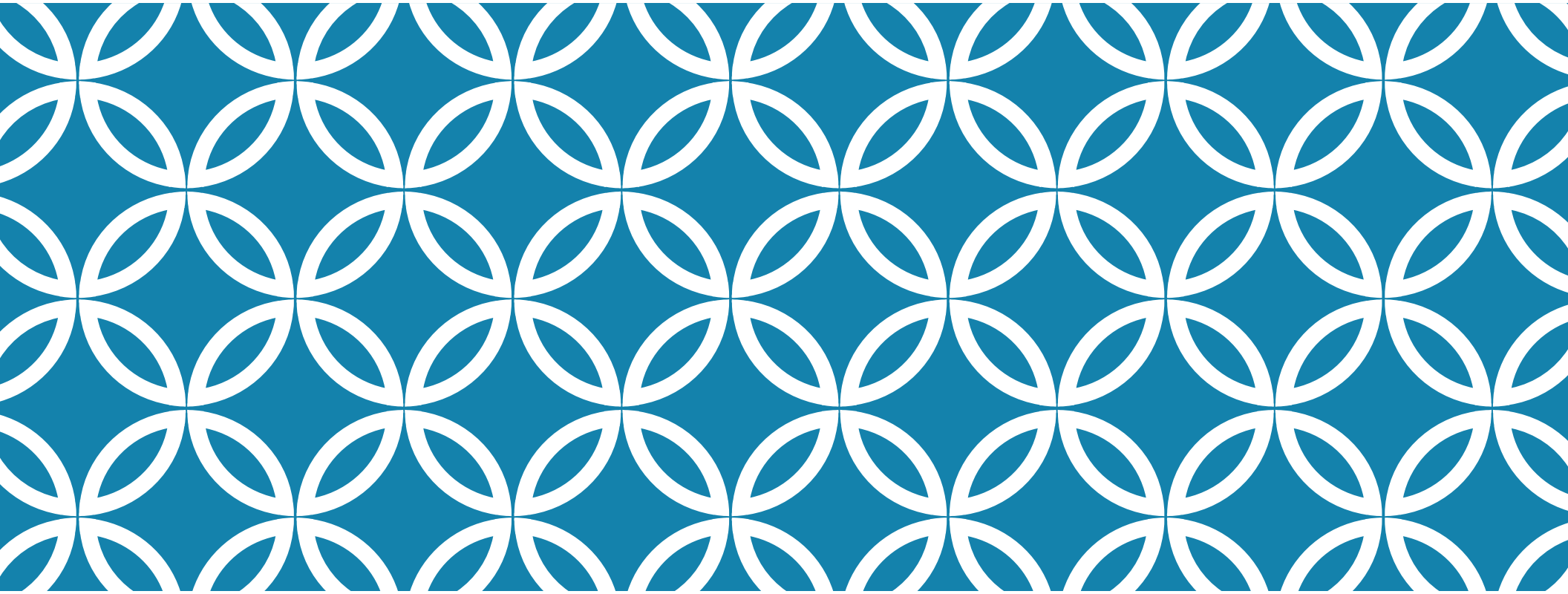


LAMBDA CALCULUS: AN INTRODUCTION

Chandreyee Chowdhury

OUTLINE

Why study lambda calculus?

Lambda calculus

- Syntax
- Evaluation
- Relationship to programming languages

Mary had a little lambda,
a function pure as snow.
And for every program that Mary wrote,
the lambda was all she needed to know.

λ is actually one of the most powerful & elegant abstractions in the history of computer science

https://pyvideo.org/europython-2017/mary-had-a-little-lambda.html

WHY STUDY LAMBDA CALCULUS?

"*Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.*"

(Landin 1966)

LAMBDA CALCULUS

- A framework developed in 1930s by Alonzo Church to study computations with functions
  - Church wanted a minimal notation to expose only what is essential
- The smallest universal programming language of the world
  - Universal-Any computable function can be expressed and evaluated

https://www.cs.cmu.edu/~fp/courses/15814-f18/lectures/Notes-15814-f18.pdf

BACKGROUND

Godel defined the class of *general recursive functions as the smallest set of* functions

- all the constant functions, the successor function, and closed under certain operations (such as compositions and recursion)

A function is computable (in the intuitive sense) if and only if it is general recursive

Church defined an idealized programming language called the *lambda calculus,*

- *a function is computable (in the intuitive sense) if* and only if it can be written as a lambda term

THE CONJECTURE

Church's Thesis : The effectively computable functions on the positive integers are precisely those functions definable in the pure lambda calculus (and computable by Turing machines).

The conjecture cannot be proved since the informal notion of "effectively computable function" is not defined precisely.

But since all methods developed for computing functions have been proved to be no more powerful than the lambda calculus, it captures the idea of computable functions

WHY STUDY λ-CALCULUS

We will see a number of important concepts in their simplest possible form, which means we can discuss them in full detail

We will then reuse these notions frequently to build the different code blocks.

The λ-calculus is of great historical and foundational significance.

The independent and nearly simultaneous development of Turing Machines and the λ -Calculus as universal computational mechanisms led to the Church-Turing Thesis

The notion of function is the most basic abstraction present in nearly all programming languages.

If we are to study programming languages, we therefore must strive to understand the notion of function.

FUNCTION CREATION

$f(x) = x + 5$

$f = \lambda x.\ x + 5$

Church introduced the notation

$$\lambda x.\ E$$

to denote a function with formal argument x and with body E

Functions do not have names

- names are not essential for the computation

Functions have a single argument

- Only one argument functions are discussed

FUNCTION APPLICATION

$f(x) = x + 5$ $\quad f(10)$

$f = \lambda x.\ x + 5$ $\quad f\ 10$

$(\lambda x.\ x + 5)\ 10$

The only thing that we can do with a function is to apply it to an argument

Church used the notation

$E_1\ E_2$

to denote the application of function E_1 to actual argument E_2

E_1 is called (ope)rator and E_2 is called (ope)rand

All functions are applied to a single argument

SIGNIFICANCE OF λ-CALCULUS

λ-calculus is the standard testbed for studying programming language features

- Because of its minimality
- Despite its syntactic simplicity the λ-calculus can easily encode:
  - numbers, recursive data types, modules, imperative features, exceptions, etc.

Certain language features necessitate more substantial extensions to λ-calculus:

- for distributed & parallel languages: π-calculus
- for object oriented languages: σ-calculus

The central concept in λ calculus is the "expression". A "name", also called a "variable", is an identifier which, for our purposes, can be any of the letters $a, b, c, \ldots$ An expression is defined recursively as follows:

<expression>	:=	<name> | <function> | <application>
<function>	:=	λ <name>.<expression>
<application>	:=	<expression><expression>

Variables x

Expressions e ::= $\lambda x.\ x \mid e \mid e_1\ e_2$

EXAMPLES OF LAMBDA EXPRESSIONS

The identity function:

$$I =_{def} \lambda x.\ x$$

A function that given an argument y discards it and computes the identity function:

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

A function that given a function f invokes it on the identity function

$$\lambda f.\ f\ (\lambda\ x.\ x)$$

NOTATIONAL CONVENTIONS

Application associates to the left

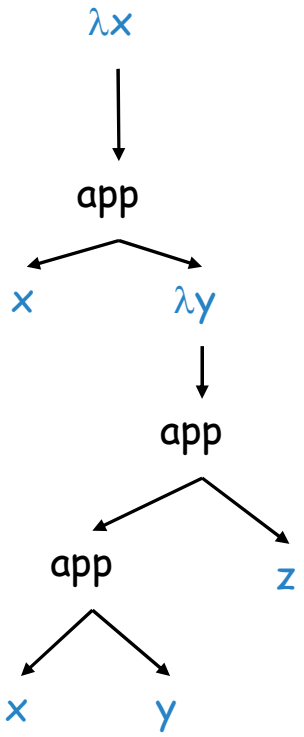
x y z parses as (x y) z

Abstraction extends to the right as far as possible

λx. x λy. x y z parses as

λ x. (x (λy. ((x y) z)))

And yields the the parse tree:



SCOPE OF VARIABLES

As in all languages with variables, it is important to discuss the notion of scope

- Recall: the scope of an identifier is the portion of a program where the identifier is accessible

An abstraction $\lambda x.\ E$ binds variable x in E

- x is the newly introduced variable
- E is the scope of x
- we say x is bound in $\lambda x.\ E$
- Just like formal function arguments are bound in the function body

FREE AND BOUND VARIABLES

$\int_0^1 x^2\,dx$

$\sum_{x=1}^{10} \frac{1}{x}$

$\lim_{x\to\infty} e^{-x}$

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

A variable is said to be free in E if it is not bound in E

Free variables are declared outside the term

We can define the free variables of an expression E recursively as follows:

Free(x) = { x}

Free(E_1 E_2) = Free(E_1) ∪ Free(E_2)

Free(λx. E) = Free(E) - { x }

Example: Free(λx. x (λy. x y z)) = { ? }

$$M \equiv (\lambda x.xy)(\lambda y.yz).$$

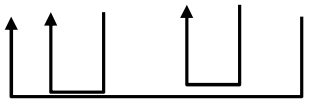
A lambda expression with no free variables is called closed.

FREE AND BOUND VARIABLES (CONT.)

Just like in any language with static nested scoping, we have to worry about variable shadowing

- An occurrence of a variable might refer to different things in different context

In λ-calculus: λx. x (λx. x) x



RENAMING BOUND VARIABLES

Two λ-terms that can be obtained from each other by a renaming of the bound variables are considered identical

Example: λx. x is identical to λy. y and to λz. z

Intuition:

- by changing the name of a formal argument and of all its occurrences in the function body, the behavior of the function does not change
- in λ-calculus such functions are considered identical

RENAMING BOUND VARIABLES (CONT.)

Convention: we will always rename bound variables so that they are all unique

- e.g., write λ x. x (λ y.y) x instead of λ x. x (λ x.x) x
- Variable capture or name clash problem would arise!

This makes it easy to see the scope of bindings

And also prevents serious confusion !

SUBSTITUTION

The substitution of E' for x in E (written [E'/x]E)

- Step 1. Rename bound variables in E and E' so they are unique (α-reduction)
- Step 2. Perform the textual substitution of E' for x in E

This is called β-reduction

We write $E \rightarrow_\beta E'$ to say that E' is obtained from E in one β-reduction step

We write $E \rightarrow^*_\beta E'$ if there are zero or more steps

(λx.x)

```
int f(int x){
  return x+10;
}
```

```
x=>x;
```

```
f(5);
```

(λx.x)(5)

E1= λx.x E2=5

(E1)(E2)

FUNCTIONS WITH MULTIPLE ARGUMENTS

Consider that we extend the calculus with the add primitive operation

The λ-term $\lambda x.\ \lambda y.$ add x y can be used to add two arguments E_1 and E_2:

$(\lambda x.\ \lambda y.$ add x y$)\ E_1\ E_2 \rightarrow_\beta$

$([E_1/x]\ \lambda y.$ add x y$)\ E_2 =$

$(\lambda y.$ add E_1 y$)\ E_2 \rightarrow_\beta$

$[E_2/y]$ add E_1 y $=$ add $E_1\ E_2$

The arguments are passed one at a time

$((\lambda x.((\lambda y.(x\ y))x))(\lambda z.w))$

((λa.a) λb.λc.b)(x) λe.f

(λb.λc.b)(x) λe.f

(λc.x)λe.f

(λf.((λg.((f f)g))(λh.(k h))))(λx.(λy.y)))

I= λx.x

```
Var I = x=>x;
Alert(I("Hi"));
```

fI=(λf.f)(λx.x)

ENCODING NATURAL NUMBERS IN LAMBDA CALCULUS

What can we do with a natural number?

- we can iterate a number of times

A natural number is a function that given an operation f and a starting value s, applies f a number of times to s:

$1 =_{def} \lambda f.\ \lambda s.\ f\ (s)$

$2 =_{def} \lambda f.\ \lambda s.\ f\ (f\ s)$

and so on

$0 =_{def} \lambda f.\ \lambda s.\ s$

ANYNUMBER=(ΛN. ΛF. ΛX.N(F(X)))

anyNumber One

=(λn. λf. λx.n(f(x)))((λg. λs.g(s))

=(λn. λf. λx.n(f(x)))((λg. λs.g(s))

=λf. λx. (f(x))

HIGHER ORDER FUNCTIONS

What is the result of $(\lambda x.\ \lambda y.\ add\ x\ y)\ E$?

- It is $\lambda y.\ add\ E\ y$

  (A function that given a value E' for y will compute add E E')

The function $\lambda x.\ \lambda y.\ E$ when applied to one argument E' computes the function $\lambda y.\ [E'/x]E$

This is one example of higher-order computation

- We write a function whose result is another function

ANYNUMBER=(ΛN. ΛF. ΛX.N(F(X)))

λn. n ((λf. f+1)(0))

Number= n=>n(i=>i+1)(0)

Successor Two

$=(\lambda n.\ \lambda f.\ \lambda x.f(n(f(x))))(\lambda f.\ \lambda x.f(f(x)))$

$=\lambda f.\ \lambda x.\ f(\lambda g.\ \lambda s.g(s))(f(f(x)))$

$=\lambda f.\ \lambda x.\ f(\lambda g.\ \lambda s.g(s))(f(f(x)))$

$=\lambda f.\ \lambda x.\ f(\lambda s.f(f(s))(x)$

$=\lambda f.\ \lambda x.\ f(f(f(x)))$

ANY NATURAL NUMBER

anyNumber = λf. λs. f s

def λn. λf. λs. n(f (s))

def ((λf. f+1)(0))

λn. λf. n ((f+1)(0))

Number= n=>(i=>i+1)(0)

COMPUTING WITH NATURAL NUMBERS

$$1 \equiv \lambda sz.s(z)$$

$$2 = \lambda sz.s(s(z))$$

```
var anyNumber = n => s=> z => n(s)(z);
```

The successor function

successor n $=_{def}$ λn.λf.λx.f(nfx)

Successor of 0 (S0) is $_{def}$ (λnfx.f(nfx)) (λsz.z)

$$\lambda yx.y((\lambda sz.z)yx) = \lambda yx.y((\lambda z.z)x) = \lambda yx.y(x) \equiv 1$$

Successor of 1 (S1) is $_{def}$ (λwyx.y(wyx)) (λsz.s(z))

Addition

$_{def}$ λm.λn.λf.λx.m(f)(n(f)(x))

2S3 is $_{def}$ (λsz.s(s(z)))(λwyx.y(wyx))(λuv.u(u(u(v))))

EVALUATION AND THE STATIC SCOPE

The definition of substitution guarantees that evaluation respects static scoping:

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

(y remains free, i.e., defined externally)

$(\lambda\ x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \rightarrow (\lambda\ x.\ (\lambda z.\ z\ x))\ (y\ (\lambda v.\ v)) \rightarrow_\beta \lambda z.\ z\ (y\ (\lambda v.\ v))$

If we forget to rename the bound y:

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

(y was free before but is bound now)

Definition: α-reduction

If v and w are variables and E is a lambda expression,

$$\lambda v \,.\, E \Rightarrow_{\alpha} \lambda w \,.\, E[v \rightarrow w]$$

provided that w does not occur at all in E, which makes the substitution $E[v \rightarrow w]$ safe. The equivalence of expressions under α-reduction is what makes part g) of the definition of substitution correct.

Definition: β-reduction

If v is a variable and E and E_1 are lambda expressions,

$$(\lambda v \,.\, E)\ E_1 \Rightarrow_{\beta} E[v \rightarrow E_1]$$

provided that the substitution $E[v \rightarrow E_1]$ is carried out according to the rules for a safe substitution.

ß- redex

BOUNDING OF VARIABLES

anyNumber one

$=(\lambda n.\ \lambda f.\ \lambda x.n(f(x)))((\lambda f.\ \lambda x.f(x))$

$=\lambda f.\ \lambda x.\ \lambda f.\ \lambda x.f(x)(f(x)))$

REDUCTION STRATEGIES

Definition : A lambda expression is in **normal for m** if it contains no β-redexes (and no δ-rules in an applied lambda calculus), so that it cannot be further reduced using the β-rule or the δ-rule. An expression in normal form has no more function applications to evaluate. ∎

Can every lambda expression be reduced to a normal form?

Is there more than one way to reduce a particular lambda expression?

If there is more than one reduction strategy, does each one lead to the same normal form expression?

Is there a reduction strategy that will guarantee that a normal form expression will be produced?

CAN EVERY LAMBDA EXPRESSION BE REDUCED TO A NORMAL FORM?

The identity function:

$(\lambda x.\ x)\ E \rightarrow [E\ /\ x]\ x = E$

Another example with the identity:

$(\lambda f.\ f\ (\lambda x.\ x))$

$(\lambda f.\ f\ (\lambda x.\ x))\ (\lambda x.\ x) \rightarrow$

$(\lambda f.\ f\ (\lambda y.\ y))\ (\lambda x.\ x) \rightarrow$

$(\lambda x.\ x)\ (\lambda y.\ y) \rightarrow$

$[\lambda y.\ y\ /x]\ x = \lambda y.\ y$

REDUCTION STRATEGIES

Can every lambda expression be reduced to a normal form?

Is there more than one way to reduce a particular lambda expression?

If there is more than one reduction strategy, does each one lead to the same normal form expression?

Is there a reduction strategy that will guarantee that a normal form expression will be produced?

IS THERE MORE THAN ONE WAY TO REDUCE A PARTICULAR LAMBDA EXPRESSION?

Example : $(\lambda y\ .\ 5)\ ((\lambda x\ .\ x\ x)\ (\lambda x\ .\ x\ x))$

Definition : A **normal order** reduction always reduces the leftmost outermost β-redex (or δ-redex) first. An **applicative order** reduction always reduces the leftmost innermost β-redex (or δ-redex) first. ∎

REDUCTION STRATEGIES

Can every lambda expression be reduced to a normal form?

Is there more than one way to reduce a particular lambda expression?

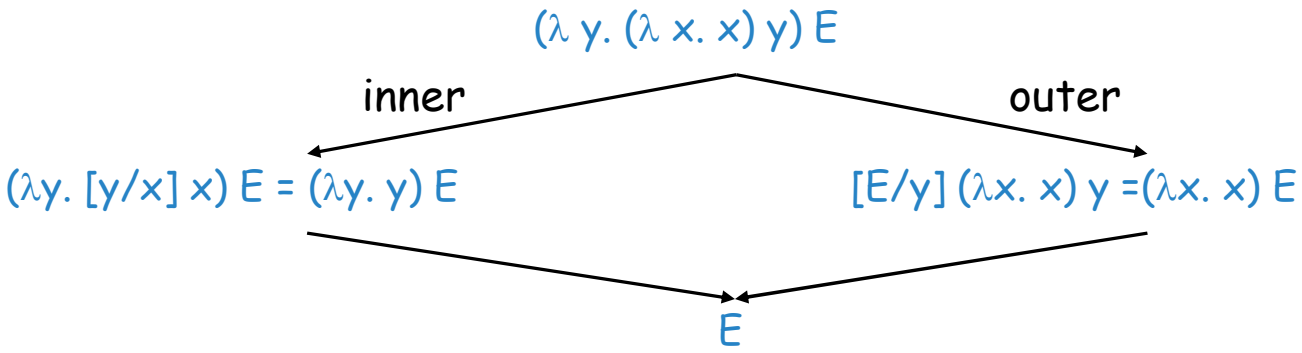
If there is more than one reduction strategy, does each one lead to the same normal form expression?

Is there a reduction strategy that will guarantee that a normal form expression will be produced?

IF THERE IS MORE THAN ONE REDUCTION STRATEGY, DOES EACH ONE LEAD TO THE SAME NORMAL FORM EXPRESSION?

$(\lambda\ y.\ (\lambda\ x.\ x)\ y)\ E$

- could reduce the inner or the outer λ
- which one should we pick?



ORDER OF EVALUATION (CONT.)

The Church-Rosser theorem says that any order will compute the same result

- A result is a λ-term that cannot be reduced further

Church-Rosser Theorem I: For any lambda expressions E, F, and G, if E $\Rightarrow^*$ F and E $\Rightarrow^*$ G, there is a lambda expression Z such that F $\Rightarrow^*$ Z and G $\Rightarrow^*$ Z.

Corollary: For any lambda expressions E, M, and N, if E $\Rightarrow^*$ M and E $\Rightarrow^*$ N where M and N are in normal form, M and N are variants of each other (equivalent with respect to α-reduction).

Some evaluations may terminate while others may diverge. If two evaluations terminate, it will be to the same normal form.

DIAMOND PROPERTY (confluence)

A normal order reduction can have either of the following outcomes

- 1. It reaches a unique (up to α-conversion) normal form lambda expression
- 2. It never terminates

Church-Rosser Theorem II: For any lambda expressions E and N, if $E \Rightarrow^* N$ where N is in normal form, there is a normal order reduction from E to N.

Unfortunately, there is no algorithmic way to determine for an arbitrary lambda expression which of these two outcomes will occur.

THE CONJECTURE

- **Church's Thesis :** The effectively computable functions on the positive integers are precisely those functions definable in the pure lambda calculus (and computable by Turing machines).

- The conjecture cannot be proved since the informal notion of "effectively computable function" is not defined precisely.

- But since all methods developed for computing functions have been proved to be no more powerful than the lambda calculus, it captures the idea of computable functions

REDUCTION STRATEGIES

- Can every lambda expression be reduced to a normal form?
- Is there more than one way to reduce a particular lambda expression?
- If there is more than one reduction strategy, does each one lead to the same normal form expression?
- Is there a reduction strategy that will guarantee that a normal form expression will be produced?

TURING MACHINE

•Turing machines are abstract machines designed in the 1930s by Alan Turing to model computable functions. It has been shown that the lambda calculus is equivalent to Turing machines in the sense that every lambda expression has an equivalent function defined by some Turing machine and vice versa.

IS THERE A REDUCTION STRATEGY THAT WILL GUARANTEE THAT A NORMAL FORM EXPRESSION WILL BE PRODUCED?

- Alan Turing proved a fundamental result, called the undecidability of the **halting problem**, which states that there is no algorithmic way to determine whether or not an arbitrary Turing machine will ever stop running. Therefore there are lambda expressions for which it cannot be determined whether a normal order reduction will ever terminate.
- But we might want to fix the order of evaluation when we model a certain language
- In (typical) programming languages, we do not reduce the bodies of functions (under a λ)
  - functions are considered values

CALL BY NAME

- Similar to Normal Order reduction
- Do not evaluate the argument prior to call
- Example:
  - $(\lambda y.\ (\lambda x.\ x)\ y)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta n}$
  - $(\lambda x.\ x)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta n}$
  - $(\lambda u.\ u)\ (\lambda v.\ v) \rightarrow_{\beta n}$
  - $\lambda v.\ v$

- there is no evaluation of the actual parameter at the moment of the call
- The actual parameter will be evaluated only during the execution of the body, if needed

```
if (x==0 || 1/x > 0.3) ... // C++
```

- Does not give any exception for divide by zero

CALL BY VALUE

Same as Applicative order reduction

Evaluate an argument prior to call

Example:

$(\lambda y.\ (\lambda x.\ x)\ y)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta v}$

$(\lambda y.\ (\lambda x.\ x)\ y)\ (\lambda v.\ v) \rightarrow_{\beta v}$

$(\lambda x.\ x)\ (\lambda v.\ v) \rightarrow_{\beta v}$

$\lambda v.\ v$

CALL BY NAME AND CALL BY VALUE

CBN

- difficult to implement
- order of side effects not predictable

CBV:

- easy to implement efficiently
- might not terminate even if CBN might terminate
- Example: (λx. λ z.z) ((λy. yy) (λu. uu))

Outside the functional programming language community, only CBV is used

http://www.lix.polytechnique.fr/~catuscia/teaching/cg428/02Spring/lecture_notes/L07.1.html#:~:text=Call%20by%20need%20is%20similar,evaluated%20each%20time%20is%20needed.

LAZY EVALUATION-CALL BY NEED

It is an evaluation strategy that combines on-demand evaluation with memoisation

- When a variable is bound, it is stored in the environment in an unevaluated form called a thunk, with the evaluation being delayed until the result is required to proceed
- When the thunk is evaluated, it is replaced by the result of its evaluation, avoiding any repetition of work if the value of the variable were to be needed again
- Benefits
  - A more compositional programming style
    - No need to ever construct the entire intermediate results
  - Infinite data and circular definitions
    - This would lead to non-termination in a strict language

COMPARISON

- In call by need the actual parameter is evaluated only the first time it is needed
- Then the value is stored and used whenever it is needed again
- In call by name, on the contrary, the parameter is re-evaluated each time it is needed
- call-by-value is less terminating than call-by-need, as there could be some diverging subterm that a call-by-need strategy would avoid evaluating
- call-by-value is typically favoured by language designers because it is much easier to understand and reason about

Definition: η-reduction

If v is a variable, E is a lambda expression (denoting a function), and v has no free occurrence in E,

$$\lambda v \,.\, (E\ v) \Rightarrow_{\eta} E.$$ ∎

$$\lambda x \,.\, (sqr\ x) \Rightarrow_{\eta} sqr$$

$$\lambda x \,.\, (add\ 5\ x) \Rightarrow_{\eta} (add\ 5).$$

The η-reduction rule can be used to justify the extensionality of functions; namely, if f(x) = g(x) for all x, then f = g

λx.(5 x) is not 5

Extensionality Theor em: If $F_1\ x \Rightarrow^* E$ and $F_2\ x \Rightarrow^* E$ where $x \notin FV(F_1\ F_2)$, then $F_1 \Leftrightarrow^* F_2$ where $\Leftrightarrow^*$ includes η-reductions.

Definition: δ-reduction

If the lambda calculus has predefined constants (that is, if it is not pure), rules associated with those predefined values and functions are called δ rules; for example, $(add\ 3\ 5) \Rightarrow_{\delta} 8$ and $(not\ true) \Rightarrow_{\delta} false$. ∎

LAMBDA CALCULUS TO PROGRAMMING

Data Types

- Booleans, numbers
- Collections

Conditional expressions

Arithmetic expressions

Recursions

a ***combinator*** is a λ-term with no free variables

ENCODING BOOLEANS IN LAMBDA CALCULUS

What can we do with a boolean?

- we can make a binary choice
- *ConditionFunction (condition, then_do, else_do) {*
- *If (condition)*
- *retun then_do*
- *Else*
- *return else_do*
- *}*

- $_{\text{def}}$ $\lambda cond.\lambda then\_do.\ \lambda else\_do.$**??**

ENCODING BOOLEANS IN LAMBDA CALCULUS

What can we do with a boolean?

- we can make a binary choice
- *ConditionFunction (condition, then_do, else_do) {*
- *If (true)*
- *return then_do*
- *Else*
- *return else_do*
- *}*

- $\equiv_{\text{def}}$ $\lambda cond.\lambda then\_do.\ \lambda else\_do.$**??**

ENCODING BOOLEANS IN LAMBDA CALCULUS

What can we do with a boolean?

- we can make a binary choice
- *ConditionFunction (condition, then_do, else_do) {*
- *If (false)*
- *return then_do*
- *Else*
- *return else_do*
- *}*

- $_{\text{def}}$ **λ*then_do*. λ*else_do*.??**

- $_{\text{def}}$ **λ*cond*.λ*then_do*. λ*else_do*.??**

ENCODING BOOLEANS IN LAMBDA CALCULUS

What can we do with a boolean?

- we can make a binary choice
- *ConditionFunction (condition, then_do, else_do) {*
- *If (true)*
- *return then_do*
- *Else*
- *return else_do*
- *}*
- True= $_{def}$ λ*then_do*. λ*else_do*.**then_do**
- **False=** $_{def}$ λ*then_do*. λ*else_do*.**else_do**

BOOLEAN DATA TYPE

A boolean is a function that given two choices selects one of them

- true $=_{def}$ λ*then_do*. λ*else_do*. *then_do*
- false $=_{def}$ λ*then_do*. λ*else_do*. *else_do*
- if_then_else$=_{def}$ λ*cond*.λ*then_do*. λ*else_do*.
- Cond (*then_do*) (*else_do*)

Example:

=if_then_els

SleepHours=(if_then_else)(Any_Assignment_Deadlines)(six)(ten)

$=(_{def}\ \lambda cond.\lambda then\_do.\ \lambda else\_do.$ **Cond** *(then_do) (else_do)*) (**Any..**)(six) (ten)

= (λ*then_do*. λ*else_do*. (Any…)(then_do)(else_do)) (six) (ten)

=(λ*else_do*. (Any…)(six)(else_do)) (ten)

=(Any…)(six)(ten)

=(true)(six)(ten)

= (λ*then_do*. λ*else_do*. then_do) (six)(ten)

=six

ConditionFunction

```
ConditionFunction (Any_Assignment_Deadlines, six,ten) {
If (Any_Assignment_Deadlines)
   return six
Else
    return ten
}
```

HANDLING BOOLEANS

($_{def}$ λ*boolean*.λ*then_do*. λ*else_do*. **boolean** (*else_do*) (*then_do*))

($_{def}$ λ*boolean*.λ*then_do*. λ*else_do*. **boolean** (*else_do*) (*then_do*)) **(true)**

=λ*then_do*. λ*else_do*. **(true)** (*else_do*) (*then_do*)

= λ*then_do*. λ*else_do*. (λ*td*. λ*ed*. **td**) (*else_do*) (*then_do*)

= λ*then_do*. λ*else_do*. *else_do*

Boolean	Outcome of the expression
true	false
false	true

Red_Green=tru

NOT(Red_Green)

NOT(NOT(Red_Green))

NOT(NOT(NOT(Red_Green)))

(five)(NOT)(Red_Green)

(four)(NOT)(Red_Green)

Is_even=

λn. n(NOT)(true)