Lambda Calculus - Informal description (1A)

Young Won Lim
9/1/22

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

This document was produced by using LibreOffice.

- **CFG for the Lambda Calculus**
- Function Abstraction
- Function Application
- Free and Bound Variables
- Beta Reductions
- Evaluating a Lambda Expression
- Currying
- Renaming Bound Variables by Alpha Reduction
- Eta Conversion
- Substitutions
- Disambiguating Lambda Expressions
- Normal Form
- Evaluation Strategies

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

CFG for Lambda Calculus (1)

The central concept in the **lambda calculus** is
an **expression** which we can think of as a program
that returns a result when evaluated
consisting of *another* **lambda calculus expression**.

Here is the grammar for lambda expressions:

expr → λ variable . expr | expr expr | variable | (expr) | constant

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

CFG for Lambda Calculus (2)

expr → λ variable . expr | expr expr | variable | (expr) | constant

A **variable** is an identifier.

A **constant** is a built-in function such as *addition* or *multiplication*,

or a constant such as an *integer* or *boolean*.

all **programming language constructs**

can be represented as **functions**

with the pure **lambda calculus**

so these **constants** are unnecessary.

However, some constants may be used for notational simplicity.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

- CFG for the Lambda Calculus
- **Function Abstraction**
- Function Application
- Free and Bound Variables
- Beta Reductions
- Evaluating a Lambda Expression
- Currying
- Renaming Bound Variables by Alpha Reduction
- Eta Conversion
- Substitutions
- Disambiguating Lambda Expressions
- Normal Form
- Evaluation Strategies

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Function Abstraction (1)

A **function abstraction**, often called a **lambda abstraction**,
is a **lambda expression** that defines a **function**.

A **function abstraction** consists of *four parts*:

a **lambda** followed by a **variable**, a **period**,
and then an **expression** as in **λx.expr**.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Function Abstraction (2)

For example, the function abstraction **λx. + x 1**

defines a **function of x** that *adds* **x** to **1**.

Parentheses can be added to lambda expressions for clarity.

Thus, we could have written this function abstraction

as **λx.(+ x 1)** or even as **(λx. (+ x 1))**.

In C this function definition might be written as

```
int addOne (int x) {
  return (x + 1);    }
```

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Function Abstraction (3)

Note that unlike C

the **lambda abstraction** does not give a **name** to the **function**.

The **lambda expression** itself is the **function**.

We say that **λx.expr** binds the **variable x** in **expr** and

that **expr** is the **scope** of the **variable**.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

- CFG for the Lambda Calculus
- Function Abstraction
- **Function Application**
- Free and Bound Variables
- Beta Reductions
- Evaluating a Lambda Expression
- Currying
- Renaming Bound Variables by Alpha Reduction
- Eta Conversion
- Substitutions
- Disambiguating Lambda Expressions
- Normal Form
- Evaluation Strategies

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Function Application (1)

A **function application**, often called a **lambda application**,
consists of an **expression** followed by an **expression**:

expr1 expr2.

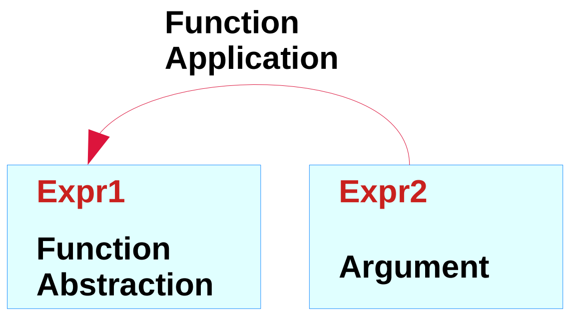
The first **expression expr1** is

a **function abstraction**

the second **expression expr2** is

the **argument** to which the **function** is applied.

All **functions** in **lambda calculus** have exactly one **argument**.
Multiple-argument **functions** are represented by **currying**,



http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Function Application (2)

the **lambda expression λx. (+ x 1) 2**

is an application of the **function λx. (+ x 1)** to the argument **2**.

This function application **λx. (+ x 1) 2** can be evaluated

by substituting the **argument 2** for the **formal parameter x**

in the **body (+ x 1)**.

Doing this we get **(+ 2 1)**.

This substitution is called a **beta reduction**.

Expr1	Expr2
Function Abstraction	**Argument**

λx. (+ x 1) 2

(+ 2 1)

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Function Application (3)

Beta reductions are like macro substitutions in **C**.

To do **beta reductions** correctly, we may need
to rename **bound variables** in **lambda expressions**
to avoid name clashes.

function application associates left-to-right; thus,

f g h = (f g) h

function application binds more tightly than **λ**; thus,

λx. f g x = (λx. (f g) x).

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Function Application (4)

Functions in the lambda calculus are first-class citizens;

functions can be used as **arguments** to functions

functions can return **functions** as **results**.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

- CFG for the Lambda Calculus
- Function Abstraction
- Function Application
- **Free and Bound Variables**
- Beta Reductions
- Evaluating a Lambda Expression
- Currying
- Renaming Bound Variables by Alpha Reduction
- Eta Conversion
- Substitutions
- Disambiguating Lambda Expressions
- Normal Form
- Evaluation Strategies

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Free and bound variables (1)

In the function definition **λx.x**

the **variable x** in the **body** of the definition (the second **x**)

is **bound** because its first occurrence in the definition is **λx**.

A **variable** that is not bound in **expr**

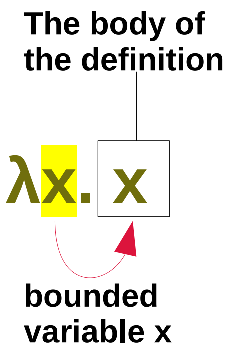
is said to be **free** in **expr**.

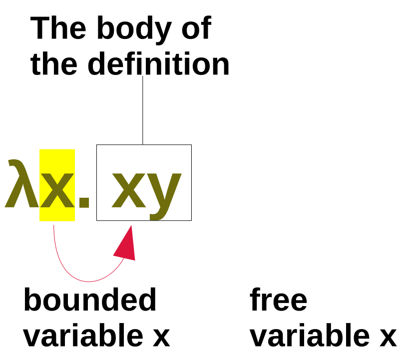
In the function **(λx.xy)**,

in the **body** of the **function**

the **variable x** is **bound**

the **variable y** is **free**.





http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Free and bound variables (2)

Every **variable** in a **lambda expression** is

either **bound** or **free**.

Bound and **free variables** have

quite a different status in functions.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Free and bound variables (3)

In the expression **(λx.x)(λy.yx)**:

in the **body** of the leftmost expression

the **variable x** is **bound** to the first lambda.

in the **body** of the second expression

the **variable y** is **bound** to the second lambda.

the **variable x** is **free**

independent of the **x** in the first expression.

The first lambda **The second lambda**

(λx. x) (λy. yx)

variable x bounded to the 1st labmda **variable y bounded to the 2nd labmda** **variable x free**

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Free and bound variables (4)

In the expression **(λx.xy)(λy.y)**:

in the body of the leftmost expression

the **variable y** is **free**.

in the body of the second expression

the **variable y** is **bound** to the second lambda.

The first lambda

The second lambda

(λx. xy) (λy. y)

variable x bounded to the 1st labmda

variable y bounded to the 2nd labmda

variable y free

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Free and bound variables (5)

Given an **expression e**, the following rules define **FV(e)**,

the set of **free variables** in **e**:

If **e** is a **variable x**, then **FV(e) = {x}**.

If **e** is of the form **λx.y**, then **FV(e) = FV(y) - {x}**.

If **e** is of the form **xy**, then **FV(e) = FV(x)** ∪ **FV(y)**.

An expression with <u>no</u> **free** variables is said to be **closed**.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

- CFG for the Lambda Calculus
- Function Abstraction
- Function Application
- Free and Bound Variables
- Beta Reductions
- Evaluating a Lambda Expression
- **Currying**
- Renaming Bound Variables by Alpha Reduction
- Eta Conversion
- Substitutions
- Disambiguating Lambda Expressions
- Normal Form
- Evaluation Strategies

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Currying (1)

All **functions** in the **lambda calculus** are

prefix and

take *exactly* one argument.

If we want to apply a **function** to more than one argument,

we can use a technique called **currying**

that treats a **function** applied to more than one argument

to a sequence of applications of one-argument **functions**.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Currying (2)

For example, to express the sum of 1 and 2

we can write (+ 1 2) as ((+ 1) 2)

the expression **(+ 1)** denotes the **function**

that adds 1 to its **argument**.

Thus ((+ 1) 2) means

the **function +** is applied to the **argument** 1

the **result** is a **function** **(+ 1)** that adds 1 to its **argument**:

(+ 1 2) = ((+ 1) 2) → 3

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Currying (3)

In lambda calculus, <u>each</u> **input** is <u>preceded</u> by a **λ** symbol.

A function can have <u>more than one</u> **input**.

Curring a **function** of <u>two</u> **inputs** <u>transforms</u> that function

into a **function** with <u>one</u> **input**

by <u>passing</u> <u>one</u> of the **inputs** into it.

currying turns **f(x,y)** to **g(y)**

g is **f** with **x** passed into it.

g only takes one input, **y**.

f(x,y) = x + y if **x = 3** then **f(3,y) = 3 + y … g(y)**

https://functional.works-hub.com/learn/higher-order-functions-lambda-calculus-currying-maps-6e539

Currying (4)

Similarly in lambda calculus:

λx.λy.(x+y) 3 y

= (λx.(λy.(x+y)) 3) y

= (λy.(3+y)) y

= λy.(3+y) y

https://functional.works-hub.com/learn/higher-order-functions-lambda-calculus-currying-maps-6e539

Currying (5)

One can curry recursively, and turn a function of any number of input to a function of that number of input minus one.

(λx.λy.λz.(x+y+z) 3) 4 5

= (λy.λz.(3+y+z)) 4 5

= (λz.(3+4+z)) 5

= (3+4+5)

https://functional.works-hub.com/learn/higher-order-functions-lambda-calculus-currying-maps-6e539

Currying (6)

a **function g** with multiple **arguments**, eg) **g 3 4**

this allows us to define a multi-argument **function**

as a **function** that returns a **function**:

define **g** as **$\lambda x.\lambda y.(x^2 + y^2)^{1/2}$**

This says that **g** takes a **parameter x** and returns a **function**

that takes a **parameter y** and returns **$(x^2 + y^2)^{1/2}$**.

g 3 4 $\Rightarrow$ **$(\lambda x.\lambda y.(x^2 + y^2)^{1/2})$ 3 4**

$\Rightarrow$ **$(\lambda y.(3^2 + y^2)^{1/2})$ 4**

$\Rightarrow$ **$(3^2 + 4^2)^{1/2} = 5$**

http://www.cburch.com/books/lambda/

Currying (7)

g 1 is the **function**

that takes a **parameter y** and returns **$(1 + y^2)^{½}$**.

define **h** to be **g 1**,

which would refer to the function **$\lambda y.(1 + y^2)^{½}$**.

The lambda calculus is particularly useful

when we want to talk about **functions**

whose **parameters** are **functions** or which return **functions**.

http://www.cburch.com/books/lambda/

Currying (8)

For example, suppose we define **s** according to the following:

s = λh.λz.h (h z).

Here, **s** takes a function **f** as a **parameter** and returns the **function** that returns the result of applying **f** twice to its argument.

Thus, if **f** is **λx.x + 1**, then we can try to determine what function **s f** represents:

http://www.cburch.com/books/lambda/

Currying (8)

s f ⇒ **(λh.λz.h (h z)) (λx.x + 1)**

⇒ **λz.(λx.x + 1) ((λx.x + 1) z)**

⇒ **λz.(λx.x + 1) ((λx.x + 1) z)**

⇒ **λz.(λx.x + 1) (z + 1)**

⇒ **λz.((λx.x + 1) (z + 1))**

⇒ **λz.(z + 1) + 1**

⇒ **λz.z + 2**

Thus, **s f** is a function that returns two more than its argument.

http://www.cburch.com/books/lambda/

- CFG for the Lambda Calculus
- Function Abstraction
- Function Application
- Free and Bound Variables
- Beta Reductions
- Evaluating a Lambda Expression
- Currying
- **Renaming Bound Variables by Alpha Reduction**
- Eta Conversion
- Substitutions
- Disambiguating Lambda Expressions
- Normal Form
- Evaluation Strategies

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Alpha reduction (1)

The name of a **formal parameter**

in a **function definition** is *arbitrary*.

We can use *any* **variable** to *name* a **parameter**,

so that the function **λx.x** is equivalent to **λy.y** and **λz.z**.

This kind of renaming is called **alpha reduction**.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Alpha reduction (2)

Note that we <u>cannot</u> <u>rename</u>

free variables in expressions.

Also note that we <u>cannot</u> <u>change</u>

the <u>name</u> of a **bound variable** in an expression

to conflict with the <u>name</u> of a **free variable** in that expression.

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Alpha reduction (3)

formal parameters are only names:

they are correct if they are consistent.

(λx . (λx . + (− x 1)) x 3) 9

↔ (λx . (λy . + (− y 1)) x 3) 9

→ ((λy . + (− y 1)) 9 3)

→ (((λy . + (− y 1)) 9) 3)

→ (+ (− 9 1) 3)

→ (+ 8 3)

→ 11

http://www.cs.columbia.edu/~aho/cs4115/Lectures/2014_EdwardsLC.pdf

Alpha reduction (4)

You've probably done this before in C or Java:

```
int add(int x, int y)
{
        return x + y;
}
```

⟺

```
int add(int a, int b)
{
        return a + b;
}
```

http://www.cs.columbia.edu/~aho/cs4115/Lectures/2014_EdwardsLC.pdf

- CFG for the Lambda Calculus
- Function Abstraction
- Function Application
- Free and Bound Variables
- Beta Reductions
- Evaluating a Lambda Expression
- Currying
- Renaming Bound Variables by Alpha Reduction
- Eta Conversion
- **Substitutions**
- Disambiguating Lambda Expressions
- Normal Form
- Evaluation Strategies

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Substitution (1)

For a **beta reduction**, we introduced the notation **[f/x]e**
to indicate that the **expression f** is to be substituted
for all free occurrences of the **formal parameter x** in the **expression e**:

(λx.e) f → **[f/x]e**

f for **x** in **e**

f **expression**
x **formal parameter**
e **expression**

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Substitution (2)

To avoid name clashes in a **substitution [f/x]e**,
first rename the **bound variables** in **e** and **f**
so they become distinct.
Then perform the textual substituion of **f** for **x** in **e**.

For example, consider the substitution **[y(λx.x)/x] λy.(λx.x)yx**.

After renaming all the **bound variables**
to make them all distinct we get **[y(λu.u)/x] λv.(λw.w)vx**.

Then doing the substitution we get **λv.(λw.w)v(y(λu.u))**.

In the first expression

x → u

In the second expression

y → v

x → w

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Substitution (3)

The rules for substitution are as follows.

assume **x** and **y** are distinct **variables**, and **e**, **f** and **g** are **expressions**.

Substitution rules for **variables**

[e/x]x = e

[e/x]y = y

Substitution rules for **function applications**

[e/x](f g) = ([e/x]f) ([e/x]g)

Substitution rules for **function abstractions**

[e/x](λx.f) = λx.f

[e/x](λy.f) = λy.[e/x]f

provided **y** is not free in **e** (this is called the "freshness" condition).

λx.e f ⟹ **[f/x]e**

f for **x** in **e**

f **expression**

x **formal parameter**

e **expression**

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Substitution (3’)

assume **x** and **y** are distinct **variables**

For **variables**

[e/x]x = e	**x** $\leftarrow$ **e**
[e/x]y = y	**y** is a **variable**
	<u>cannot</u> contain **x**
	<u>no</u> substitution

$$\lambda x.e\ f \Rightarrow [f/x]e$$

e for **x** in **f**

e expression

x formal parameter

f expression

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Substitution (3')

assume **e**, **f** and **g** are **expressions**.

For **function applications**

[e/x](f g) = ([e/x]f) ([e/x]g)

f and **g** are **expressions**

can contain the **formal parameter e**

λx.e f → **[f/x]e**

e for **x** in **f**

[e/x](f g)

([e/x]f) ([e/x]g)

distribution of the **substitution**

over the **expressions f** and **g**

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Substitution (3'')

assume **x** and **y** are distinct **variables**, and **e**, **f** and **g** are **expressions**.

Substitution rules for **function abstractions**

[e/x](λx.f) = λx.f

x in the **expression f** is a **formal parameter** (bounded)

can be renamed by an **alpha reduction**

then the **expression f** does not have **x** for a **substitution**

[e/x](λy.f) = λy.[e/x]f

provided **y** is not free in **e** (this is called the "**freshness**" condition).

λx.e f → **[f/x]e**

e for **x** in **f**

[e/x](λx.f)

(λx.f)

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Substitution (3'')

assume **x** and **y** are distinct **variables**, and **e**, **f** and **g** are **expressions**.

Substitution rules for **function abstractions**

[e/x](λx.f) = λx.f

[e/x](λy.f) = λy.[e/x]f

provided **y** is <u>not</u> free in **e** (this is called the "**freshness**" condition).

the **expression f** may contain **variable x**

the **expression e** may contain **variable y**

λx.e f → **[f/x]e**

e for **x** in **f**

[e/x](λx.f)

[e/x](λy.f)

(λy.[e/x]f)

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

Substitution (4)

Examples:

[y/y](λx.x) = λx.x

[y/x]((λx.y) x)
= ([y/x](λx.y)) ([y/x]x)
= (λx.y)y

Note that the freshness condition does <u>not</u> allow us
to make the substitution **[y/x](λy.x) = λy.([y/x]x) = λy.y**
because **y** is free in the **expression y**.
(λy.x) **[y/x]**

Substitution rules
for **function abstractions**

[e/x](λx.f) = λx.f

[e/x](λy.f) = λy.[e/x]f
provided **y** is <u>not</u> free in **e**
"**freshness**" condition).

the **expression f**
may contain **variable x**
the **expression e**
may contain **variable y**

http://www.cs.columbia.edu/~aho/cs4115/Lectures/15-04-13.html

References

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

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