Lambda Calculus (λ Calculus)

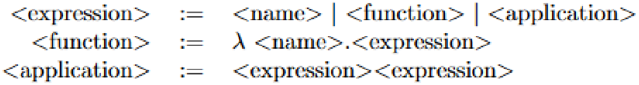
October 25, 2019

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 by Henk Barendregt, 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 It Is 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 – Gdel
   - rewrite systems – Post
   - the lambda calculus – Church
   - combinatory logic – Schnfinkel, 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, cons objects, Booleans ... If you want these things, you must encode them using functions.
- No state or side effects. It is purely functional. Thus we can think exclusively in terms of the substitution model.
- The order of evaluation is irrelevant.
- Only unary (one-argument) functions.

Implementation in Scheme/DrRacket

- Syntax implemented in Scheme:

$$\begin{array}{lcll} e & \rightarrow & x & \text{Variable} \\ & \mid & (_{x}\ \lambda\ (x)\ e\ )_{x} & \text{a lambda expression} \\ & \mid & (e\ e) & \text{Application} \end{array}$$

$((_{x}\ \lambda\ (x)\ (+\ x\ 1)\ )_{x}\ 2)$

Compute $.(2)$ where $.(x) = x + 1$

Lambda Calculus: Review

- Smallest language
  Syntax

  $$\begin{array}{lcccc} S \rightarrow & \textit{Name} & \mid\ \textit{Function} & \mid\ \textit{Application} \\ & x & \mid\ \lambda x.x & \mid\ (\lambda x.x\ \lambda x.x) \\ & & & (\lambda x.x\ \lambda y.y) \\ & & & \lambda x.x\ y \\ & & & (\lambda x.x)\ y \\ & & & (\lambda x.x\ y) \\ & & & (\lambda(x)\ y)\ \lambda x.y \\ & & & (\textit{Lambda}(x)\ y) \\ & & & (_x\lambda(x)\ y)_x \end{array}$$

  $(\quad)$ helps with clarity
  $(_x \quad)_x$ 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 one another.

Lambda Calculus: Review

- Semantics:
  - λ expression
  - λ term
  - β-reduction

    e.g., $\underbrace{(\lambda x.x)}_{\text{FD}} \quad \underbrace{x}_{\text{AP}}$

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

    After β-reduction, the expression will be evaluated as x.

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$\rightarrow$e2]

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

β-reduction (Contd.)

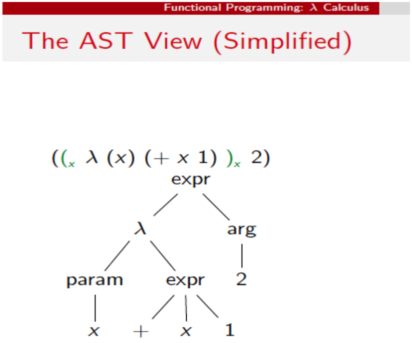
((λ(x) e1) e2) evaluates the expression e1 by replacing every (free) occurrences of x in e1 using e2, denoted as e1[x$\rightarrow$e2]

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

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

β-reduction (Contd.)

AST view for a similar example:



Functional Programming: λ Calculus

The AST View (Simplified)

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

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$\rightarrow$e2]

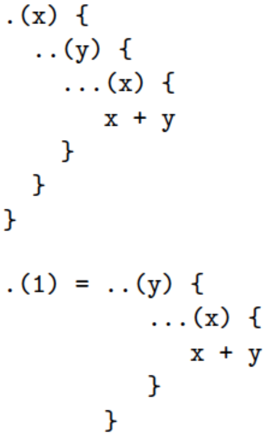
- ((λ(x) (λ(y) (+ x y))) 1)
- ((((λ (x) (λ(y) (λ (x) (+ x y)))) 1) 2) 3)
  = (((λ(y) (λ (x) (+ x y))) 2) 3)
  = ((λ (x) (+ x 2)) 3)
  = 5

β-reduction (Contd.)

Another way to visualize:

How about

$$((((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (_x\ \lambda\ (x)\ (+\ x\ y)\ )_x\ )_y\ )_x\ 1)\ 2)\ 3)\ )$$



```
.(x) {
   ..(y) {
      ...(x) {
         x + y
      }
   }
}

.(1) = ..(y) {
          ...(x) {
             x + y
          }
       }
```

α-renaming

α-Conversion
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

Expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations. In consequence, arguments are not evaluated before they are passed to a function, but only when their values are actually used. (call by name)

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

Soln: $(\lambda(y)y)$

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)

Recap

- $(_x\ \lambda\ (x)\ e\ )_x$: a lambda expression representing definition of function
- $(\ (_x\ \lambda\ (x)\ e\ )_x\ p)$: a lambda expression representing application of a function.
  - Formal parameter: x
  - Actual argument: p
  - Computation: $e[x \mapsto p]$, replace free occurrences of x in e with p (β-reduction)
- Order of β-reduction does not impact the result if each β-reduction terminates

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

More Examples

(λ(f) (λ(x) (f x)))
Soln:
It means application of function f on x. It cannot be reduced any further through β-reduction.

(((λ(f) (λ(x) (f x))) (λ(y) y)) (λ(z) (z z)))
F A_1 A_2
Soln:
Let, (λ(y) y) = g and (λ(z) (z z)) = v,
So, the equation becomes,
((λ(x) (g x)) v)
= (g v)
= ((λ(y) y) v)
= v
= (λ(z) (z z))

Church encoding

Software that could, in theory, be translated into lambda calculus. 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
- Once real numbers are defined, complex numbers are naturally encoded as a pair of real numbers.

Church Encoding

Representing data and operations using functions

- non-negative integers, 0, 1, 2 ...
- booleans, true, false, and, or, not, ite
- pairs
- Rational numbers may be encoded as a pair of signed numbers.
- Real numbers may be encoded by a limiting process that guarantees that the difference from the real value differs by a number which may be made as small as we need

Natural Numbers (Church Numbers)

- Encoding of numbers: 0, 1, 2, . . . , as functions such that their semantics follows the natural number semantics.
- 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

- $((\mathit{two}\ g)\ z)$: 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).

What about?

$((_z\ \lambda(z)\ ((\mathit{three}\ f)\ z)\ )_z\ \mathit{two})$

Soln:
$(f(\ f(\ f\ \mathit{two})))$

Encoding Natural Number

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

Successor Function

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

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

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

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:

How about?

$(succ\ (succ\ zero))$

More with successors

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.

Natural Numbers Encoding in λ-calculus (Revisit)

Representation of:

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

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

... (Definitions from slide 27)

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

- For instance:
  1. ((two g) z) =
     $(((_f\ \lambda(f)\ (_x\ \lambda(x)(f\ (f\ x))_x)_f$ g) z) =
     (g (g z)) // after two β-reductions
- To re-iterate, what is meaning of (two g) ?
  (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.

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 $(_f\ \lambda(f)(_x\ \lambda(x)\ (f\ x)_x)_f\ =$ one
    (succ one) should return $(_f\ \lambda(f)(_x\ \lambda(x)\ (f\ (f\ x))_x)_f\ =$ two
- Soln: Slide 31.

Natural Numbers Encoding in λ-calculus (Revisit)

- Addition operation/function takes two natural numbers as arguments and produces a new natural number whose semantics/representation is the sum of two inputs.
  - As add requires two arguments and returns a natural number representation which takes two other arguments, the form of add function is:
    $({}_m\, \lambda(m)({}_n\, \lambda(n)({}_f\, \lambda(f)({}_x\, \lambda(x) \ldots)_x)_f)_n)_m$
    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:
    $({}_m\, \lambda(m)({}_n\, \lambda(n)({}_f\, \lambda(f)({}_x\, \lambda(x)\ ((((m\ succ)\ n)\ f)\ x)\ )_x)_f)_n)_m$
  - Exercise: ((add (succ zero)) (succ zero))

Booleans

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:

Soln: or a b:

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

What is the adequate set of operators for boolean logic?

More

- 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

Review and Further Reading

- Concepts: Lambda abstraction and function application, high order functions
- bound and free variables
- currying
- $\beta-$reduction and $\alpha-$conversion
- church encoding

Further reading: pass by name, pass by value, lazy evaluation

- 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`