Lambda Calculus - Formal description (1A)

Young Won Lim
11/2/22

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Definition

Lambda expressions are composed of:

- **variables** v1, v2, ...;
- the **abstraction symbols λ** (lambda) and **.** (dot);
- **parentheses** ().

The set of lambda expressions, **Λ**, can be defined inductively:

- If x is a variable, then x ∈ Λ.
- If x is a variable and M ∈ Λ, then (λx.M) ∈ Λ.
- If M, N ∈ Λ, then (M N) ∈ Λ.

instances of rule 2 are known as **abstractions** (λx.M)

instances of rule 3 are known as **applications** **(M N)**

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Free and bound variables (1)

The **abstraction operator**, **λ**, is said to bind its **variable** wherever it occurs in the **body** of the **abstraction**.

Variables that fall within the scope of an **abstraction** are said to be bound.

In an expression **λx.M**,
the part **λx** is often called binder,
as a hint that the **variable x** is getting bound
by appending **λx** to **M**.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Free and bound variables (3)

All other variables (unbound) are called **free**.

For example, in the expression **λy.x x y**,

y is a bound **variable** and

x is a free **variable**.

Also a **variable** is bound by its *nearest* **abstraction**.

In **λx.y (λx.z x)**, the single occurrence of **x** in the expression is bound by the second **lambda**: .

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Free and bound variables (4)

The set of **free variables FV(M)** of a **lambda expression M**,

is defined by recursion on the structure of the **terms**, as follows:

FV(x) = {x}, where **x** is a **variable**

FV(λx.M) = FV(M) \ {x} **x** is a **bound variable**

FV(M N) = FV(M) ∪ FV(N)

An **expression** that contains no **free variables** is said to be *closed*.

Closed lambda expressions are also known as **combinators**

and are equivalent to **terms** in **combinatory logic**.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction (1)

The meaning of **lambda expressions** is defined
by how **expressions** can be reduced.[21]

There are three kinds of **reduction**:

- **α-conversion**: changing bound variables;
- **β-reduction**: applying functions to their arguments;
- **η-reduction**: which captures a notion of extensionality.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction (2)

two expressions are

α-equivalent,

if they can be α-converted into the same expression.

β-equivalent,

if they can be β-converted into the same expression.

η-equivalent,

if they can be η-converted into the same expression.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction (5)

The term **redex** (**reducible expression**),
refers to subterms that can be reduced by one of the reduction rules.

For example, **(λx.M) N** is a **β-redex**
in expressing the substitution of **N** for **x** in **M**.

The **expression** to which a **redex** reduces
is called its **reduct**; the **reduct** of **(λx.M) N** is **M[x := N]**.

If **x** is <u>not</u> free in **M**, **λx.M x** is also an **η-redex**, with a **reduct** of **M**.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

α-conversion (1)

α-conversion (**α-renaming**)

allows **bound variable** names to be changed.

For example, **α-conversion** of **λx.x** might yield **λy.y**.

terms that differ only by **α-conversion** are called **α-equivalent**.

Frequently, in uses of **lambda calculus**,

α-equivalent **terms** are considered to be equivalent.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

α-conversion (2)

The precise **rules** for **α-conversion** are not completely trivial.

First, when α-converting an **abstraction**,
the only **variable occurrences** that are renamed
are those that are bound to the same **abstraction**.

For example, an α-conversion of **λx.λx.x** could result in **λy.λx.x**,
but it could not result in **λy.λx.y**.

The latter has a different meaning from the original.
This is analogous to the programming notion of variable shadowing.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

α-conversion (3)

Second, **α-conversion** is <u>not</u> <u>possible</u>
if it would result in a **variable** getting <u>captured</u> by a <u>different</u> **abstraction**.

For example, if we replace **x** with **y** in **λx.λy.x**,
we get **λy.λy.y**, which is not at all the same.

In programming languages with **static scope**,
α-conversion can be used to make **name resolution** <u>simpler</u>
by ensuring that <u>no</u> **variable name** <u>masks</u> a **name**
in a **containing scope**
(see **α-renaming** to make **name resolution** <u>trivial</u>).

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

α-conversion (4)

In the **De Bruijn index notation**,

any two α-equivalent terms are syntactically identical.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Substitution (1)

Substitution, written **M[V := N]**,

is the process of replacing all **free occurrences**

of the **variable V** in the **expression M** with **expression N**.

Substitution on **terms** of the **lambda calculus**

is defined by recursion on the structure of **terms**,

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Substitution (1')

note: **x** and **y** are only **variables**

while **M** and **N** are any **lambda expression**

$x[x := N] = N$

$y[x := N] = y$, if $x \neq y$

$(M1\ M2)[x := N] \quad = M1[x := N]\ M2[x := N]$

$(\lambda x.M)[x := N] \quad = \lambda x.M$

$(\lambda y.M)[x := N] \quad = \lambda y.(M[x := N])$, if $x \neq y$ and $y \notin FV(N)$

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Substitution (2)

To substitute into an **abstraction**,

it is sometimes necessary to α-convert the **expression**.

For example, it is not correct for **(λx.y)[y := x]** to result in **λx.x**,

because the substituted **x** was supposed to be free

but ended up being bound.

$(\lambda y.M)[x := N] = \lambda y.(M[x := N])$, if $x \neq y$ and $y \notin FV(N)$

The correct substitution in this case is **λz.x**, up to α-equivalence.

Substitution is defined uniquely up to α-equivalence.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

β-reduction

β-reduction captures the idea of **function application**.

β-reduction is defined in terms of **substitution**:

the β-reduction of **(λV.M) N** is **M[V := N]**.

For example, assuming some encoding of 2, 7, ×,

we have the following β-reduction: **(λn.n × 2) 7** → **7 × 2**.

β-reduction can be seen to be the same

as the concept of **local reducibility** in **natural deduction**,

via the **Curry–Howard isomorphism**.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

η-reduction

η-reduction expresses the idea of **extensionality**,
which in this context is that two **functions** are the same
if and only if they give the same result for all arguments.

η-reduction converts between **λx.f x** and **f**
whenever **x** does not appear free in **f**.

η-reduction can be seen to be the same as
the concept of **local completeness** in **natural deduction**,
via the **Curry–Howard isomorphism**.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Normal form and confluence (1)

For the **untyped lambda calculus**,
β-reduction as a **rewriting rule** is
neither strongly normalising
nor weakly normalising.

However, it can be shown that **β-reduction** is confluent
when working up to **α-conversion**
(i.e. we consider two **normal forms** to be equal
if it is possible to **α-convert** one into the other).

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Normal form and confluence (2)

Therefore, both strongly normalising terms
and weakly normalising terms have a unique normal form.

For strongly normalising terms,
any reduction strategy is guaranteed to yield the normal form,

whereas for weakly normalising terms,
some reduction strategies may fail to find the normal form.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (1)

Whether a term is normalising or not,

and how much work needs to be done in normalising it if it is,

depends to a large extent on the reduction strategy used.

Common reduction strategies include:

- **Normal order**
- **Applicative order**
- **Full β-reductions**

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (2)

Common reduction strategies include:

- **Normal order**

  The leftmost, outermost **redex** is always reduced first.

  That is, whenever possible the **arguments** are

  substituted into the **body** of an **abstraction**

  before the **arguments** are reduced.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (3)

Common reduction strategies include:

- **Applicative order**

  The leftmost, innermost **redex** is always reduced first.

  Intuitively this means a function's **arguments**

  are always reduced before the **function** itself.

  Applicative order always attempts to apply **functions**

  to **normal forms**, even when this is not possible.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (4)

Common reduction strategies include:

- **Full β-reductions**

  Any **redex** can be reduced at any time.

  This means essentially the lack of

  any particular reduction strategy

  — with regard to reducibility, "all bets are off".

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (5)

Weak reduction strategies do not reduce under lambda abstractions:

- **Call by value**
- **Call by name**

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (6)

Weak reduction strategies do not reduce under lambda abstractions:

- **Call by value**

  A **redex** is reduced only when its right hand side has reduced to a **value** (**variable** or **abstraction**).

  Only the outermost **redexes** are reduced.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (7)

Weak reduction strategies do not reduce under lambda abstractions:

- **Call by name**

  As normal order, but no reductions

  are performed inside **abstractions**.

  For example, **λx.(λy.y)x** is in normal form

  according to this strategy, although it contains the **redex (λy.y)x**.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (8)

Strategies with sharing reduce computations

that are "the same" in parallel:

- **Optimal reduction**
- **Call by need**

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (9)

Strategies with sharing reduce computations
that are "the same" in parallel:

- **Optimal reduction**
  As normal order, but computations
  that have the same label are reduced simultaneously.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Reduction strategies (10)

Strategies with sharing reduce computations
that are "the same" in parallel:

- **Call by need**

  As call by name (hence weak), but **function applications** that would duplicate **terms** instead name the **argument**, which is then reduced only "when it is needed".

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf