \mapsto M] &= M \\
y[x \mapsto M] &= M && \text{if } x \neq y \\
(M\,N)[x \mapsto L] &= (M[x \mapsto L]\,N[x \mapsto L]) \\
(\lambda x.\,M)[y \mapsto N] &= \lambda x.\,M[y \mapsto N] && \text{if } x \neq y \\
(\lambda x.\,M)[y \mapsto N] &= \lambda x.\,M && \text{if } x = y
\end{aligned}
$$

A variable may be captured by an abstraction.

$$(\lambda x.y)[y \mapsto x] = \lambda x.\,x$$

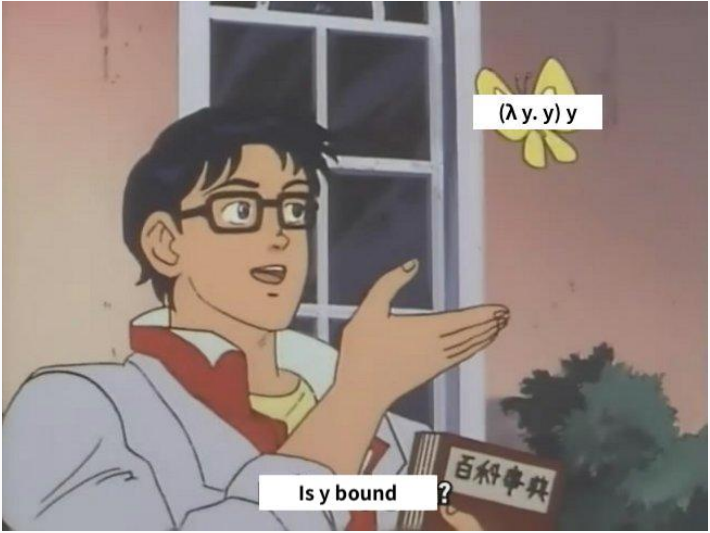
Definition 8

The set **FV** of free variables of a term M is defined by

$$\begin{aligned}
\mathbf{FV}(x) &= \{x\} \\
\mathbf{FV}(\lambda x.\, M) &= \mathbf{FV}(M) - \{x\} \\
\mathbf{FV}(M\ N) &= \mathbf{FV}(M) \cup \mathbf{FV}(N)
\end{aligned}$$

A variable y which occurs in M is free if $y \in \mathbf{FV}(M)$; bound otherwise. A λ-term M is *closed* or a *combinator* if $\mathbf{FV}(M) = \emptyset$.

1. A variable can be *free* and *bound* at the same time.
2. An occurrence of a variable can only be either *free* or *bound*.



1. A variable can be *free* and *bound* at the same time.
2. An occurrence of a variable is either *free* or *bound*.

Capture-Avoiding Substitution

Capture-avoiding substitution of L for the free occurrences of x is a *partial* function from λ-terms to λ-terms defined by

$$
\begin{aligned}
x[x \mapsto L] &= L \\
y[x \mapsto L] &= y && \text{if } x \neq y \\
(M\,N)[x \mapsto L] &= (M[x \mapsto L]\; N[x \mapsto L]) \\
(\lambda x.\, M)[x \mapsto L] &= \lambda x.\, M \\
(\lambda y.\, M)[x \mapsto L] &= \lambda y.\, M[x \mapsto L] && \text{if } x \neq y \text{ and } y \notin \mathbf{FV}(L)
\end{aligned}
$$

Definition 9 (Freshness)

A variable y is fresh for L if $y \notin \mathbf{FV}(L)$.

Congruence

Definition 10

1. A congruence on λ-terms is a relation R on λ-terms subject to following rules

$$\frac{M_1 \; R \; M_2 \qquad N_1 \; R \; N_2}{(M_1 \, N_1) \; R \; (M_2 \, N_2)} \qquad\qquad \frac{M_1 \; R \; M_2}{(\lambda x.\, M_1) \; R \; (\lambda x.\, M_2)}$$

2. The congruence closure $\overline{R}$ of a relation R is the smallest congruence containing R.
3. The congruence and equivalence closure of a relation R is the smallest congruence and equivalence relation containing R.

Homework

Define congruence closure using inference rules.

Renaming of Bound Variables

If a variable y is *fresh* for M, the bound variable x of $\lambda x.\, M$ to y can be renamed without changing the meaning.

Definition 11 (α-conversion)

α-conversion is a relation $\to_\alpha$ defined by

$$(\lambda x.\, M) \longrightarrow_\alpha \lambda y.\, M[x \mapsto y] \quad \text{if } y \text{ is fresh for } M.$$

Let $=_\alpha$ be the congruence and equivalence closure of $\longrightarrow_\alpha$. We say that M and N are *α-equivalent* if $M =_\alpha N$.

Convention

λ-terms are equal up to α-equivalence/renaming of bound variables.

η-conversion

Pointy style is *assumed* to be equivalent to point-free style.

η-reduction $\lambda x.\,(M\,x) \longrightarrow_{\eta} M$

η-expansion $M \longrightarrow_{\eta} \lambda x.\,(M\,x)$ where x is fresh for M.

η-equivalence the congruence and equivalence closure of $\longrightarrow_{\eta}$.

η-equivalence is a form of *extensionality* limited to λ-terms.

$$f = g \iff \forall x.\, f(x) = g(x)$$

Evaluation i

Definition 12 (β-conversion)

β-conversion is a relation defined by

$$(\lambda x.\, M)\, N \longrightarrow_\beta M[x \mapsto N]$$

Any term of this form $(\lambda x.\, M)\, N$ is called a β-*redex*.

Good:

$$((\lambda x.\, \lambda y.\, x)\, M) \longrightarrow_\beta (\lambda y.\, x)[x \mapsto M] = \lambda y.\, x[x \mapsto M] = \lambda y.\, M$$

Bad:

$$(\lambda x.\, \lambda y.\, x)\, M\, N \longrightarrow_\beta ?$$

Evaluation ii

Definition 13

The full β-reduction is a relation on λ-terms defined by

$$\frac{M_1 \longrightarrow_{\beta} M_2}{M_1 \longrightarrow_{\beta 1} M_2} \qquad \frac{M_1 \longrightarrow_{\beta 1} M_2}{M_1\, N \longrightarrow_{\beta 1} M_2\, N}$$

$$\frac{M_1 \longrightarrow_{\beta 1} M_2}{\lambda x.\, M_1 \longrightarrow_{\beta 1} \lambda x.\, M_2} \qquad \frac{N_1 \longrightarrow_{\beta 1} N_2}{M\, N_1 \longrightarrow_{\beta 1} M\, N_2}$$

Now fixed:

$$(\lambda x.\, \lambda y.\, x)\, M\, N \longrightarrow_{\beta 1} (\lambda y.\, M)\, N \longrightarrow_{\beta 1} M[\, y \mapsto N\,] \longrightarrow_{\beta 1} \ldots$$

Programming in Lambda Calculus

Boolean values and conditional can be encoded as closed λ-terms.

Boolean

$$
\begin{aligned}
\texttt{True} &:= \lambda x\, y.\, x \\
\texttt{False} &:= \lambda x\, y.\, y
\end{aligned}
$$

Conditional

$$\texttt{if\_then\_else\_} := \lambda b\, x\, y.\, b\, x\, y$$

$$\texttt{if True then } M \texttt{ else } N \longrightarrow_{\beta *} M$$

$$\texttt{if False then } M \texttt{ else } N \longrightarrow_{\beta *} N$$

for any two λ-terms M and N.

Natural numbers can be encoded as λ-terms, so can arithmetic operations.

Church numerals

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

for $n > 0$ where $f^{n+1}(M) := f(f^n\, M))$.

Successor

$$\begin{aligned} \mathtt{succ} &:= \lambda n.\,\lambda f x.\,f\,(n\,f\,x) \\ \mathtt{succ}\ \mathsf{c}_n &\longrightarrow_{\beta *} \mathsf{c}_{n+1} \end{aligned}$$

for any natural number $n \in \mathbb{N}$.

Predecessor

$$\begin{aligned} \mathtt{pred} &:= \lambda n.\,\lambda f x.\,? \\ \mathtt{pred}\ \mathsf{c}_0 &\longrightarrow_{\beta *} \mathsf{c}_0 \\ \mathtt{pred}\ \mathsf{c}_{n+1} &\longrightarrow_{\beta *} \mathsf{c}_n \end{aligned}$$

Addition

$$\begin{aligned} \mathtt{add} &:= \lambda n\, m.\, \lambda f x.\, m\, f\, (n\, f\, x) \\ \mathtt{add}\ \mathtt{c}_n\ \mathtt{c}_m &\longrightarrow_{\beta*} \mathtt{c}_{n+m} \end{aligned}$$

Conditional

$$\begin{aligned} \mathtt{ifz} &:= \lambda n\, x\, y.\, n\, (\lambda z.\, y)\, x \\ \mathtt{ifz}\ \mathtt{c}_0\ M\ N &\longrightarrow_{\beta*} M \\ \mathtt{ifz}\ \mathtt{c}_{n+1}\ M\ N &\longrightarrow_{\beta*} N \end{aligned}$$

Here is a list of common combinators:

1. $\omega := \lambda x.\, x\, x$
2. $\Omega := (\lambda x.\, x\, x)\,(\lambda x.\, x\, x) = \omega\,\omega$
3. $\mathsf{I} := \lambda x.\, x$, the *identity*.
4. $\mathsf{S} := \lambda f\, g\, x.\, (f\, x)\,(g\, x)$.

Exercise

1. Evaluate $\mathtt{succ}\, \mathsf{c}_0$ and $\mathtt{add}\, \mathsf{c}_1\, \mathsf{c}_2$.
2. Define `pred`.
3. Define Boolean operations, i.e. `not`, `and`, and `or`.
4. Is ω allowed in Haskell?

The summation $\sum_{i=0}^{n} i$ for $n \in \mathbb{N}$ *can be* defined as

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

Can we avoid the self-reference? Consider the function $G \colon (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ defined by

$$(Gf)(n) := \begin{cases} 0 & \text{if } n = 0 \\ n + f(n-1) & \text{otherwise.} \end{cases} \tag{1}$$

Assuming that $\mathtt{sum}'$ is a fixed-point of G, i.e. $G(\mathtt{sum}') = \mathtt{sum}'$, we can show that $\mathtt{sum}' = \mathtt{sum}$ by induction.

Proposition 14 (Curry's paradoxical combinator)

Define

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

Then,

$$\begin{aligned} \mathsf{Y}F &\longrightarrow_{\beta 1} \underline{(\lambda x.\,F\,(x\,x))\,(\lambda x.\,F\,(x\,x))} \\ &\longrightarrow_{\beta 1} F\,(\underline{(\lambda x.\,F\,(x\,x))\,(\lambda x.\,F\,(x\,x))}) \end{aligned}$$

for every λ-term F.

Example 15 (Summation, formally)

Using the combinators we have known so far, the equation (1) can be defined as λ-terms:

$$\begin{aligned} G &:= \lambda f\, n.\, \mathtt{ifz}\ n\ b c_0\ (\mathtt{add}\ n\ (f\ (\mathtt{pred}\ n))) \\ \mathtt{sum} &:= \mathsf{Y} G \end{aligned}$$

Try to evaluate $\mathtt{sum}$ with, say, c_3.

Here is a fixed-point operator such that $\Theta F \longrightarrow_{\beta*} F(\Theta F)$.

Proposition 16 (Turing's fixed-point combinator)

Define

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

Then,

$$\Theta F \longrightarrow_{\beta*} F(\Theta F)$$

Try Turing's fixed-point combinator with G to define $\sum_{i=0}^{n} i$.

$$\begin{aligned} G &:= \lambda f\, n.\, \mathtt{ifz}\ n\ bc_0\ (\mathtt{add}\ n\ (f(\mathtt{pred}\ n))) \\ \mathtt{sum} &:= \Theta\, G \end{aligned}$$

Exercise

1. Define the *flip* operation, i.e. a λ-term `flip` such that

$$\texttt{flip}\ M\ N\ P \longrightarrow_{\beta *} M\ P\ N$$

2. Define the multiplication $m \times n$ on Church numerals.
3. Define the factorial $n!$ on Church numerals.

Properties of Lambda Calculus

Example 17

Suppose $M \in \Lambda$ and $y \notin \mathsf{FV}(M)$. Then, consider

$$(\lambda y.\, M)\,((\lambda x.\, x\,x)(\lambda x.\, x\,x))$$

Observations:

- Some evaluation may diverge while some may converge.
- Full β-reduction lacks for determinacy.

Question:

- Does every path give the same evaluation?

Let $\longrightarrow_{\beta *}$ denote the reflexive and transitive closure of $\longrightarrow_{\beta 1}$.

Theorem 18 (Church-Rosser Property)

Given N_1 and N_2 with $M \longrightarrow_{\beta *} N_1$ and $M \longrightarrow_{\beta *} N_2$, there is L such that

$$\begin{tikzcd} & M \arrow[dl, "\beta *"'] \arrow[dr, "\beta *"] & \\ N_1 \arrow[dr, dashed, "\beta *"'] & & N_2 \arrow[dl, dashed, "\beta *"] \\ & L & \end{tikzcd}$$

A term is in normal form if $M \not\longrightarrow_{\beta 1}$.

Corollary 19 (Uniqueness of normal forms)

Suppose that N_1 and N_2 are in normal form. Then,

$$
\begin{tikzcd}
 & M \arrow[dl, "\beta *"'] \arrow[dr, "\beta *"] & \\
N_1 = & N & = N_2
\end{tikzcd}
$$

Homework

Show this corollary.

Corollary 20

Let $=_\beta$ denote the congruence closure of $\longrightarrow_{\beta 1}$.

1. *If $M =_\beta N$, then there exists L such that*

$$M \longrightarrow_{\beta *} L \;{}_{\beta *}\!\longleftarrow N$$

2. *If in addition N is in normal form, then $M \longrightarrow_{\beta *} N$.*

Homework

Show this corollary.

Evaluation Strategies i

An evaluation strategy is a procedure of selecting β-redexes to reduce. It is a subset $\longrightarrow_{\mathrm{ev}}$ of the full β-reduction $\longrightarrow_{\beta 1}$.

Innermost β-redex does not contain any β-redex.

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

the leftmost-outermost strategy reduces the leftmost outermost β-redex in a λ-term first. For example,

$$
\begin{aligned}
& \underline{(\lambda x.\,(\lambda y.\,y)\,x)} \quad \underline{(\lambda x.\,(\lambda y.\,y\,y)\,x)} \\
\longrightarrow_{\beta 1} & \underline{(\lambda y.\,y)} \quad \underline{(\lambda x.\,(\lambda y.\,y\,y)\,x)} \\
\longrightarrow_{\beta 1} & \lambda x.\,\underline{(\lambda y.\,y\,y)} \quad \underline{x} \\
\longrightarrow_{\beta 1} & (\lambda x.\,x\,x) \\
\not\longrightarrow_{\beta 1} &
\end{aligned}
$$

Evaluation Strategies iii

the leftmost-innermost strategy reduces the leftmost innermost β-redex in a λ-term first. For example,

$$
\begin{aligned}
& (\lambda x.\, \underline{(\lambda y.\, y)\ \ x})\,(\lambda x.\,(\lambda y.\, y\,y)\,x) \\
\longrightarrow_{\beta 1} & (\lambda x.\, x)\ (\lambda x.\, \underline{(\lambda y.\, y\,y)\ \ x}) \\
\longrightarrow_{\beta 1} & \underline{(\lambda x.\, x)\ \ (\lambda x.\, x\,x)} \\
\longrightarrow_{\beta 1} & (\lambda x.\, x\,x) \\
\not\longrightarrow_{\beta 1} &
\end{aligned}
$$

the rightmost-innermost/outermost strategy are defined similarly where λ-terms are reduced from right to left instead.

Evaluation Strategies iv

Call-by-value strategy rightmost-outermost but not inside any λ-abstraction

Call-by-name strategy leftmost-outermost but not inside any λ-abstraction

Proposition 21 (Determinacy)

Each of evaluation strategies is deterministic.

Exercise

Evaluate Ω and $\mathsf{K}_1\,(\lambda x.\,x)\,\Omega$ respectively using call-by-value and call-by-name strategy where

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

Homework

Draw and evaluate above λ-terms in abstract syntax tree with the rightmost-innermost/outermost strategies.

Definition 22

1. M is in *normal form* if $M \not\longrightarrow_{\beta 1} N$ for any N.
2. M is *weakly normalising* if $M \longrightarrow_{\beta *} N$ for some N in normal form.

1. Ω does not have a normal form.
2. K_1 is normal and thus weakly normalising.
3. $(\mathsf{K}_1 z)\,\Omega$ is weakly normalising.

Normalising ii

Theorem 23

The leftmost-outermost strategy reduces every weakly normalising λ-term M to its normal form N.

Definition 24

For λ-calculus, a value is just a λ-abstraction.

Proposition 25

Under call-by-name and call-by-value strategies, every value is in normal form.

Remark

The definition of capture-avoiding substation is widely adopted, it is still ill-defined. Recursion is always a total function. Advanced mathematics [Pit13] is needed to resolve this issue.

Issues with named variables may be lifted by using nameless representation of terms. For example, in de Bruijin's representation every λ-term has a canonical form, see [Pie02, Chapter 6].

In fact, every computable function on natural numbers is definable in terms of λ-terms. For interested readers, see [BB84, Chapter 3] for further detail. Therefore, λ-calculus is Turing-complete.

References i

- Henk Barendregt and Erik Barendsen, *Introduction to lambda calculus*, Nieuw Arch. voor Wiskd. **4** (1984), no. 2, 337–372.
- Benjamin C. Pierce, *Types and programming languages*, MIT Press, 2002.
- Andrew M. Pitts, *Nominal sets*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, July 2013.