Monad P3 : Lambda Calculus (1F)

Young Won Lim
7/2/22

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

This document was produced by using LibreOffice.

Lambda Calculus

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

Lambda calculus (3) – function abstraction

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

Lambda calculus (4) – function abstraction

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

Lambda calculus (4) – function abstraction

the **function abstraction λx. + x 1**

C function definition

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

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**

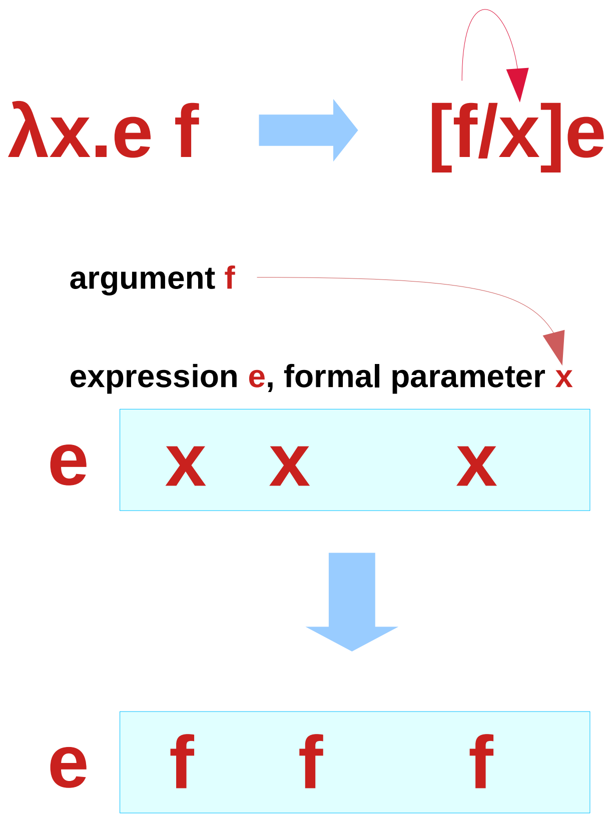
and that **expr** is the **scope** of the **variable**.

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

Lambda calculus (5) – function application

A **function application λx.e f** is evaluated
by substituting the **argument f**
for *all free occurrences* of the **formal parameter x**
in the body **e** of the **function definition**.

We will use the notation **[f/x]e** to indicate
that **f** is to be substituted for all free occurrences of **x**
in the expression **e**.



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

Lambda calculus (5) – free and bound variables

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), the variable x in the body of the function

is bound and the variable y is free.

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

Lambda calculus (5) – free and bound variables

In the expression (λx.x)(λy.yx):

- The variable x in the body of the leftmost expression is bound to the first lambda.
- The variable y in the body of the second expression is bound to the second lambda.
- The variable x in the body of the second expression is free.
- Note that x in second expression is independent of the x in the first expression.

In the expression (λx.xy)(λy.y):

- The variable y in the body of the leftmost expression is free.
- The variable y in the body of the second expression is bound to the second lambda.

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

Lambda calculus (5) – free and bound variables

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 no free variables is said to be closed.

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

Lambda calculus (6) – beta reduction

Examples:

(λx.x)y → [y/x]x = y
in the express **x**,
substitute the parameter **x** with the argument **x**

(λx.xzx)y → [y/x]xzx = yzy
in the express **xzx**,
substitute the parameter **x** with the argument **y**

(λx.z)y → [y/x]z = z
in the express **z**,
substitute the parameter **x** with the argument **y**

since the formal parameter **x** does not appear in the body **z**.

This **substitution** in a **function application** is called
a **beta reduction** and we use a right arrow to indicate it.

λx.e f → [f/x]e

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

Lambda calculus (7) – beta reduction

If **expr1** → **expr2**, we say **expr1** reduces to **expr2** in one step.

In general, **(λx.e)f** → **[f/x]e** means that

applying the **function (λx.e)** to the **argument expression f**
reduces to the **expression [f/x]e**
where the **argument expression f** is substituted
for the function's **formal parameter x** in the **function body e**.

λx.e f ➡ **[f/x]e**

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

Lambda calculus (8) – beta reduction

A **lambda calculus expression** (aka a "**program**") is
"run" by *computing a final result*
by repeatly applying **beta reductions**.

We use →* to denote the reflexive and transitive closure of →;
that is, zero or more applications of **beta reductions**.

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

Lambda calculus (9) – beta reduction

Examples:

(λx.x)y → y

illustrating that **λx.x** is the **identity function**

(λx.xx)(λy.y) → (λy.y)(λy.y) → (λy.y);

thus, we can write **(λx.xx)(λy.y) →* (λy.y)**.

we have applied a **function** to a **function**

as an argument and the **result** is a **function**.

λx.e f ➡ **[f/x]e**

(λx.xx) (λy.y) — **function argument**

(λy.y)(λy.y)

(λy.y)(λy.y) — **indentity function**

(λy.y)

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

Lambda calculus (10) – beta reduction

Examples:

(λx.x)y → y

illustrating that **λx.x** is the **identity function**

(λx.xx)(λy.y) → (λy.y)(λy.y) → (λy.y);

thus, we can write **(λx.xx)(λy.y) →* (λy.y)**.

→* to denote the reflexive and transitive closure of →

that is, zero or more applications of beta reductions

Transitive relation

x R y and **y R z** then **x R z**

Reflexive relation

x R x

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

Evaluation models of a function

Call-by-value:

arguments are evaluated before a function is entered

Call-by-name:

arguments are passed unevaluated

Call-by-need:

arguments are passed unevaluated
but an expression is only evaluated once
and shared upon subsequent references

http://dev.stephendiehl.com/fun/005_evaluation.html

Comparisons

Call by name is non-memoizing non-strict evaluation strategy
where the **value**(s) of the **argument**(s) need only be found
when actually used inside the **function's body**, each time anew:

Call by need is memoizing non-strict a.k.a. lazy evaluation strategy
where the **value**(s) of the **argument**(s) need only be found
when used inside the **function's body** for the first time,
and then are available for any further reference:

Call by value is strict evaluation strategy
where the **value**(s) of the **argument**(s) must be found
before entering the function's body:

https://stackoverflow.com/questions/61601125/haskell-semantics-call-by-name-value

Comparisons

Call by name	non-memoizing	non-strict
Call by need	memoizing	non-strict
Call by value		strict

https://stackoverflow.com/questions/61601125/haskell-semantics-call-by-name-value

Comparisons

Call by name the **value**(s) of the **argument**(s) need only be found when actually used inside the **function's body**, each time anew:	non-memoizing	non-strict
Call by need the **value**(s) of the **argument**(s) need only be found when used inside the **function's body** for the first time, and then are available for any further reference:	memoizing	non-strict
Call by value the **value**(s) of the **argument**(s) must be found before entering the function's body:		strict

https://stackoverflow.com/questions/61601125/haskell-semantics-call-by-name-value

Memoization / Sharing

Memoization is a technique
for storing **values** of a **function**
instead of recomputing them
each time the **function** is called.

Sharing means that **temporary data** is physically stored,
if it is used multiple times.

https://wiki.haskell.org/Memoization

Strictness

Strict evaluation, or eager evaluation, is an evaluation strategy where **expressions** are evaluated as soon as they are bound to a **variable**.

when **x = 3 * 7** is read, **3 * 7** is immediately computed and **21** is bound to **x**.

Conversely, with **lazy evaluation** **values** are only computed when they are needed.

In the example **x = 3 * 7**, **3 * 7** isn't evaluated until it's needed, like if you needed to output the value of **x**.

https://en.wikibooks.org/wiki/Haskell/Strictness https://wiki.haskell.org/Sharing

Laziness

Haskell is a **non-strict** language, and most implementations use a strategy called **laziness** to run your program.
Basically **laziness == non-strictness + sharing**.

Laziness can be a useful tool for improving performance, but more often than not it reduces performance by adding a **constant overhead** to everything.

https://wiki.haskell.org/Performance/Strictness

Laziness

Because of **laziness**, the compiler can't evaluate a function **argument** and pass the **value** to the function,

it has to record the **expression** in the **heap** in a **suspension** (or **thunk**) in case it is evaluated later.

Storing and evaluating **suspensions** is costly, and unnecessary if the **expression** was going to be evaluated anyway.

https://wiki.haskell.org/Performance/Strictness

Call by **name**

```
h x = x : (h x)
g xs = [head xs, head xs - 1]

g (h 2)  = let {xs = (h 2)} in [head xs, head xs - 1]
         = [let {xs = (h 2)} in head xs,      let {xs = (h 2)} in head xs - 1]
         = [head (h 2),                       let {xs = (h 2)} in head xs - 1]
         = [head (let {x = 2} in x : (h x)}), let {xs = (h 2)} in head xs - 1]
         = [let {x = 2} in x,                 let {xs = (h 2)} in head xs - 1]
         = [2,                                let {xs = (h 2)} in head xs - 1]
         = ....
```

https://stackoverflow.com/questions/61601125/haskell-semantics-call-by-name-value

Call by **need**

```
h x = x : (h x)
g xs = [head xs, head xs - 1]

g (h 2)  = let {xs = (h 2)}           in [head xs, head xs - 1]
         = let {xs = (2 : (h 2))}     in [head xs, head xs - 1]
         = let {xs = (2 : (h 2))}     in [2,       head xs - 1]
       = ....
```

https://stackoverflow.com/questions/61601125/haskell-semantics-call-by-name-value

Call by **value**

```
h x = x : (h x)
g xs = [head xs, head xs - 1]

g (h 2)  = let {xs = (h 2)}                   in [head xs, head xs - 1]
         = let {xs = (2 : (h 2))}             in [head xs, head xs - 1]
         = let {xs = (2 : (2 : (h 2)))}       in [head xs, head xs - 1]
         = let {xs = (2 : (2 : (2 : (h 2))))} in [head xs, head xs - 1]
         = ....
```

All the above assuming g (h 2) is entered at the GHCi prompt
and thus needs to be printed in full by it.

https://stackoverflow.com/questions/61601125/haskell-semantics-call-by-name-value

Reductions in the expression **f x**

Given an **expression f x**

Call-by-value: Evaluate **x** to **v**
Evaluate **f** to **λy.e**
Evaluate **[y/v]e**

Call-by-name: Evaluate **f** to **λy.e**
Evaluate **[y/x]e**

Call-by-need: Allocate a thunk v for **x**
Evaluate **f** to **λy.e**
Evaluate **[y/v]e**

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by **value** (1)

Call by value is an extremely common evaluation model.
Many programming languages both imperative and functional
use this evaluation strategy.

The essence of **call-by-value** is that
there are two categories of expressions: **terms** and **values**.

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by **value** (2)

Values are **lambda expressions** and other **terms**
which are in **normal form** and cannot be reduced further.

All **arguments** to a **function** will be reduced to **normal form**
before they are bound inside the lambda and
reduction only proceeds once the **arguments** are reduced.

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by **value** (3)

For a simple arithmetic expression, the reduction proceeds as follows. Notice how the subexpression (2 + 2) is evaluated to normal form before being bound.

(\x. \y. y x) (2 + 2) (\x. x + 1)

=> (\x. \y. y x) 4 (\x. x + 1)

=> (\y. y 4) (\x. x + 1)

=> (\x. x + 1) 4

=> 4 + 1

=> 5

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by **name** (1)

In **call-by-name** evaluation,

the **arguments** to lambda expressions are substituted as is,

evaluation simply proceeds from left to right

substituting the outermost lambda or reducing a value.

If a substituted expression is not used it is never evaluated.

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by **name** (2)

For example, the same expression we looked at for **call-by-value**
has the same normal form but arrives at it
by a different sequence of reductions:

(\x. \y. y x) (2 + 2) (\x. x + 1)
=> (\y. y (2 + 2)) (\x. x + 1)
=> (\x. x + 1) (2 + 2)
=> (2 + 2) + 1
=> 4 + 1
=> 5

Call-by-name is non-strict, although very few languages use this model.

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by **need** (1)

Call-by-need is a special type of non-strict evaluation
in which unevaluated **expressions** are represented
by **suspensions** or **thunks** which are passed
into a **function** unevaluated and
only evaluated when needed or forced.

When the **thunk** is forced
the **representation** of the **thunk** is updated
with the computed **value**
and is not recomputed upon further reference.

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by **need** (2)

The **thunks** for unevaluated lambda expressions
are allocated when evaluated,
and the resulting computed **value**
is placed in the same **reference**
so that subsequent **computations** share the result.

If the **argument** is never needed
it is never computed,
which results in a trade-off
between **space** and **time**.

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by **need** (3)

Since the evaluation of subexpression
does not follow any pre-defined order,
any impure functions with side-effects
will be evaluated in an unspecified order.

As a result call-by-need can only effectively
be implemented in a purely functional setting.

http://dev.stephendiehl.com/fun/005_evaluation.html

Call by value (3)

For a simple arithmetic expression,

the reduction proceeds as follows.

Notice how the subexpression (2 + 2) is evaluated

to **normal form** before being bound.

(\x. \y. y x) (2 + 2) (\x. x + 1)

=> **(\x. \y. y x) 4 (\x. x + 1)**

=> **(\y. y 4) (\x. x + 1)**

=> **(\x. x + 1) 4**

=> **4 + 1**

=> *5*

http://dev.stephendiehl.com/fun/005_evaluation.html

References

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

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