Lambda Calculus (λ Calculus)

March 10, 2021

Outline

- what is lambda calculas
- beta reducton and alpha renaming
- lazy evaluation
- Church encoding

Overview

- The smallest programming language
- Syntax and examples
- $\alpha-$renaming and $\beta-$reduction
- order of evaluation
- λ encoding
- Books: The Lambda Calculus. Its Syntax and Semantics, An Introduction to Functional Programming Through Lambda Calculus

Smallest Universal Programming Language

- $e ::= x | \lambda x.e | e_0\ e_1$

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

- As a programming language, sometimes a concrete implementation of lambda calculus also supports predefined constants such as '0' '1' and predefined functions such as '+' '*'

Examples

- $\lambda x.x$ (lambda abstraction: building new function)
- $(\lambda x.x)\ y$ (function application)

The parenthesis is to separate function and function application.

What is λ Calculus and Why is it important?

1. A mathematical language; A formal computation model for functional programming; a theoretical foundation for the family of functional programming languages.
2. Study interactions between functional abstraction and function applications
3. By Alonzo Church in the 1930s
4. In 1920s - 1930s, the mathematicians came up different systems for capturing the general idea of computation:
   - Turing machines – Turing
   - m-recursive functions – Gödel
   - rewrite systems – Post
   - the lambda calculus – Church
   - combinatory logic – Schönfinkel, Curry

   These systems are all computationally equivalent in the sense that each could encode and simulate the others.

The Mathematical Precursor to Scheme

Mathematical formalism to express computation using functions:

- Everything is a function. There are no other primitive types—no integers, strings, objects, booleans ... If you want these things, you must encode (simulate) them using functions.
- No state or side effects. It is purely functional. Thus we can think exclusively in terms of the substitution model.
- Only unary (one-argument) functions.

Implementation in Scheme/DrRacket

- Syntax implemented in Scheme:

$$
\begin{array}{rcll}
e & \rightarrow & x & \text{Variable} \\
 & | & (_x\ \lambda\ (x)\ e\ )_x & \text{a lambda expression} \\
 & | & (e\ e) & \text{Application}
\end{array}
$$

$$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$$

Lambda Calculus: Review

- Smallest language
- Syntax

S →	*Name*	*Function*	*Application*
	x	$\lambda x.x$	$\lambda x.x\ y$
			$(\lambda x.x)\ y$
			$(\lambda x.x\ y)$
			$(\lambda(x)\ y)$
			$(Lambda(x)\ y)$
			$({}_x\lambda(x)\ y)_x$

  ( ) helps with clarity
  $({}_x\ )_x$ helps with clarity
- Turing machine is the model for imperative programming language, and, λ calculus is the model for functional programming language. They have equal computing power and can simulate each other.

Lambda Calculus Semantics

- β-reduction

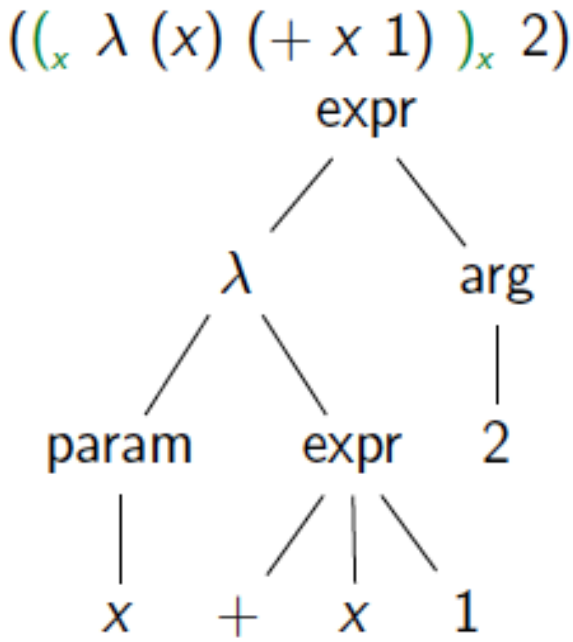
$$(\lambda x.x) \quad x$$

Function definition Actual parameter

The bound variable x will be replaced by the argument expression in the body expression x.

β-reduction

The AST view:



β-reduction

((λ(x) e1) e2) evaluates the expression e1 by replacing every ("free") occurrences of x in e1 using e2, denoted as e1[x→e2]

- ((λ(x) (λ(y) (+ x y))) 1)
- ((((λ (x) (λ(y) (λ (x) (+ x y)))) 1) 2) 3)

β-reduction (Contd.)

((λ(x) e1) e2) evaluates the expression e1 by replacing every ("free") occurrences of x in e1 using e2, denoted as e1[x→e2]

- ((λ(x) (λ(y) (+ x y))) 1)
  output: (λ(y) (+ 1 y))

β-reduction

((λ(x) e1) e2) evaluates the expression e1 by replacing every ("free") occurrences of x in e1 using e2, denoted as e1[x→e2]

- ((λ(x) (λ(y) (+ x y))) 1)
- ((((λ (x) (λ(y) (λ (x) (+ x y)))) 1) 2) 3)
  = (((λ(y) (λ (x) (+ x y))) 2) 3)
  = ((λ (x) (+ x 2)) 3)
  = 5

α-renaming

Rename a variable:

- ((((λ(x) (λ(y) (λ(x) (+ x y)))) 1) 2) 3)
- ((((λ(x) (λ(y) (λ(z) (+ z y)))) 1) 2) 3)

The Order of Evaluation

The order of evaluation does not impact β-reduction results if the evaluation terminates:

((λ(x) (+ x 1)) ((λ(y) (+ y 1)) 2))

Lazy Evaluation

- The evaluation is deferred until their results are needed.
- Arguments are not evaluated before they are passed to a function, but only when their values are actually used.

((λ(x) (+ x 1)) ((λ(y) (+ y 1)) 2))

((λ(x) 1) ((λ(y) (+ y 1)) 2))

Currying

Pure lambda calculus pairs one variable with one λ

- Functions with many parameters
  $(_{x\ y}\ \lambda\ (x\ y)\ e\ )_{x\ y}$: two formal parameters x and y
- Semantically equivalent expression: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ e\ )_y\ )_x$

Concept introduced by Haskell Curry.

Simulating the Computation

What is the result of ((λ(x)x) (λ(y)y))

Soln: (λ(y)y)

Identity Function

(λ(x)x): this function applies to any entity return the entity itself

f(x) = x+0 or f(x) = x*1 or f(x) = x-0

Self-Application Function

(λ(x)(x x)): this function applies to any entity will return the entity aplies to itself

((λ(x)(x x)) 3)
Soln: (3 3)

((λ(x)(x x)) (λ(y) y))
Soln: ((λ(y) y) (λ(y) y))
(λ(y) y)

More Examples

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

- $(f\ x)$ means application of function f on x.
- It is a function without formal parameter, cannot be reduced any further through β-reduction.

$$\underset{F}{(}\ ((\lambda(f)\ (\lambda(x)\ (f\ x)\ ))\ \underset{A_1}{(\lambda(y)\ y\ ))}\ \underset{A_2}{(\lambda(z)\ (z\ z)\ ))}$$

Let $(\lambda(y)\ y\ ) = g$ and $(\lambda(z)\ (z\ z)\ ) = v$

$((\lambda(x)\ (g\ x)\ )\ v)$
$= (g\ v)$
$= ((\lambda(y)\ y\ )\ v)$
$= v$
$= (\lambda(z)\ (z\ z)\ )$

Church encoding

- In mathematics, Church encoding is a means of representing data and operators in the lambda calculus
- The Church numerals are a representation of the natural numbers using lambda notation.
- The method is named for Alonzo Church, who first encoded data in the lambda calculus this way.

Church encoding

Representing data and operations using functions:

- non-negative integers: 0, 1, 2 ..., and their computation of addition, subtraction, multiplication ...
- boolean, true, false, and, or, not, ite (if then else control flow)
- pairs and lists
- rational numbers: may be encoded as a pair of signed numbers
- computable real numbers may be encoded by a limiting process that guaranetees that the difference from the real value differs by a number which maybe as small as we need (considering pi)
- complex numbers are naturally encoded as a pair of real numbers.

Encoding Natural Numbers (Church Numbers)

- Encoding of numbers: 0, 1, 2, . . . , as functions such that their semantics follow the natural number semantics (e.g., counting, sequencing)
- Intuition: The number n means how many times one can do certain operation.
- A natural number encoding function takes two arguments (function and entity on which the function is to be applied)

Encoding Natural Numbers

zero $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ x\ )_x\ )_f$

one $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$

two $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ (f\ x))\ )_x\ )_f$

n $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ \ldots(f\ x)\ldots))\ )_x\ )_f$

Assume f is operation and x is the object on which the operation is done.

E.g.: f is adding '1' to the list

E.g.: x is an empty list

Then,
meaning of zero is empty list ( )
meaning of one is (1)
meaning of two is (11)
meaning of three is (111)

Example

What is the semantics of

- $((two\ g)\ z)$: two applications of g on z.

$$(((_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ (f\ x))\ )_x\ )_f\ g)\ z) = (g\ (g\ z))$$

- $((n\ g)\ z)$: n applications of g on z, where n is a natural number.
- $(_z\ \lambda\ (z)\ ((n\ g)\ z)\ )_z$: n applications of g on the formal parameter z, where n is a natural number. This result is a function (z if the formal parameter of the function).

More examples of understanding the functions of natural numbers

(two g) =
($(_f\ \lambda(f)\ (_x\ \lambda(x)(f\ (f\ x))_x)_f$ g) =
$(_x\ \lambda(x)(g\ (g\ x))_x)$

This is a function that takes a parameter x and returns the result of applying the function g twice on x.

($(_z\ \lambda(z)\ ((three\ f)\ z)\ )_z\ two$)

Soln:
(f(f(f two)))

Operations on natural numbers

- successor function: succinct representation of any number
- addition
- multiplication
- subtraction

the Successor functinon

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

n: the number whose successor we want to compute.
$((n\ f)\ x)$: n applications of the function f on x, i.e., $(f^n x)$. Therefore, $(f\ ((n\ f)\ x))$ is $(f\ (f^n\ x))$, which is representation of $n+1$.

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$(succ\ zero)$
$= ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n\ zero)$
$= \quad (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f$

$(zero\ f) = ((_g\ \lambda\ (g)\ (_y\ \lambda\ (y)\ y\ )_y\ )_g\ f) = (_y\ \lambda\ (y)\ y\ )_y$

Therefore,
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f =$
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((_y\ \lambda\ (y)\ y\ )_y\ x))\ )_x\ )_f =$
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f =$
one

Exercise

(succ (succ zero))

Encoding Add using the Successor functinon

Semantics of Addition: Addition takes two natural numbers as arguments and produces a new natural number whose semantics/representation is the sum of two inputs.

- Add requires two arguments and returns a natural number representation:
  $(_m\, \lambda(m)(_n\, \lambda(n)(_f\, \lambda(f)(_x\, \lambda(x)\, \ldots\,)_x)_f)_n)_m$
  The definition is follows:
  - ((m succ) n) generates:
    (succ (succ (succ . . . (succ n)) . . .))
    This produces a natural number equivalent to m+n
  - It should be applied to some function f to represent the natural number
    ((((m succ) n) f) x)
- Therefore, the add function is:
  $(_m\, \lambda(m)(_n\, \lambda(n)(_f\, \lambda(f)(_x\, \lambda(x)\, ((((m\, succ)\, n)\, f)\, x)\,)_x)_f)_n)_m$

Exericse

((add (succ zero)) (succ zero))

Natural Numbers Encoding in λ-calculus (Revisit)

Representation of:

$$\text{zero}\ \ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ x\ )_x\ )_f$$
$$\text{one}\ \ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$$

Each number is represented by the number of times some function is applied to some entity.

Natural Numbers Encoding in λ-calculus (Revisit)

- Develop a successor function *succ* that takes a natural number as an argument and returns the next natural number.
  - Example:
    (succ zero) should return $(_f\ \lambda(f)(_x\ \lambda(x)\ (f\ x)_x)_f = \text{one}$
    (succ one) should return $(_f\ \lambda(f)(_x\ \lambda(x)\ (f\ (f\ x))_x)_f = \text{two}$

Natural Numbers Encoding in λ-calculus (Revisit)

add: $({}_m\ \lambda\ (m)\ ({}_n\ \lambda\ (n)\ ({}_f\ \lambda\ (f)\ ({}_x\ \lambda\ (x)\ ((((m\ succ)\ n)\ f)\ x)\ )_x\ )_f\ )_n\ )_m$

Apply *succ* m times to create a function that is applied n. E.g., $m = 2$ and $n = 3$ results in *succ* of *succ* of 3, which is 5.

Boolean and if-then-else (ITE)

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c$

$$
\begin{aligned}
&(((ite\ true)\ s_1)\ s_2) = \\
&(((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2) = \\
&\qquad (((_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((true\ t)\ e)\ )_e\ )_t\ s_1)\ s_2) = \\
&\qquad\qquad ((_e\ \lambda\ (e)\ ((true\ s_1)\ e)\ )_e\ s_2) = \\
&\qquad\qquad\qquad ((true\ s_1)\ s_2) = \\
&(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x\ s_1)\ s_2) = \\
&\qquad (((_y\ \lambda\ (y)\ s_1\ )_y\ s_2) = s_1
\end{aligned}
$$

Boolean Operators

not a: if *a* then false else true. (((*ite a*) *false*) *true*)

```
or a b:
      ( (ite a) true
            ((ite b) true false)
      )
```

What is the adequate set of operators for boolean logic?

Boolean Operations

- Basic operators: and, or, not
- which then can be extended to xor, implication and equievelnce

$$x \to y = \neg x \vee y$$
$$x \oplus y = \neg(x \equiv y) = (x \vee y) \wedge (\neg x \vee \neg y) = (x \wedge \neg y) \vee (\neg x \wedge y)$$
$$x \equiv y = \neg(x \oplus y) = (x \wedge y) \vee (\neg x \wedge \neg y)$$

These definitions give rise to the following truth tables giving the values of these operations for all four possible inputs.

Secondary operations. Table 1

x	y	$x \to y$	$x \oplus y$	$x \equiv y$
0	**0**	1	0	1
1	**0**	0	1	0
0	**1**	1	1	0
1	**1**	1	0	1

More about Lambda Caculus

- there is no recursion in lambda calculus
- do we program in lambda calculus? mostly no
- it helps understand functional computation
- it helps design new functional programming languages

Problem Solving

- beta reduction:
  - parsing the function
  - understand what is the function, what is the actual parameters
  - remember steps of function applications, even when functions and actual parameters can be complex
- beta reduction and encoding recognition (e.g., what does this function compute?):
  - remember existing encoding and functions
  - understand the semantics of certain functions and patterns, e.g., identity function, natural numbers ...

Further Learning Materials

- Lambda calculus examples:
  `https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/lambda-calculus-handout.pdf`
- Programming Languages: Lambda Calculus - 1
  `https://www.youtube.com/watch?v=v1IlyzxP6Sg`
- Programming Languages: Lambda Calculus - 2
  `https://www.youtube.com/watch?v=Mg1pxUKeWCk`