LAMBDA CALCULUS: AN INTRODUCTION

Chandreyee Chowdhury

LAMBDA CALCULUS

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

"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus." -(Landin 1966)

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

EFFECTIVE COMPUTATION

❑ Mathematicians knew various algorithms for computing things effectively, but they weren't quite sure how to define " effectively computable" in a general way that would allow them to distinguish between the computable and the noncomputable

WHAT IS COMPUTABLE?

In the 1930s, two researchers from two countries Alan Turing from England and Alonzo Church from the US started analyzing the question of what can be computed.

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

- Turing argued that a function is computable if and only if it can be computed on a Turing machine

Because these vastly dissimilar formalisms are all computationally equivalent, the common notion of computability that they embody is extremely robust, which is to say that it is invariant under fairly radical perturbations in the model.

All these mathematicians with their pet systems turned out to be looking at the same thing from different angles. This was too striking to be mere coincidence.

They soon came to the realization that the commonality among all these systems must be the elusive notion of effective computability that they had sought for so long. Computability is not just Turing machines, nor the lambda-calculus, nor the mu-recursive functions, nor the Pascal programming language, but the common spirit embodied by them all.

Alonzo Church gave voice to this thought, and it has since become known as Church's thesis (or the

Church-Turing thesis).

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 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

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

$f = \lambda x. \, x + 5$ $f \; 10$

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

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 ʌ-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, 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:

$$
\begin{aligned}
<\text{expression}> &:= <\text{name}> \mid <\text{function}> \mid <\text{application}> \\
<\text{function}> &:= \lambda <\text{name}>.<\text{expression}> \\
<\text{application}> &:= <\text{expression}><\text{expression}>
\end{aligned}
$$

Variables x

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

EXAMPLES OF LAMBDA EXPRESSIONS

The identity function:

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

Polymorphic expression

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

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

Constant function is defined as

$\lambda y.z$

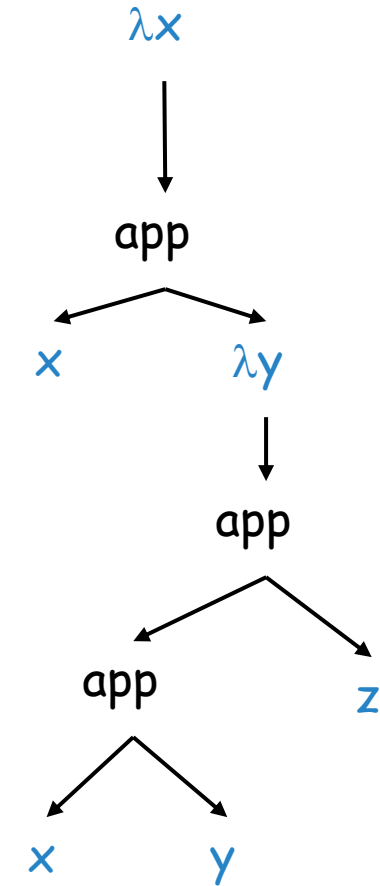# NOTATIONAL CONVENTIONS

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 λx. E binds variable x in E

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

FREE AND BOUND VARIABLES

A variable is said to be <u>free</u> 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$_1$) $\cup$ Free(E$_2$)

Free(λx. E) = Free(E) - { x }   x is free.

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.

RENAMING BOUND VARIABLES

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

Example: $\lambda x.\ x$ is identical to $\lambda y.\ y$ and to $\lambda z.\ z$

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

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

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

$$= \lfloor (\lambda z.w)\backslash x \rfloor ((\lambda y.(x\ y))x))$$
$$= (\lambda y.((\lambda z.w)y))(\lambda z.w)$$
$$= [(\lambda z.w)\backslash y]((\lambda z.w)y)$$
$$= (\lambda z.w)(\lambda z.w)$$
$$= [(\lambda z.w)\backslash z]w$$
$$= w$$

((λa.a) λb.λc.b) (x) λe.f

$((\lambda f.((\lambda g.((f\ f)g))(\lambda h.(k\ h))))(\lambda x.(\lambda y.y)))$

EVALUATION AND THE STATIC SCOPE

The definition of substitution guarantees that evaluation respects static scoping:
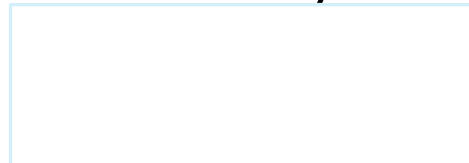
$(\lambda\ x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x))$

(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→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.

β- redex

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:

one $=_{def}$ λf. λs. f (s)

two $=_{def}$ λf. λs. f (f (s))

and so on

zero $=_{def}$ λf. λs. s

To translate the lambda expression to the programming world with datatypes
(i => i+1)(0)

one(i => i+1)(0)

ANYNUMBER=(ΛN. ΛF. ΛX.N(F(X)))

anyNumber One

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

$=>^*_ß \lambda f. \lambda x. (f(x))$

```
function(n){
    Return λf. λx.n(f(x));

    }
```