Lambda Calculus: An Introduction

Chandreyee Chowdhury

Outline

- Why study lambda calculus?

- Lambda calculus
  - Syntax
  - Evaluation
  - Relationship to programming languages

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

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

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

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

- 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 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 ::= $\lambda x.\ x$ | e | $e_1\, e_2$

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

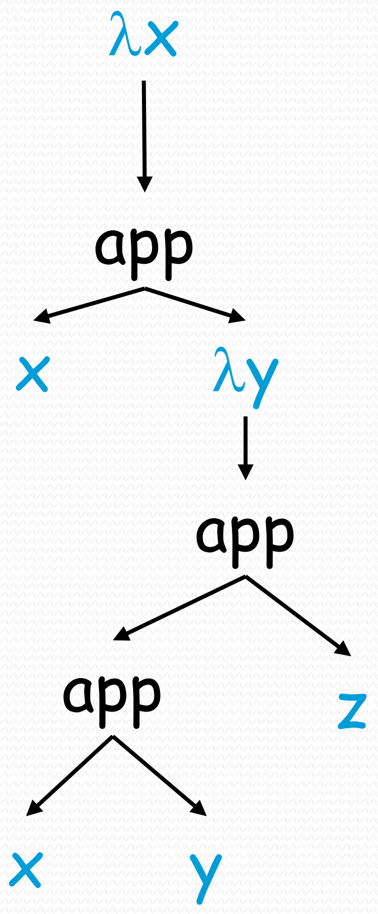
  $x\ y\ z$ parses as $(x\ y)\ z$

- Abstraction extends to the right as far as possible

  $\lambda x.\ x\ \lambda y.\ x\ y\ z$ parses as

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

- 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: $\lambda x.\ x\ (\lambda x.\ x)\ x$

Renaming Bound Variables

- Two λ-terms that can be obtained from each other by a renaming of the bound variables are considered identical
- Example: $\lambda x.\ x$ 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 $E \to_\beta E'$ to say that E' is obtained from E in one β-reduction step
- We write $E \to^*_\beta E'$ if there are zero or more steps

- $(\lambda x.x)$
- int f(int x){
  - return x;
- }
- x=>x;

- f(5);
- $(\lambda x.x)(5)$
- $E1= \lambda x.x$ $E2=5$
- (E1)(E2)

Functions with Multiple Arguments

- Consider that we extend the calculus with the add primitive operation
- The λ-term $\lambda x.\ \lambda y.\ \text{add}\ x\ y$ can be used to add two arguments E_1 and E_2:

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

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

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

  $[E_2/y]\ \text{add}\ E_1\ y = \text{add}\ E_1\ E_2$

- The arguments are passed one at a time

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

$((\lambda a.a)\ \lambda b.\lambda c.b)\ (x)\ \lambda e.f$

$(\lambda b.\lambda c.b)\ (x)\ \lambda e.f$

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

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

- I= $\lambda x.x$
- `Var I = x=>x;`
- `Alert(I("Hi"));`
- fI= $(\lambda f.f)\ (\lambda x.x)$

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

$)((\lambda g.\ \lambda s.g(s))$

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

function(n){

Return λf. λx.n(f(x));

}

anyNumber=(λn. λf. λx.n(f(x)))

- λn. n ((λf. f+1)(0))

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

- Successor=(λn. λf. λx.f(n(f(x))))
- Successor(ONE)
- =(λn. λf. λx.f(n(f(x))))(λf. λx.f(x))
- = λf. λx. f(f(x))
- (Successor)(two)=(λn. λf. λx.f(n(f(x))))(λg.λs.g(g(s)))

- Successor Two
- $=(\lambda n.\ \lambda f.\ \lambda x.f(n(f(x))))(\lambda f.\ \lambda x.f(f(x)))$

Any Natural Number

anyNumber = $\lambda f.\ \lambda s.\ f\ s$

- $_{\text{def}}$ $\lambda n.\ \lambda f.\ \lambda s.\ n(f\ (s))$
- $_{\text{def}}$ $((\lambda f.\ f{+}1)(0))$
- $\lambda n.\ \lambda f.\ n\ ((f{+}1)(0))$

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

Addition

- 3+2
- Apply successor 3 times on 2
- nf(x)→ n times f is applied on x
- 3 successor(number)
- Three successor two
- Successor= $\lambda n.\lambda f.\lambda x.f(nfx)$
- One=$\lambda g.\lambda s.g(s)$

- $\lambda m.\lambda n.\lambda f.\lambda x.m(f)(n(f)(x))$ one one