Lambda Calculus

lambda calculus 是一个编程语言，也是一种计算模型（与图灵机计算能力等价）

Syntax

使用 BNF 写出 lambda term 的话，其形式有三种（设 M, N 为 lambda term）

- 变量 x 是 lambda term
- λx. M 是 lambda term（lambda abstraction）
- M N 是 lambda term（lambda application）

上述定义构成了 pure lambda calculus

Conventions

λ 所定义的 body 会尽可能向右扩展，而 lambda application 是左结合的，即 λx. M N = λx. (M N)，M N P = (M N) P

在 lambda calculus 中，函数可以被作为参数或返回值使用

lambda calculus 中函数只能接受一个参数，多参数函数是通过 currying 实现的：λ(x,y). x - y $\to$ λx. λy. x - y

Free and bound variables

对于一个函数 λx. x + y，x 被称为 bound variable，y 被称为 free variable

bound variable 起到占位符的作用，可以任意替换，λ 作用的范围即为 bound variable 的 scope，而 free variable 的名是有意义的，不能随意替换

如果对于 term M, N，定义 fv(M) 为 M 的 free variable 集合，则有

- fv(x) = {x}
- fv(λ x. M) = fv(M) $\setminus$ {x}
- fv(M N) = fv(M) $\cup$ fv(N)

Substitution

Definition: M[N/x]: replace x by N in M

这个定义同样是基于 term 递归定义的

- x[N/x] = N

- y[N/x] = y
- (M P)[N/x] = (M[N/x]) (P[N/x])

在对 lambda abstraction 替换时，要注意替换对象只能为 free variable，替换前要考虑替换后是否会有歧义。考虑 (λx. x - y)[x/y]，此时可以先改变 bound variable 的名字：(λz. z - y)[x/y] = λz. z - x

根据此原理补充上述定义

- (λx. M)[N/x] = λx. M
- (λy. M)[N/x] = λy. (M[N/x]) if y $\notin$ fv(N)
- (λy. M)[N/x] = λy. (M[z/y][N/x]) if y $\in$ fv(N) and z is unused

有了 substitution 就可以定义 lambda calculus 的两个基本规则

- α-equivalence: λx. M = λy. M[y/x]，即 bound variable 可以任意替换（前提是替换结果 y is unused）
- β-reduction: (λx. M) N $\to$ M[N/x]，即可以将参数带入 lambda abstraction

Reduction

Normal form

在对 lambda calculus 做 reduction 时，有几个基本的 reduction rule

$$\frac{}{(\lambda x.\, M)\ N \to M[N/x]}$$

$$\frac{M \to M'}{M\ N \to M'\ N}$$

$$\frac{N \to N'}{M\ N \to M\ N'}$$

$$\frac{M \to M'}{\lambda x.\, M \to \lambda x.\, M'}$$

如果定义 β-redex (reducible expression) 为形如 (λx. M) N 的 term，则定义 β-normal form 为不含 β-redex 的 lambda term

normal form 不能进一步进行 β-reduction

Confluence：可以按照任意顺序运算 lambda term，如果其有结果，则其结果唯一，与运算顺序无关

可以递归定义 $\to^*$ 为零步或多步 reduction

Basis. $M \to^0 M' \iff M = M'$

Induction. $M \to^{k+1} M' \iff \exists M''. M \to M'' \wedge M'' \to^k M'$

于是有 $M \to^* M' \iff \exists k. M \to^k M'$

则 Confluence 的形式化描述为

$$M \to^* M_1 \wedge M \to^* M_2 \implies \exists M'. M_1 \to^* M' \wedge M_2 \to^* M'$$

根据 α-equvalence，每个 term 最多只有一个 normal form，但不是所有的 term 都有 normal form，如 (λx. x x) (λx. x x)，对其进行 reduction 会陷入无限循环，除此之外即使是有 normal form 的 term，视运算顺序也会有得不出 normal form 的情况，如 (λu. λv. v) ((λx. x x) (λx. x x))

Reduction strategies

有两种主要的化简策略

- Normal-order reduction：选择最左最外（不包含于任何 redex 内）的 redex
- Applicative-order reduction：选择最左最内（内部不包含任何 redex）的 redex

如果一个 term 有 normal form，则 normal-order reduction 一定会得出这个 form

Evaluation

Evaluation 与 Reduction 的区别在于

- 只对 closed term (no free variables) 进行求值
- 不对 lambda 内部的内容进行 reduce

一旦得出一个 lambda abstraction 则 evaluation 立即停止，此时得到的 term 称为 canonical form，显然

- a closed normal form must be a canonical form
- not every closed canonical form is a normal form

在一个终止的 normal-order reduction 序列中一定含有第一个 canonical form

evaluation 同样有其规则

Normal-order evaluation

$$\frac{}{\lambda x. M \Rightarrow \lambda x. M}$$

$$\frac{M \Rightarrow \lambda x. M' \quad M'[N/x] \Rightarrow P}{M\ N \Rightarrow P}$$

Eager-order evaluation

$$\overline{\lambda x.\, M \Rightarrow_E \lambda x.\, M}$$

$$\frac{M \Rightarrow_E \lambda x.\, M' \quad N \Rightarrow_E N' \quad M'[N'/x] \Rightarrow_E P}{M\ N \Rightarrow_E P}$$

即 eager-order 将参数化简至 canonical form 再进行替换

如果写成 small step 则为

Normal-order evaluation (small-step)

$$\overline{(\lambda x.\, M)\ N \to M[N/x]}$$

$$\frac{M \to M'}{M\ N \to M'\ N}$$

Eager-order evaluation (small-step)

$$\overline{(\lambda x.\, M)\ (\lambda y.\, N) \to M[(\lambda y.\, N)/x]}$$

$$\frac{M \to M'}{M\ N \to M'\ N}$$

$$\frac{N \to N'}{(\lambda x.\, M)\ N \to (\lambda x.\, M)\ N'}$$

Programming in lambda calculus

Basic

可以定义出布尔值和相关的操作符

- True $\equiv$ λx. λy. x
- False $\equiv$ λx. λy. y
- not $\equiv$ λb. b False True
- and $\equiv$ λb. λb'. b b' False
- or $\equiv$ λb. λb'. b True b'
- if b then M else N $\equiv$ b M N

以及自然数和相关的操作符（Church numerals）

- 0 $\equiv$ λf. λx. x
- 1 $\equiv$ λf. λx. f x
- 2 $\equiv$ λf. λx. f (f x)
- succ $\equiv$ λn. λf. λx. f (n f x)

- iszero $\equiv \lambda$n. λx. λy. n (λz. y) x
- add $\equiv \lambda$n. λm. λf. λx. n f (m f x)
- mult $\equiv \lambda$n. λm. λf. n (m f)

以及 pair

- (M N) $\equiv \lambda$f. f M N
- $\pi_0 \equiv \lambda$p. p (λx. λy. x)
- $\pi_1 \equiv \lambda$p. p (λx. λy. y)

tuple 的定义同理

Recursive

阶乘函数：fact(n) = if (n == 0) then 1 else n * fact(n - 1)

由于 lambda 没有名，故不能在函数体内引用自身

在数学上函数的 fix point 是指一个输入满足 x = f(x)

fact 也可以写成

fact = (λf. λn. if (n == 0) then 1 else n * f (n-1)) fact

如果令 F = λf. λn. if (n == 0) then 1 else n * f (n-1)，则 fact = F fact，即 fact 是 F 的 fix point

在 lambda calculus 中，满足每个 term 都有一个 fix point，定义 fix point combinator 为一个高阶函数 h 满足

$$\forall f. h\ f = f\ (h\ f)$$

即 h 应用于任何函数都可以得到其 fix point，Turing 和 Church 都给出了 fix point combinator

- Turing: Θ，令 A = λx. λy. y (x x y)，则 Θ = A A
- Church: Y = λf. (λx. f (x x)) (λx. f (x x))

使用 combinator 即可得出 fact 的定义：fact = Θ F