λ-Calculus

Untyped λ-Calculus

陳亮廷 Chen, Liang-Ting

Formosan Summer School on Logic, Language, and Computation 2024

Institute of Information Science
Academia Sinica

評分標準

期限 七月十號下午 4 點 20

分數比例 作業 (22.5%)
考試 (100%)

Email `liang.ting.chen.tw@gmail.com`

作業繳交請遵循以下規定

1. 請用 A4 紙（其餘規格尺寸不收），寫上姓名與學號
2. 請清楚標示作業次、題號並且力求字跡清楚
3. 作業可現場親交給我，或用 PDF 檔案（如翻拍請用掃描軟體後製）後寄到我 email 信箱，格式如下：

   主旨 `[FLOLAC] Lambda HW%x%`
   附件 `Lambda-HW{1, 2, 3} - 學號 - 姓名.pdf`
   內文 (可空白)

Untyped λ-Calculus: Introduction

Anonymous functions can be defined in many languages, e.g.,

Haskell `\x f -> f x`

OCaml `fun x f -> f x`

This type of expression is inspired by the λ-notation introduced by Alan Turing's supervisor, Alonzo Church, who was seeking a foundation for mathematics.

In λ-notation

$$\lambda x.\, e$$

means 'a function that maps the argument x to expression e' where x may appear in e. E.g., the above examples can be expressed as

$$\lambda x\, f.\, f\ x$$

An example of λ-notation

The idea of function application in λ-notation is straightforward.

For example, in high school we may say a function $f(x) := x^2$ with the variable x and write

$$f(3) = 3^2 = 9$$

In λ-notation, we write

$$(\lambda x.\, x^2)\ 3 = x^2[3/x] = 3^2 = 9$$

where $x^2[3/x]$ means 'the substitution of 3 for x in the expression x^2'.

What is λ-calculus

λ-calculus is a *language of functions in λ-notation* consisting of three constructs:

abstraction	functions can be introduced $\lambda x. t$
application	functions can be applied to an argument $t\ u$
variable	variables are terms

where a *term* means a minimal unit of expression.

That is, every term in λ-calculus is in one and only one of the above forms.

λ-calculus can be understood as a programming language *without* any built-in data types and suffices to define every computable function.

λ-calculus itself is a fruitful subject but it is also useful:

- it serves as a prototype of programming languages which can be reasoned about mathematically and rigorously;
- the methodology we develop to understand λ-calculus can be used to study and design other programming languages.

The common practice in PL research is to start with a variant of typed λ-calculus and a *language feature* in question and investigate properties of this prototype language.

Moreover, λ-calculus has a strong connection with *logic* and *mathematics* which is a topic for another day.

What to expect next?

For λ-calculus, we will consider following topics in programming language in a style of mathematical formalism.

1. How programs can be identified *up to variable renaming*? E.g., $\lambda x.\, x$ should be 'equal' to $\lambda y.\, y$.
2. How do programs *compute*? E.g., the application $(\lambda x.\, x)\, 3$ of the identity to 3 should compute to 3.
3. How programs can be identified *computationally*? E.g.,

   $$(\lambda x.\, x)\, 3 \quad \text{and} \quad (\lambda y.\, 3)\, 10$$

   should be 'computationally equal' as they should compute to the same term (but not each other).
4. How to write programs in λ-calculus?

Untyped λ-Calculus: Statics

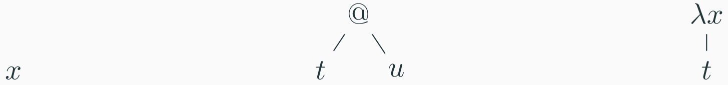
Syntax of λ-calculus

To define the language of λ-calculus, we need a primitive notion of *variables* first. Let us fix a *countably infinite* set V for variables.

The set $\Lambda(V)$ of λ-terms over V is defined *inductively* as

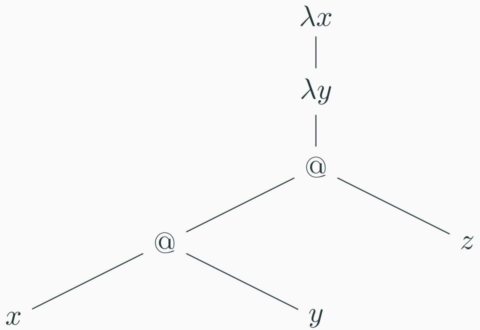
variable	$x \in \Lambda(V)$ if x is in V
application	$t@u \in \Lambda(V)$ if $t, u \in \Lambda(V)$
abstraction	$\lambda x.\, t$ if $x \in V$ and $t \in \Lambda(V)$

Each construct can be represented as a node in a tree, i.e.



for a variable x, an application $t@u$, and an abstraction $\lambda x.\, t$.

The expression $\lambda x.\,(\lambda y.\,((x@y)@z))$ can be represented as



Important

Brackets '(' and ')' are not part of a term but are used for grouping a subterm.

The validity of the expression can be justified by its very definition:

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

1. x, y, and z are in V, so x, y, z are terms;
2. x and y are terms, so $x@y$ is a term;
3. $(x@y)@z$ is a term since $x@y$ is a term and z is a term;
4. $\lambda y.\,((x@y)@z)$ is a term since $(x@y)@z$ is a term and y is a variable;
5. $\lambda y.\,((x@y)@z)$ is a term and x is a variable, so $\lambda x.\,(\lambda y.\,((x@y)@z))$ is a term.

Convention

@ *is omitted if a term is written as a sequence of symbols, so we write*

$$t\ u \qquad \textit{instead of} \qquad t@u$$

The need for some conventions

For arithmetic expressions, we typically write

$$3 * 4 + 7 * 2 \qquad \text{to mean} \qquad (3 * 4) + (7 + 2)$$

by the precedence convention.

We'd also like to have some conventions to omit brackets without any ambiguity. E.g., one should be able to write

$$\lambda xy.\, x\ y\ z \qquad \text{to mean} \qquad \lambda x.\, (\lambda y.\, ((x\ y)\ z))$$

because

1. multiple abstractions means a function with multiple arguments;
2. applying a function to multiple arguments can be achieved by applying a function to an argument to get another function for the next argument;
3. applications occur more often than abstractions in a body.

Consecutive abstractions

$$\lambda x_1\, x_2\ \dots x_n.\, M \equiv \lambda x_1.\, (\lambda x_2.\, (\dots (\lambda x_n.\, M) \dots))$$

Consecutive applications

$$M_1\ M_2\ M_3\ \dots\ M_n \equiv (\dots ((M_1\ M_2)\ M_3) \dots)\ M_n$$

Function body extends as far right as possible $\lambda x.\, M\ N$ **means** $\lambda x.\, (M\ N)$ **instead of** $(\lambda x.\, M)\ N$.

1. $(x\ y)\ z \equiv x\ y\ z$
2. $\lambda s.\, (\lambda z.\, (s\ z)) \equiv \lambda s\, z.\, s\ z$
3. $\lambda a.\, (\lambda b.\, (a\ (\lambda c.\, a\ b))) \equiv \lambda a\, b.\, a\ (\lambda c.\, a\ b)$
4. $(\lambda x.\, x)\ (\lambda y.\, y) \equiv (\lambda x.\, x)\ \lambda y.\, y$

Exercise

Draw the corresponding abstract syntax tree for each of the following terms:

1. $x\ (y\ z)$
2. $x\ y\ z$
3. $\lambda sz.\, s\ z$
4. $(\lambda x.\, x)\ (\lambda y.\, y)$
5. $\lambda ab.\, a\ (\lambda c.\, a\ b)$

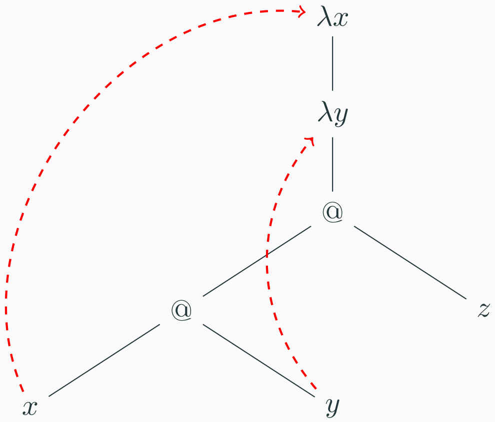
Let's discuss an important notion of syntax: *variable binding*.

In the expression $f(x) = x^2$, the variable x in the expression x^2 is bound to x of f and the meaning of $f(x)$ is the same as $f(y) = y^2$.

Similarly, following expressions exhibit the variable binding in various forms:

1. $\sum_{x=0}^{n} x$
2. $\int_0^1 e^y \, \mathrm{d}y$
3. $f(x, y) = x^2 + y^2$
4. ...
5. $\lambda y. (\lambda x. y)$ means a function that takes an argument y returns a constant function at y
6. $\lambda x. (\lambda y. y)$ means a constant function that always returns the identity

The binding structure can be visualised in an abstract syntax tree:



It is common sense that renaming variables of a program should not alter its meaning: the point of having a name for a variable to look for where it applies to.

Intuitively, two terms t and u are α-equivalent, written as

$$t =_\alpha u$$

if t and u have the same binding structure, *regardless of their variable names*, in their abstract syntax trees.

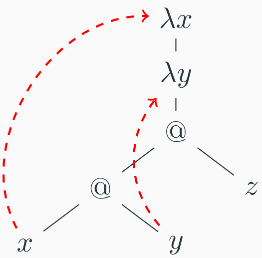
Quest

How to define α-equivalence formally?

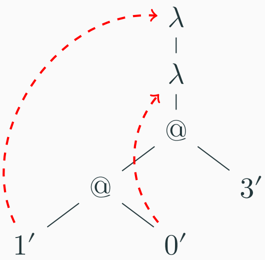
Idea

We discard names completely and use indices i' to represent variable bindings.

The index i' points to the i-th innermost λ-node from the variable:



becomes



This representation is invented by a Dutch mathematician, N. G. de Bruijn, while implementing a system for formalising mathematics.

Good This representation does solve many problems:

1. α-equivalence coincides with syntactic equality, i.e.

$$t =_\alpha u \iff t = u.$$

2. Machine-readable.
3. No variable renaming is involved.

Bad 'Don't throw the baby out with the bathwater'.

Idea

Using the nominal representation, we define $t =_\alpha u$ if variables t and u can be renamed 'suitably' to exactly the same term.

With naive renaming, a renamed variable might be captured by some λ, breaking the binding structure. For example, y can be renamed to anything but x in

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

to preserve the same binding structure.

Hence, variable renaming has to be constrained to variables that do not occur in the term to avoid changing the binding structure.

Quest

How to define the occurrence of a variable and variable renaming?

To define a function from λ-terms, we may use the 'fold':

Theorem

Given a target set X and functions

$$f_1 \colon V \to X$$
$$f_2 \colon X \times X \to X$$
$$f_3 \colon V \times X \to X$$

there exists a unique $\hat{f} \colon \Lambda(V) \to X$ such that

$$\hat{f}\, x = f_1\, x \qquad \hat{f}(t\ u) = f_2(\hat{f}\, t, \hat{f}\, u) \qquad \hat{f}(\lambda x.\, t) = f_3(x, \hat{f}\, t)$$

We define the set $\mathbf{Var}(t)$ of variables in a term t by structural recursion with the target set $\mathcal{P}V$ and

$$\mathbf{Var}_1(x) = \{x\}$$
$$\mathbf{Var}_2(S_1, S_2) = S_1 \cup S_2$$
$$\mathbf{Var}_3(x, S) = \{x\} \cup S$$

That is, $\mathbf{Var}$ is a function from $\Lambda(V)$ to $\mathcal{P}V$ such that

$$\mathbf{Var}(x) = \{x\}$$
$$\mathbf{Var}(t\ u) = \mathbf{Var}(t) \cup \mathbf{Var}(u)$$
$$\mathbf{Var}(\lambda x.\, t) = \{x\} \cup \mathbf{Var}(t)$$

We say x occurs in t if $x \in \mathbf{Var}(t)$, i.e. x appear in t somewhere.

A *transposition* $(x\,y)$ is a function that swaps x and y but fixes everything else, i.e.

$$(x\,y)\ z = \begin{cases} y & z = x \\ x & z = y \\ z & \text{otherwise} \end{cases}$$

The *variable permutation* by a transportation $\pi = (y\,z)$ is defined by

$$\begin{aligned} \pi \cdot x &= \pi\ x \\ \pi \cdot (t\ u) &= (\pi \cdot t)\ (\pi \cdot u) \\ \pi \cdot (\lambda x.\, t) &= \lambda(\pi\ x).\, (\pi \cdot t) \end{aligned}$$

E.g.,

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

Now we are ready to formulate what we mean by α-equivalence

Definition 1 (α-equivalence)

α-equivalence is a relation $t =_\alpha u$ between terms t and u defined inductively as

$$\frac{}{x =_\alpha x} \text{ if } x \in V \qquad \frac{t_1 =_\alpha t_2 \qquad u_1 =_\alpha u_2}{t_1\ u_1 =_\alpha t_2\ u_2}$$

$$\frac{(z\ x) \cdot t =_\alpha (z\ y) \cdot u}{\lambda x.\, t =_\alpha \lambda y.\, u} \text{ if } z \notin \mathbf{Var}(t, u)$$

The third case is the interesting one: $\lambda x.\, t$ and $\lambda y.\, u$ are equal up to renaming bound variables if the variables x and y can be swapped with a variable z that does not exist in t and u.

Example 2

Show that $(\lambda y.\, y)\ z =_\alpha (\lambda x.\, x)\ z$.

Proof.

By definition

$$\dfrac{\dfrac{\dfrac{}{(y\ y) \cdot y =_\alpha (y\ x) \cdot x}}{\lambda y.\, y =_\alpha \lambda x.\, x} \qquad \dfrac{}{z =_\alpha z}}{(\lambda y.\, y)\ z =_\alpha (\lambda x.\, x)\ z}$$

where $(y\ y) \cdot y = () \cdot y = y$ and $(y\ x) \cdot x = y$, so it follows that $(\lambda y.\, y)\ z =_\alpha (\lambda x.\, x)\ z$. □

α-equivalence satisfies the following properties

- reflexivity $t =_\alpha t$ for any term t;
- symmetry $u =_\alpha t$ if $t =_\alpha u$;
- transitivity $t =_\alpha v$ if $t =_\alpha u$ and $u =_\alpha v$.

Easy to prove reflexivity and symmetry (try it!) but tricky to prove transitivity.

We are mainly in interested in terms up to α-equivalence, as the name of a bound variable does not matter. Hence, we consider λ-terms *modulo* α-equivalence, i.e.

$$[t]_\alpha = \{\, u \in \Lambda(V) \mid t =_\alpha u \,\}$$

as well as the (quotient) set:

$$\Lambda(V)/{=_\alpha} := \{\, [t]_\alpha \mid t \in \Lambda(V) \,\}.$$

Which of the following pairs are α-equivalent? If so, prove it.

1. x and y if $x \neq y$
2. $\lambda x\, y.\, y$ and $\lambda z\, y.\, y$
3. $\lambda x\, y.\, x$ and $\lambda y\, x.\, y$
4. $\lambda x\, y.\, x$ and $\lambda x\, y.\, y$

Challenge

Is it true that α-equivalent terms have the same de Bruijn representation?

Can you come up with a strategy to prove your conjecture?

Untyped λ-Calculus: Dynamics

The evaluation of λ-calculus is of this form

$$\boxed{\cdots \underbrace{(\lambda x.\, t)\, u}_{\beta\text{-redex}} \cdots} \longrightarrow_{\beta 1} \boxed{\cdots \qquad \underbrace{t\,[u/x]}_{\text{substitution of } N \text{ for } x \text{ in } M} \qquad \cdots}$$

In λ-calculus, defining substitution is subtle:

> *Variable x in u may be captured by an abstraction $\lambda x.\, t$, if the substitution $[u/x](\lambda x.\, t)$ is naively carried out.*

How to evaluate the following terms? Remember that we shall not discriminate α-variants.

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

A notion of the *scope* of a variable is needed to know which variable is available in scope to be substituted.

We use the notion of *free variable*: a variable y is free if $y \in \mathbf{FV}(t)$ where $\mathbf{FV}(t)$ is defined by

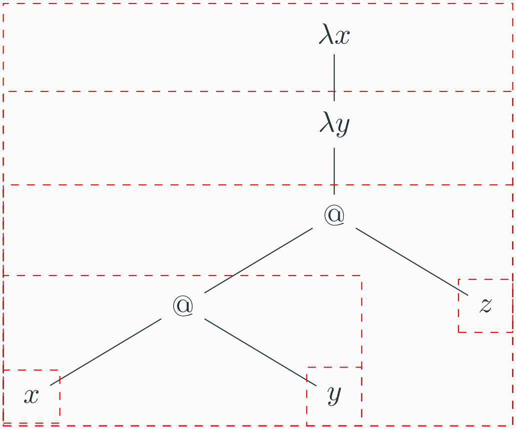
$$\begin{aligned} \mathbf{FV}(x) &= \{x\} \\ \mathbf{FV}(t\ u) &= \mathbf{FV}(t) \cup \mathbf{FV}(u) \\ \mathbf{FV}(\lambda x.\, t) &= \mathbf{FV}(t) - \{x\} \end{aligned}$$

A variable y is bound in t if it occurs in t but is not free.

Proposition 3

$\mathbf{FV}$ *respects* α*-equivalence, i.e. if* $t =_\alpha u$*, then* $\mathbf{FV}(t) = \mathbf{FV}(u)$.

Compute the set $\mathbf{FV}(t)$ of free variables for each subtree t of the following abstract syntax tree:



Given a term t and a variable x, the capture-avoiding substitution

$$\_[t/x] \colon \Lambda \to \Lambda$$

of t for x is defined on terms by

$$y[t/x] = \begin{cases} t & \text{if } x = y \\ y & \text{otherwise} \end{cases}$$

$$(t_1\ t_2)[t/x] = (t_1[t/x])\ (t_2[t/x])$$

$$(\lambda y.\, u)[t/x] = \begin{cases} \lambda y.\, (u[t/x]) & \text{if } x \neq y \text{ and } y \notin \mathbf{FV}(t) \\ ? & \text{otherwise} \end{cases}$$

If the clause $?$ is reached, then *rename* the bound variable y to some variable fresh for x and t, i.e. some z such that $z \neq y$ and $z \notin \mathbf{FV}(t)$, before proceeding.

A β-redex is a term of the form $(\lambda x.\, t)\ u$ where computation can be performed upon and the application can be reduced to $t[u/x]$.

Definition 4

The *one-step (full) β-reduction* is a relation between terms defined inductively by following rules:

$$\frac{}{(\lambda x.\, t)\ u \longrightarrow_\beta t[u/x]} \qquad \frac{t_1 \longrightarrow_\beta t_2}{t_1\ u \longrightarrow_\beta t_2\ u}$$

$$\frac{t_1 \longrightarrow_\beta t_2}{\lambda x.\, t_1 \longrightarrow_\beta \lambda x.\, t_2} \qquad \frac{u_1 \longrightarrow_\beta u_2}{t\ u_1 \longrightarrow_\beta t\ u_2}$$

For example, $(\boxed{(\lambda xy.\, x)\ t})\ u \longrightarrow_\beta \boxed{(\lambda y.\, t)\ u} \longrightarrow_\beta t[u/y]$.

Write down a sequence of β-reductions and *circle* all β-redexes while reducing a term:

1. $(\lambda x.\, x)\ z$
2. $((\lambda x.\, x)\ y)\ \ ((\lambda z.\, z)\ x)$
3. $\lambda n\, x\, y.\, n\ (\lambda z.\, y)\ x$
4. $(\lambda n\, x\, y.\, n\ (\lambda z.\, y)\ x)\ \ \lambda f\, x.\, x$

Multi-step full β-reduction

It is convenient to represents a sequence of β-reductions

$$t \longrightarrow_\beta t_1 \longrightarrow_\beta \ldots \longrightarrow_\beta u$$

by a single relation $t \longrightarrow\!\!\!\!\!\rightarrow_\beta u$.

Definition 5

The *multi-step (full) β-reduction* is a relation defined inductively by

$$\frac{}{t \twoheadrightarrow_\beta t} \text{ (0-step)}$$

$$\frac{t \longrightarrow_\beta u \qquad u \twoheadrightarrow_\beta v}{t \twoheadrightarrow_\beta v} \text{ ($n+1$-step)}$$

$t \longrightarrow\!\!\!\!\rightarrow_\beta u$ IS TRANSITIVE

Lemma 6

For every derivations of $t \twoheadrightarrow_\beta u$ and $u \twoheadrightarrow_\beta v$, there is a derivation of $t \twoheadrightarrow_\beta v$.

We often say "if $t \twoheadrightarrow_\beta u$ and $u \twoheadrightarrow_\beta v$ then $t \twoheadrightarrow_\beta v$" instead.

Proof.

By induction on the derivation d of $t \twoheadrightarrow_\beta u$:

1. If d is given by (0-step), then $t =_\alpha u$.
2. If d is given by (n+1-step), i.e. there is u' s.t. $t \longrightarrow_\beta u'$ and $u' \twoheadrightarrow_\beta u$.
   By induction hypothesis, every derivation $u' \twoheadrightarrow_\beta u$ gives rise to a derivation of $u' \twoheadrightarrow_\beta v$, so by (n+1-step) $t \twoheadrightarrow_\beta v$.

□

The reduction relation $t \longrightarrow_\beta u$ is directed, i.e. $t \longrightarrow_\beta u$ does not imply $u \longrightarrow_\beta t$. We may consider a notion of undirected equality based on β-reduction, while arguing the computational equality:

Definition 7

We say that t and u are β-equal, written $t =_\beta u$, if

$$\frac{t \longrightarrow_\beta u}{t =_\beta u}(\beta) \qquad \frac{}{t =_\beta t}(\text{refl}) \qquad \frac{t =_\beta u}{u =_\beta t}(\text{sym}) \qquad \frac{t =_\beta u \qquad u =_\beta v}{t =_\beta v}(\text{trans})$$

It is clear that $t \twoheadrightarrow_\beta u$ implies $t =_\beta u$ (why?). How about the converse?

SUMMARISE HERE ALL THE RELATIONS WE HAVE SEEN SO FAR.

Programming in λ-Calculus

Boolean and conditional can be encoded as combinators.

Boolean

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

Conditional

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

for any two λ-terms M and N.

Natural numbers as well as arithmetic operations can be encoded in untyped lambda calculus.

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

Successor

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

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

Addition

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

Conditional

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

Exercise

1. Define Boolean operations `not`, `and`, and `or`.
2. Evaluate `succ` $\mathbf{c}_0$ and `add` $\mathbf{c}_1$ $\mathbf{c}_2$.
3. Define the multiplication `mult` over Church numerals.

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

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

This cannot be done in λ-calculus directly. (Why?)

Note that unfolding sum as many times as it requires gives

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

The *Y combinator* is defined as a term

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

Proposition 8

$\mathbf{Y}$ *is a fixed-point operator, i.e.*

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

for every λ-term F. In particular, $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$.

Intuitively, $\mathbf{Y}F$ defines recursion where F describes each iteration.

Summation via Y

We encode the following recursion

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

by defining each iteration G with an additional function f so that $\texttt{sum} := \mathbf{Y}G$:

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

For example, letting $G' := ((\lambda x.\, G\ (x\ x))\ (\lambda x.\, G\ (x\ x)))$, we have

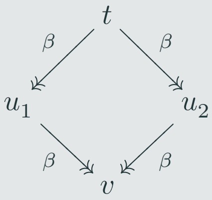
$$\begin{aligned} \texttt{sum}\ \mathbf{c}_1 &\longrightarrow_\beta G'\ \mathbf{c}_1 \\ &\longrightarrow_\beta G\ G'\ \mathbf{c}_1 \\ &\longrightarrow_\beta (\lambda n.\, \texttt{ifz}\ n\ \mathbf{c}_0\ (\texttt{add}\ n\ (G'\ (\texttt{pred}\ n))))\ \mathbf{c}_1 \\ &\longrightarrow_\beta \texttt{ifz}\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

1. Evaluate $\texttt{sum}\,\mathbf{c}_1$ to its normal form in detail.
2. Define the factorial $n!$ with Church numerals.

Homework

Theorem 9 (Church-Rosser)

Given u_1 *and* u_2 *with* $t \twoheadrightarrow_\beta u_1$ *and* $t \twoheadrightarrow_\beta u_2$*, there is* v *such that* $u_1 \twoheadrightarrow_\beta v$ *and* $u_2 \twoheadrightarrow_\beta v$*.*



1. (2.5%) Show that $t \twoheadrightarrow_\beta u$ implies $t =_\beta u$.
2. (2.5%) Show that if $t =_\beta u$ then there is a *confluent* term v of t and u, i.e. $t \twoheadrightarrow_\beta v$ and $u \twoheadrightarrow_\beta v$.

Appendix: Evaluation strategy

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 $\longrightarrow_\beta$.

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

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

Evaluation strategies II

the leftmost-outermost (*normal order*) strategy **reduces the leftmost outermost β-redex in a term first. For example,**

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

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

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

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

CBV versus CBN

Call-by-value strategy **rightmost-outermost but not under any abstraction**

Call-by-name strategy **leftmost-outermost but not under any abstraction**

Proposition 10 (Determinacy)

Each of evaluation strategies is deterministic, i.e. if $M \longrightarrow_\beta N_1$ ***and*** $M \longrightarrow_\beta N_2$ ***then*** $N_1 = N_2$.

Normalisation

Definition 11

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

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

Theorem 12

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

Appendix: Takahashi's Proof of Confluence

Takahashi's Proof of Confluence

Proving the Church-Rosser property (or confluence) can be quite tricky. This section presents a straightforward strategy based on a notion of complete development, which unfolds as many β-redexes as possible *statically*.

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

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

Confluence: Parallel reduction

Let $M \Longrightarrow_\beta N$ denote the *parallel reduction* defined by

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

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

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

Confluence: Properties of parallel reduction

Lemma 13

1. $M \Longrightarrow_\beta M$ *holds for any term* M,
2. $M \longrightarrow_\beta N$ *implies* $M \Longrightarrow_\beta N$, *and*
3. $M \Longrightarrow_\beta N$ *implies* $M \twoheadrightarrow_\beta N$.

In particular, $M \Longrightarrow_\beta^* N$ *if and only if* $M \twoheadrightarrow_\beta N$.

Lemma 14 (Substitution respects parallel reduction)

$M \Longrightarrow_\beta M'$ *and* $N \Longrightarrow_\beta N'$ *imply* $M[N/x] \Longrightarrow_\beta M'[N'/x]$.

Theorem 15 (Triangle property)

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

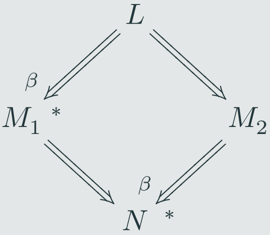
Proof sketch.

By induction on $M \Longrightarrow_\beta N$. $\square$

Strip Lemma

Theorem 16

If $L \Longrightarrow_\beta^* M_1$ and $L \Longrightarrow_\beta M_2$, then there exists N satisfying that $M_1 \Longrightarrow_\beta N$ and $M_2 \Longrightarrow_\beta^* N$, i.e.



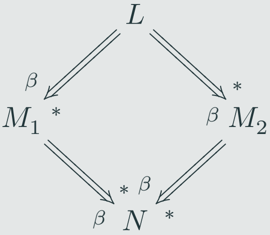
Proof sketch.

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

Confluence

Theorem 17

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



Corollary 18

The confluence of $\longrightarrow\!\!\!\!\!\!\twoheadrightarrow_\beta$ holds.