FLOLAC 2026

λ-Calculus

Untyped λ-Calculus, Part I

陳亮廷 Chen, Liang-Ting

Formosan Summer School on Logic, Language, and Computation 2026

Institute of Information Science
Academia Sinica

評分標準

分數比例　考試 (100%)

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`

Python `lambda x: M`

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 are introduced as $\lambda x.\, t$

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.

Why should we study λ-calculus?

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 the following topics in programming languages in a style of mathematical formalism.

1. How are programs 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 are programs 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: Static

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.

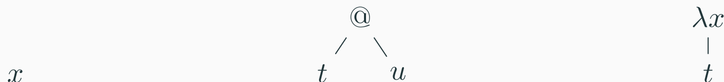
The set $\Lambda(V)$ of λ-terms over V is defined *inductively* as

variable $x \in \Lambda(V)$ if x is in V

application $t@u \in \Lambda(V)$ if $t, u \in \Lambda(V)$

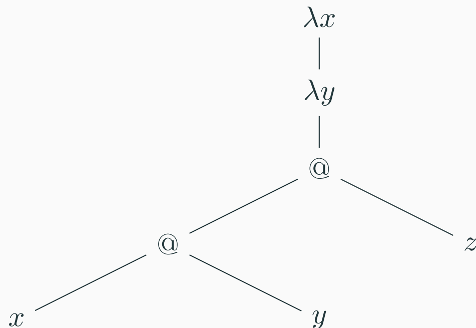
abstraction $\lambda x.\, t \in \Lambda(V)$ if $x \in V$ and $t \in \Lambda(V)$

Each construct is 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 $\lambda x.\,(\lambda y.\,((x@y)@z))$ is represented as



Important

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

The validity of the expression is 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.

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

1. $x \mathbin{@} (y \mathbin{@} z)$
2. $(x \mathbin{@} y) \mathbin{@} z$
3. $\lambda s.\, (\lambda z.\, (s \mathbin{@} z))$
4. $(\lambda x.\, x) \mathbin{@} (\lambda y.\, y)$
5. $\lambda a.\, (\lambda b.\, (a \mathbin{@} (\lambda c.\, (a \mathbin{@} b))))$

The set $\Lambda(V)$ of λ-terms over V can also be given as inference rules:

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

$$\frac{t \in \Lambda(V) \qquad u \in \Lambda(V)}{t@u \in \Lambda(V)} \text{ App}$$

$$\frac{t \in \Lambda(V)}{\lambda x.\, t \in \Lambda(V)} \text{ Abs, } x \in V$$

Assuming $x, y, z \in V$, the following derivation shows that $\lambda x.\,(\lambda y.\,((x@y)@z))$ is a well-formed lambda term over V.

$$\dfrac{\dfrac{\dfrac{\dfrac{x \in V}{x \in \Lambda(V)}\textsc{Var} \quad \dfrac{y \in V}{y \in \Lambda(V)}\textsc{Var}}{x@y \in \Lambda(V)}\textsc{App} \quad \dfrac{z \in V}{z \in \Lambda(V)}\textsc{Var}}{\dfrac{(x@y)@z \in \Lambda(V)}{\lambda y.\,((x@y)@z) \in \Lambda(V)}\textsc{Abs}}\textsc{App}}{\lambda x.\,(\lambda y.\,((x@y)@z)) \in \Lambda(V)}\textsc{Abs}$$

Assume that all variables appearing below are in V. Derive the following terms using their formation rules. That is, show that the following are valid λ-terms.

1. $x \mathbin{@} (y \mathbin{@} z)$
2. $(x \mathbin{@} y) \mathbin{@} z$
3. $\lambda s.\, (\lambda z.\, (s \mathbin{@} z))$
4. $(\lambda x.\, x) \mathbin{@} (\lambda y.\, y)$
5. $\lambda a.\, (\lambda b.\, (a \mathbin{@} (\lambda a.\, (a \mathbin{@} b))))$

Question

What's the difference between the AST representation and derivations? Identify the similarity between these two.

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. The convention allows us to write

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

where

1. multiple abstractions means a function with multiple arguments;
2. applying a function to multiple arguments is 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

$$M_1\ M_2\ M_3\ \dots\ M_n \equiv (\dots ((M_1 \mathbin{@} M_2) \mathbin{@} M_3) \mathbin{@} \dots) \mathbin{@} M_n$$

Function body extends as far right as possible

$$\lambda x.\, M\ N \quad \textbf{means} \quad \lambda x.\, (M\ N)\ \textbf{instead of}\ (\lambda x.\, M)\ N.$$

For example,

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

Apply the convention to rewrite the following terms:

1. $x \mathbin{@} (y \mathbin{@} z)$
2. $(x \mathbin{@} y) \mathbin{@} z$
3. $\lambda s.\, (\lambda z.\, (s \mathbin{@} z))$
4. $(\lambda x.\, x) \mathbin{@} (\lambda y.\, y)$
5. $\lambda a.\, (\lambda b.\, (a \mathbin{@} (\lambda a.\, (a \mathbin{@} b))))$

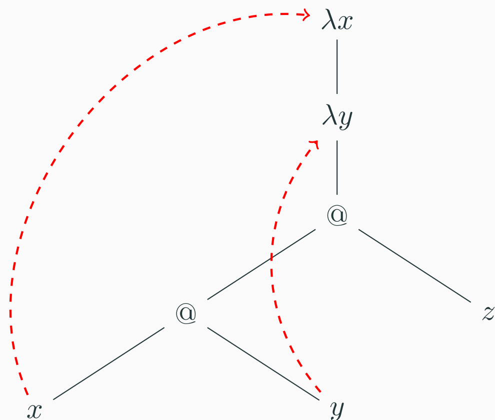
Let's discuss an important notion of syntax: *variable binding*.

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, the 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 is visualised in an abstract syntax tree:



The variable x at the leaf is bound by its nearest enclosing abstraction over x in the AST. Similarly for y.

Draw the binding for each variable in the AST of the following terms:

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

They are the same terms used previously, so you may just reuse your previous answers.

The point of naming a variable is to determine where it applies, so renaming variables should not alter its meaning.

For example, $f(x, y) = x^2 + y$ and $f(a, b) = a^2 + b$ represent the same function.

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

$$t =_\alpha u$$

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

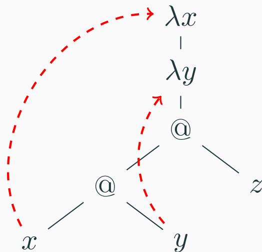
Question

How to define α-equivalence formally?

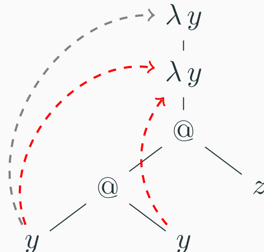
Idea

Using the named representation, define $t =_\alpha u$ if variables in t and u are renamed 'suitably' to exactly the same term.

After a naive renaming, a renamed variable might be captured by some λ accidentally breaking the original binding structure.



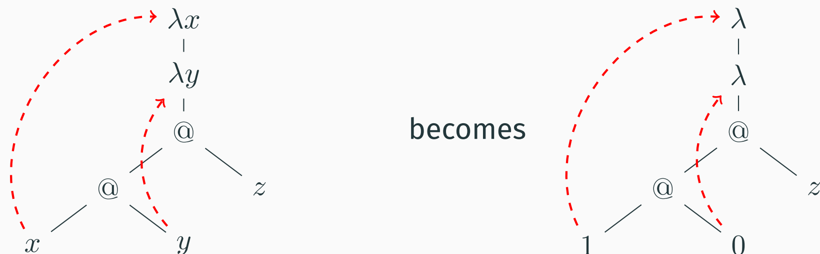
becomes



Idea

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

The index i' points to the i-th nearest λ-node from the variable (going upward):



Free variables are left named in this informal presentation.

Pros and cons of De Bruijn representation

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 Hard for humans to read.

It is a solution that defeats its own purpose: named variables are used to look up variables easily for humans!

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?

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

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

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

Compute $\mathbf{Var}(t)$ for the following terms t by definition step by step.

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

Idea

To rename a binder safely, we swap the binder with a fresh variable $z \notin \mathbf{Var}(t)$.

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 transposition $\pi = (y\,z)$ is defined by

$$\pi \cdot x = \pi\ x$$
$$\pi \cdot (t\ u) = (\pi \cdot t)\ (\pi \cdot u)$$
$$\pi \cdot (\lambda x.\, t) = \lambda(\pi\ x).\, (\pi \cdot t)$$

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

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

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

The third case is the interesting one: the fresh variable z serves as a common temporary name for the two binders.

Example 2

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

Proof.

By definition

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

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

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?

Exercise

Show that 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 $u =_\alpha v$.

That is, $=_\alpha$ is an equivalence relation.

We are mainly 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:

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

Untyped λ-Calculus: Dynamics

The evaluation of λ-calculus is of this form

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

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

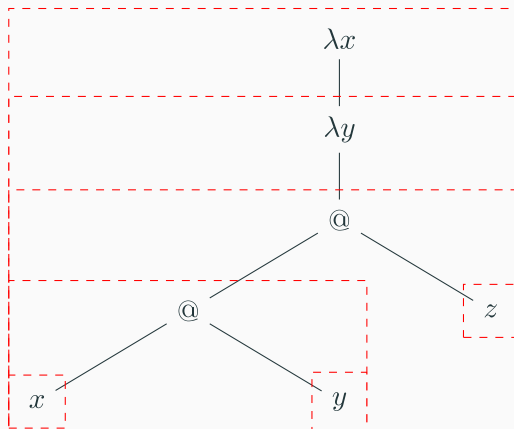
$$\begin{aligned} \mathbf{FV}(x) &= \{x\} \\ \mathbf{FV}(t\ u) &= \mathbf{FV}(t) \cup \mathbf{FV}(u) \\ \mathbf{FV}(\lambda x.\, t) &= \mathbf{FV}(t) - \{x\} \end{aligned}$$

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

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

of t for x is defined on terms by

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

$$(u\ v)[t/x] = (u[t/x])\ (v[t/x])$$

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

The last clause means that we *rename* the bound variable y to some variable fresh for x, t, and u before proceeding.

Compute the following by definition step by step.

- $(\lambda y.\, x)[t/x]$ if $x \neq y$ and $y \notin \mathbf{FV}(t)$.
- $(\lambda x\, y.\, n\ (\lambda z.\, y)\ x)\,[\lambda f\, x.\, x/n]$, assuming all variables are distinct.

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

Definition 4

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

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

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

For example, $(\boxed{(\lambda x\, y.\, x)\ t})\ u \longrightarrow_\beta (\lambda y.\, t)\ u \longrightarrow_\beta t$ assuming $y \notin \mathbf{FV}(t)$.

A term t is in normal form if $t \not\longrightarrow_\beta u$ for any u.

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

It is convenient to represent a sequence of β-reductions

$$t \longrightarrow_\beta t_1 \longrightarrow_\beta \dots \longrightarrow_\beta u$$

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

Definition 5

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

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

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

Lemma 6

For every pair of 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 $u = t$, so the given derivation of $u \twoheadrightarrow_\beta v$ is already a derivation of $t \twoheadrightarrow_\beta v$.
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 $t \longrightarrow\!\!\!\!\!\rightarrow_\beta u$ implies $t =_\beta u$ (why?). How about the converse?

Notation	Meaning
$x \in \Lambda(V)$	x is a variable term over V
$t@u \in \Lambda(V)$	application of t to u, with $t, u \in \Lambda(V)$
$t\ u$	conventional notation for $t@u$
$\lambda x.\, t \in \Lambda(V)$	abstraction, with $x \in V$ and $t \in \Lambda(V)$
$t =_\alpha u$	t and u have the same binding structure
$\mathbf{Var}(t)$	variables occurring in t
$\mathbf{FV}(t)$	free variables of t
$t[u/x]$	capture-avoiding substitution of u for x in t
$t \longrightarrow_\beta u$	one full β-reduction step
$t \twoheadrightarrow_\beta u$	zero or more full β-reduction steps
$t =_\beta u$	symmetric, reflexive, transitive closure of $\longrightarrow_\beta$