Lambda Calculus (4A) – Normal forms

Young Won Lim
9/5/22

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

This document was produced by using LibreOffice.

Normal Form (2)

The expression **(λx.x x)(λx.x x)** does not have a **normal form**

because it always evaluates to itself.

(λx.x x)(λx.x x)

(λx.x x) (λx.x x)

We can think of this expression

as a representation for an **infinite loop**.

The expression **(λx. λy. y)((λz.z z)(λz.z z))**

can be reduced to the normal form **λy.y**.

(λx. λy. y)((λz.z z)(λz.z z))

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

Normal Form (2)

Q: If a lambda expression does have a normal form, do all choices of reduction sequences get there?

(λx.λy. y)((λz.z z)(λz.z z))

A: No. Consider the following lambda expression:

(λx.λy.y)((λz.zz)(λz.zz))

This lambda expression contains two redexes: the first is the whole expression (the application of (λx.λy.y) to its argument); the second is the argument itself: ((λz.zz)(λz.zz)). The second redex is the one we used above to illustrate a lambda expression with no normal form; each time you beta-reduce it, you get the same expression back. Clearly, if we keep choosing that redex to reduce we're never going to find a normal form for the whole expression. However, if we reduce the first redex we get: λy.y, which is in normal form. Therefore, the sequence of choices that we make can determine whether or not we get to a normal form.

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/1.LAMBDA-CALCULUS.html

Normal Form (3)

(**Church-Rosser Theorem**)

Suppose an **expression A** can be reduced

by a sequence of **reductions** to an **expression B**,

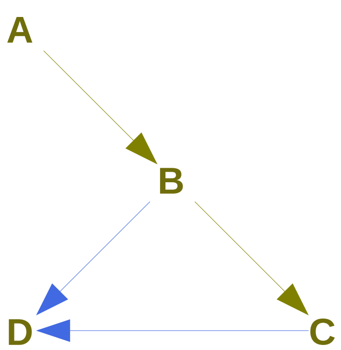
and it can be reduced by *another* sequence of **reductions**

to *another* **expression C**.

Then there exists some **expression D**

that can be reached from a sequence of **reductions** from **B**

and also from a sequence of **reductions** from **C**.



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

Normal Form (4)

Essentially, this theorem says that

no **reduction** will ever be a wrong turn.

As long as we can find a **reduction** to perform,

then it will still be possible to reach

whatever destination somebody else can find.

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

Normal Form (5)

We call an **expression** **irreducible**

if there are no **reductions**

that can be performed on the **expressions**,

such as **1** or **λx.x** or **λf.f (λy.y)**,

but not **(λy.y) f**, which can be reduced to **f**.

An **irreducible expression** is sometimes

said to be in **normal form**.

Not counting **α-reductions** as reductions here

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

Normal Form (7)

Not all **expressions** can be reduced to **irreducible form**.

One of the simplest is **(λx.x x) (λx.x x)**

An application of beta-reduction to **(λx.x x) (λx.x x)**
simply returns us to the same expression we already have.

Even worse is the expression
(λx.x x x) (λx.x x x),

which will get longer each time we try to reduce it.

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

Normal Form (8)

The **Church-Rosser Theorem** implies

that there cannot be *two different*

irreducible forms of an **expression**.

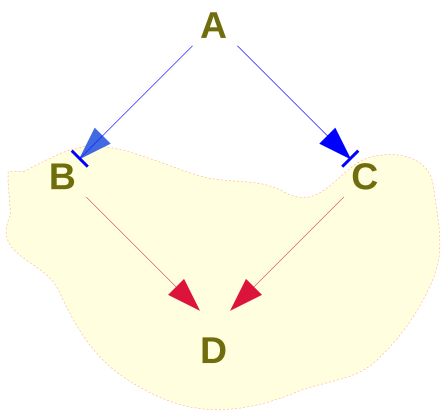
After all, if **A** could be reduced to *two distinct*

irreducible forms, **B** and **C**,

then the theorem says we would be able

to reduce both **B** and **C**,

and so they are actually not irreducible.



Contradiction!

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

Normal Form (9)

A natural question to ask is: Is there a technique for always reaching irreducible form when it exists? One important evaluation order is eager evaluation (or sometimes applicative order of evaluation or strict evaluation), in which an argument is always reduced before it is applied to a function. This is the ordering used in most programming languages, where we evaluate the value of an argument before passing it into a function.

$(\lambda x.x + 1)\ ((\lambda y.2 \times y)\ 3) \quad \Rightarrow \quad (\lambda x.x + 1)\ (2 \times 3) \Rightarrow (\lambda x.x + 1)\ 6$

$\quad \Rightarrow \quad 6 + 1 \Rightarrow 7$

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

Normal Form (10)

Unfortunately, eager evaluation does not always reach irreducible form when it exists. Consider the expression

(λx.1) ((λx.x x) (λx.x x)).

Using eager evaluation, we would first try to reduce the argument, but that simply reduces to itself. (Before trying to reduce (λx.x x) (λx.x x), though, we'd first have to examine the argument, λx.x x. In this case, though, there are no reductions to perform.) Yet this expression can reduce to irreducible form, for if we apply the argument to λx.1 immediately, we would reach 1 without needing to reduce the argument at any time. Eager evaluation, though, would never get us there.

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

Normal Form (11)

Alternatively, lazy evaluation order (sometimes called the normal order of evaluation) has us always pass an argument into a function unsimplified, only reducing the argument when needed.

(λx.x + 1) ((λy.2 × y) 3) ⇒ ((λy.2 × y) 3) + 1

⇒ (2 × 3) + 1 ⇒ 6 + 1 ⇒ 7

It turns out, mathematicians have proven that lazy evaluation does guarantee that we reach irreducible form when possible.

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

Normal Form (12)

If an expression can be reduced to an irreducible expression, then lazy evaluation order will reach it.

Due to this theorem, this evaluation order is sometimes called normal order (since an irreducible expression is said to be in normal form).

(Technically, we'll subtly distinguish the terms lazy evaluation and normal evaluation, as described in Section 2.1.)

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

Evaluation Strategies (1)

An **evaluation strategy** specifies the order

in which **beta reductions** for a **lambda expression** are made.

Some **reduction** orders for a lambda expression

may yield a **normal form**

while other orders *may not*.

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

Evaluation Strategies (2)

For example, consider the given expression

(λx.1)((λx.x x)(λx.x x))

This expression has two **redexes**:

The *entire* **expression** is a **redex**

in which we can apply the **function (λx.1)**

to the **argument ((λx.x x)(λx.x x))**

to yield the **normal form** 1.

this **redex** is the leftmost outermost **redex**

in the given expression.

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

Evaluation Strategies (3)

The **subexpression ((λx.x x)(λx.x x))** is also a **redex**

in which we can apply the **function (λx.x x)**

to the **argument (λx.x x)**.

Note that this **redex** is the leftmost innermost **redex**

in the given expression.

But if we evaluate this redex we get same **subexpression**:

(λx.x x)(λx.x x) → (λx.x x)(λx.x x).

Thus, continuing to evaluate the leftmost innermost **redex**

will not terminate and no **normal form** will result.

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

Evaluation Strategies (4)

There are two common reduction orders for **lambda expressions**:

normal order evaluation and

applicative order evaluation.

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

Evaluation Strategies (5)

Normal order evaluation

we always reduce the leftmost outermost redex at each step.

The first reduction order above is a normal order evaluation.

a remarkable property of lambda calculus is
that every **lambda expression** has a unique **normal form**
if one exists.
Moreover, if an expression has a **normal form**,
then normal order evaluation will always find it.

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

Evaluation Strategies (6)

Applicative order evaluation

we always reduce the leftmost innermost redex at each step.

The second reduction order above is
an applicative order evaluation.

rhus, even though an expression may have a **normal form**,
applicative order evaluation may fail to find it.

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 **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