

example-image.pdf

形式语言及其逻辑

作者：面皮/Mepy

时间：20220806

版本：0.1

所谓形式，不过是同义反复，然而构造之却并非如此。

目录

第 1 章 无类型 λ 演算 Untyped λ-Calculus	1
1.1 语法 Syntax	1
1.2 闭项 Closed Term	2
1.3 等式语义 Equation Semantics	3
1.4 布尔值 Boolean	4
1.5 元组 Tuple	7
1.6 递归 Recursion	7
1.7 列表 List	9
1.8 自然数 Natural Numbers	10

第 1 章 无类型 λ 演算 Untyped λ -Calculus

内容提要

□ 语法 1.1

□ 封闭 1.4

□ 代换 1.8

□ 等式 1.9

□ 布尔 1.10 1.11

□ 元组 1.14

□ ω 组合子 1.17

□ Y 组合子 1.18

本章首先介绍 [无类型] λ 演算的语法, 然后引入等式语义. 借由等式语义, 我们在 λ 演算中建模布尔值等概念, 以演示 λ 演算的图灵等价性, 这意味着所有可计算的程序均可在 λ 演算中建模.

1.1 语法 Syntax

比起图灵机, λ 演算从语法上更像一门编程语言, 且仅有变量, 函数 [定义] 与 [函数] 调用三种:

定义 1.1 (λ 演算的语法, the syntax of λ -calculus)

λ 演算中仅有一类对象, 我们称之为表达式 (expression) 或词项 (term), 并用元变量 t 表示, 定义如下:

$t = x$ (变量)

$= \lambda x. t'$ (函数)

$= t_1 t_2$ (调用)

尽管本节内容是语法, 但为了更好地理解 λ 演算, 我们在此通过定义概念的方式指出调用 $t_1 t_2$ 的语义:

t_1 是被调用的函数, 执行时具有 $\lambda x. t'_1$ 的形式, 其中 x 是 t_1 的 [形式] 参数 (parametre);

t_2 是传入函数的 [实际] 参数 (argument).

关于优先级与结合性, 我们约定: $()$ 提升优先级至最高; 调用优先于函数; 调用左结合.



笔记 上述定义中的变量 x 与元变量 t 是两个层级的事物:

变量 (variable) 是 λ 演算中的一部分, 通常使用 x, y 等字母表示;

元变量 (meta-variable) 是我们作为高于 λ 演算的创造者, 对词项的定义/命名/称呼, 通常使用 t 表示.

元 (meta) 这一词根, 表示超越、在某之上的含义, 又如形而上学 (meta-physics).

下面给出一些词项的例子, 其中 $t_1 = t_2$ 表示了 t_1 是 t_2 略去多余括号的写法:

1. x
2. $\lambda x. x y = \lambda x. (x y)$
3. $(\lambda x. x) y$
4. $x y z = (x y) z$
5. $x (y z)$
6. $\lambda x. \lambda y. x = \lambda x. (\lambda y. x)$
7. $\lambda x. \lambda y. y = \lambda x. (\lambda y. y)$

第 1 个例子是变量 x . 第 2 个与第 3 个例子是函数与调用的嵌套, 这说明了调用优于函数.

第 4 个与第 5 个例子均是调用自身的嵌套, 这说明了调用的左结合性. 注意, 不同于常见的编程语言, λ 演算中的调用不需要括号, 括号仅用来表示最高优先级.

第 6 个与第 7 个例子均是函数自身的嵌套. 从定义中可以看出, λ 演算中的函数总是单变量的. 尽管我们没有多变量函数, 但通过形如这两个例子的手段, 我们可以模拟多参数函数, 这种手段称为 Curry 化. 至于多参数函数的调用, 举第 4 个例子 $x y z$, 我们对函数 x 传入参数 y 调用之后, 再次传入参数 z 进行调用, 这就实现了传入两个参数 y 与 z .

1.2 闭项 Closed Term

上一节中给出的例子有一处与我们的编程经验冲突, 以 $\lambda x.x y$ 为例, 例中的 y 代表什么含义? 我们都知道, y 这类未被定义/初始化的变量会引入未知的错误. 而 λ 演算中变量被定义的唯一方法, 仅仅是作为函数参数而被绑定. λ 演算中, 变量是一个空洞的存在, 唯有经由函数绑定, 变量才能获得语义. 那些被绑定的变量, 具有确定的语义, 称为绑定变量 (bound variables); 那些未被绑定的便是自由变量 (free variables):

定义 1.2 (绑定变量, bound variables)

词项 t 所含有的绑定变量 $BV(t)$ 如下定义:

$$\begin{aligned} BV(x) &= \emptyset && \text{(变量)} \\ BV(\lambda x.t') &= \{x\} \cup BV(t') && \text{(函数)} \\ BV(t_1 t_2) &= BV(t_1) \cup BV(t_2) && \text{(调用)} \end{aligned}$$

定义 1.3 (自由变量, free variables)

词项 t 所含有的自由变量 $FV(t)$ 如下定义:

$$\begin{aligned} FV(x) &= \{x\} && \text{(变量)} \\ FV(\lambda x.t') &= FV(t') - \{x\} && \text{(函数)} \\ FV(t_1 t_2) &= FV(t_1) \cup FV(t_2) && \text{(调用)} \end{aligned}$$

我们可以举一些例子:

1. $BV(\lambda x.x y) = \{x\}, FV(\lambda x.x y) = \{y\}$
2. $BV(x y z) = \emptyset, FV(x y z) = \{x, y, z\}$
3. $BV(\lambda x.\lambda y.x) = \{x, y\}, FV(\lambda x.\lambda y.x) = \emptyset$
4. $BV(\lambda x.\lambda y.y) = \{x, y\}, FV(\lambda x.\lambda y.y) = \emptyset$

自由变量为空的词项所含有的变量全部都被绑定了, 因而语义具有确定性, 我们定义这种重要的词项.

定义 1.4 (封闭, closed)

词项 t 称为封闭的 (closed), 或称其闭项 (closed term) 当且仅当 $FV(t) = \emptyset$, 我们用元变量 s 表示闭项;
词项 t 相对于词项 t' 封闭 (closed) 当且仅当 $FV(t) \cap BV(t') = \emptyset$.

引入闭项, 对语义很重要. 此处仅从同名变量的角度解释, 考虑以下例子:

$$t = \lambda x.(\lambda x.x)(x x)$$

我们知道, $FV(t) = \emptyset$, 这个闭项出现了 x 与 x , 分别是外层函数绑定与内层函数绑定的. 语义上, 我们使用静态作用域来解释, 即每个函数的参数都会创建一个新的作用域, 这个作用域到函数体结束时自动销毁. 使用闭项的语言是, $(x x)$ 相对于 $\lambda x.x$ 不是封闭的.

我们有一个显然的引理:

引理 1.1

闭项 s 相对于任意词项 t 封闭.

证明 s 是闭项, 所以 $FV(s) = \emptyset$, 对于任意词项 t , $\emptyset \cap BV(t) = \emptyset$ 成立.

1.3 等式语义 Equation Semantics

在讨论 λ 演算的语义——词项作为程序 (program) 的运行——之前, 我们有必要明晰一下语义等价 (semantic equivalence) 这一概念, 而与之相对的是定义等价 (definitional equivalence).

定义等价 $=$, 又称语法等价, 是指在定义语法之时就确定的等价, 或者称其为形式等价, 即两个词项具有相同的形式. 比如 x 与 x 定义等价, 又如 $\lambda x.x$ 与 $\lambda x.x$ 定义等价. 我们说形式等价就是在重复同一个东西, 即承认同一律 $A = A$. 此外, 我们省略括号的记法就是一种定义等价, 这是我们在定义语法的时候就约定的结合性与优先级.

语义等价是指两个词项的语义相同, 即程序有相同的运行效果. 比如 $\lambda x.x$ 与 $\lambda y.y$, 他们只是参数名字不同罢了, 他们拥有相同的运行效果.

我们着重提出 λ 演算的三种语义等价, 分别记为 $=_\alpha, =_\beta, =_\gamma$.

α 等价 $=_\alpha$ 就是我们前面提到的参数名字不同但语义相同的那个等价: $\lambda x.x =_\alpha \lambda y.y$. 这个语义等价最根本, 因此使用第 1 个希腊字母 α , 我们也称其为换名 (renaming). 结合刚才所说的作用域, 我们通过换名来解除歧义, 比如 $\lambda x.\lambda x.x =_\alpha \lambda x.\lambda y.y$, 又如 $\lambda x.(\lambda x.x)(x x) =_\alpha \lambda y.(\lambda x.x)(y y)$

γ 等价 $=_\gamma$ 相对来说用得少, 因而排序上只获得了第 3 个希腊字母 γ . 他的定义是:

定义 1.5 (γ 等价, γ -equivalence)

γ 等价 $=_\gamma$ 如下定义:

1. 对于任意的参数 x 与任意的词项 t' , $\lambda y.(\lambda x.t') y =_\gamma \lambda x.t'$
2. 如果 $t' =_\gamma t''$, 那么对于任意的参数 x , $\lambda x.t' =_\gamma \lambda x.t''$
3. 如果 $t_1 =_\gamma t'_1$ 且 $t_2 =_\gamma t'_2$, 那么 $t_1 t_2 =_\gamma t'_1 t'_2$

对于 λ 演算这种抽象语言, 在忽略运行效率的情况下, 函数多套一层函数的壳得到的词项等价于它本身. 从这里的定义中, 我们可以注意到:

- 条件 2 与条件 3 是子结构传递性质, 在之后的章节中, 我们会看到这是归纳定义等价关系的方法.

函数 $\lambda x.t'$ 的子结构 t' , 调用 $t_1 t_2$ 的子结构 t_1, t_2 的等价性质均会传递给父结构.

β 等价 $=_\beta$ 引入函数调用的语义, 他次于 α 等价, 因此使用第 2 个希腊字母 β , 我们也称其为化简 (reduction):

定义 1.6 (β 等价, $[\beta]$ 化简, β -equivalence, $[\beta]$ -reduction)

β 等价 $=_\beta$ 如下定义:

1. 对于任意的参数 x 与任意的词项 t'_1, t_2 , 如果 t_2 相对于 t'_1 封闭, 那么 $(\lambda x.t'_1)t_2 =_\beta t'_1 \frac{t_2}{x}$
2. 如果 $t' =_\beta t''$, 那么对于任意的参数 x , $\lambda x.t' =_\beta \lambda x.t''$
3. 如果 $t_1 =_\beta t'_1$ 且 $t_2 =_\beta t'_2$, 那么 $t_1 t_2 =_\beta t'_1 t'_2$

我们在这里引入了一个新的记号 $t_1 \frac{t_2}{x}$, 在说明这个记号实际所指之前, 我们先来补充一件事, 即换名的定义:

定义 1.7 (α 等价, 换名, α -equivalence, $[\alpha]$ -renaming)

α 等价 $=_\alpha$ 如下定义:

1. 对于任意的参数 x 与任意的词项 t' , $\lambda x.t' =_\alpha \lambda y.(t' \frac{y}{x})$
2. 如果 $t' =_\alpha t''$, 那么对于任意的参数 x , $\lambda x.t' =_\alpha \lambda x.t''$
3. 如果 $t_1 =_\alpha t'_1$ 且 $t_2 =_\alpha t'_2$, 那么 $t_1 t_2 =_\alpha t'_1 t'_2$

也就是说, 记号 $t_1 \frac{t_2}{x}$ 同时定义了 α 等价与 β 等价, 这一记号称为代换 (substitution). 不严格地说, $t_1 \frac{t_2}{x}$ 将 t_1 中出现的 x 换成 y . 严格来说, 我们还要考虑作用域对变量的影响. 我们立刻定义代换, 注意这里的 $=$ 是定义等同:

定义 1.8 (代换, substitution)

$t_1 \frac{t_2}{x}$ 表示将 t_1 中的 x 代换为 t_2 , 如下定义:

$$x \frac{t}{y} = \text{如果 } x = y \text{ 那么 } t \text{ 否则 } x \quad (\text{变量})$$

$$\lambda x. t' \frac{t}{y} = \text{如果 } x = y \text{ 那么 } \lambda x. t' \text{ 否则 } \lambda x. (t' \frac{t}{y}) \quad (\text{函数})$$

$$(t_1 t_2) \frac{t}{y} = (t_1 \frac{t}{y}) (t_2 \frac{t}{y}) \quad (\text{调用})$$



最后, 我们来解释一下记号 $t_1 \frac{t_2}{x}$. 我们可以形象地把代换看成约分, 如下例中 \square 内所写一般:

$$x \frac{y}{x} = [\cancel{x} \frac{y}{\cancel{x}}] = y$$

此时, 我们强调 $=_\beta$ ^{1.6} 第 1 个条件中的相对封闭要求, 下面给出一个例子说明不相对封闭会带来错误.

$$\begin{aligned} \lambda y. \lambda z. (\lambda x. \lambda y. x) y z \\ =_\beta \lambda y. \lambda z. (\lambda y. y) z =_\beta \lambda y. \lambda z. z \end{aligned} \quad (\text{错误})$$

$$\begin{aligned} \lambda y. \lambda z. (\lambda x. \lambda y. x) y z \\ =_\alpha \lambda w. \lambda z. (\lambda x. \lambda y. x) w z =_\beta \lambda w. \lambda z. w =_\alpha \lambda y. \lambda z. y \end{aligned} \quad (\text{正确})$$

直觉上 $\lambda y. \lambda z. y \equiv \lambda y. \lambda z. z$ 不可接受. 注意 (错误) 进行 β 代换时, 忽视了相对封闭性, 即 y 相对于 $\lambda y. x$ 不封闭. 根据引理 1.1, 我们可以放心地将闭项 s 作为实际参数来调用函数.

接下来我们就可以定义等式语义了.

定义 1.9 (等式语义, equation semantics)

$t \equiv t'$ 表示 t 语义等价 t' , 是满足下列条件其一的自反对称传递闭包:

- $t =_\alpha t'$
- $t =_\beta t'$
- $t =_\gamma t'$

在上下文不引起歧义的情况下, 我们也可以用 $=$ 代替 \equiv .



1.4 布尔值 Boolean

前文一直提及两个二元函数, 即:

定义 1.10 (布尔值, boolean)

- $T = p_1 = \lambda x. \lambda y. x$
- $F = p_2 = \lambda x. \lambda y. y$

其中 T 表示真 (true), F 表示假 (false), p 表示挑选 (pick).



从前文中, 我们对等式语义已经充分熟悉了, 有下列命题:

命题 1.1

设 s_1, s_2 是两个任意的闭项, 那么有:

1. $T s_1 s_2 = p_1 s_1 s_2 = s_1$
2. $F s_1 s_2 = p_2 s_1 s_2 = s_2$



证明


$$\begin{aligned}
T \ s_1 \ s_2 &= p_1 \ s_1 \ s_2 \\
&= (\lambda x. \lambda y. x) \ s_1 \ s_2 && \text{(挑选 } s_1\text{)} \\
&= (\lambda y. s_1) \ s_2 \\
&= s_1 \\
F \ s_1 \ s_2 &= p_2 \ s_1 \ s_2 \\
&= (\lambda x. \lambda y. y) \ s_1 \ s_2 && \text{(挑选 } s_2\text{)} \\
&= (\lambda y. y) \ s_2 \\
&= s_2
\end{aligned}$$

我们通过这个命题, 可以得知, $T = p_1$ 挑选两个参数中的第 1 个, $F = p_2$ 挑选两个参数中的第 2 个. 为什么我们说他们就是真和假呢? 我们知道, 布尔值出现的一个重要原因, 就是要做 if 语句:

if(b) then{s1} else{s2}

当 $b=T$ 时, 我们执行 s_1 ; 当 $b=F$ 时, 我们执行 s_2 . 由上述命题, T 要挑选第 1 个, F 要挑选第 2 个, 满足要求.

我们说, p_1, p_2 的语义建模了布尔值.

 **笔记** 建模指的是将一个概念在某个系统内表达, 也称将概念嵌入系统, 在此处, 概念是布尔值, 而系统是 λ 演算. 我们要直观感受的图灵等价, 就是在 λ 演算内建模图灵机所能进行的运算.

既然建模了布尔值, 我们乘胜追击, 将布尔值的运算进行建模:

定义 1.11 (布尔运算, boolean operation)

$$\begin{aligned}
\& &= \lambda g. \lambda h. g \ h \ F && \text{(布尔与)} \\
| &= \lambda g. \lambda h. g \ T \ h && \text{(布尔或)} \\
! &= \lambda g. g \ F \ T && \text{(布尔非)}
\end{aligned}$$

命题 1.2

上述定义的布尔运算满足对应的真值表.

证明 我们只验证布尔与, 布尔或同布尔非的验证留作练习.

$$\begin{aligned}
 & \& T T \\
 & = (\lambda g. \lambda h. g \ h \ F) \ T \ T \\
 & = T \ T \ F & \text{(真与真)} \\
 & = T \\
 & \& T F \\
 & = (\lambda g. \lambda h. g \ h \ F) \ T \ F \\
 & = T \ F \ F & \text{(真与假)} \\
 & = F \\
 & \& F T \\
 & = (\lambda g. \lambda h. g \ h \ F) \ F \ T \\
 & = F \ T \ F & \text{(假与真)} \\
 & = F \\
 & \& F F \\
 & = (\lambda g. \lambda h. g \ h \ F) \ F \ F \\
 & = F \ F \ F & \text{(假与假)} \\
 & = F
 \end{aligned}$$

这意味着, 我们将布尔值及其运算彻底嵌入 λ 演算中.

另外, 为了之后定义的方便, 我们引入记号:

定义 1.12

假设 x_1, \dots, x_n 两两不等, $t = \lambda x_1. \dots \lambda x_n. t'$ 可以简记为 $t \ x_1 \ \dots \ x_n = t'$.



命题 1.3

假设 x_1, \dots, x_n 两两不等,

$t \ x_1 \ \dots \ x_n = t'$, 任意 m 满足 $0 \leq m \leq n$, 任意闭项 s_1, \dots, s_m , 都有 $(t \ s_1 \ \dots \ s_m) \ x_{m+1} \ \dots \ x_n = t' \ \frac{s_1}{x_1} \ \dots \ \frac{s_m}{x_m}$;

任意 m 满足 $0 \leq m \leq n$, 若 $\{x_1, \dots, x_m\} \cap BV(t') = \emptyset$, 则有 $(t \ x_1 \ \dots \ x_m) = \lambda x_{m+1}. \dots \lambda x_n. t'$;

特别地, 当 $m = n$ 时, $t \ x_1 \ \dots \ x_n = t'$, 这是简记的原因.



证明 前半命题是显然的. 后半命题只需证明任意 k 满足 $1 \leq k \leq m$, x_k 相对 $t_k = \lambda x_{k+1}. \dots \lambda x_n. t'$ 封闭:

$FV(x_k) \cap BV(t_k) = \{x_k\} \cap (\{x_{k+1}, \dots, x_n\} \cup BV(t')) = \{x_k\} \cap BV(t') = \emptyset$, 因为 $\{x_1, \dots, x_k, \dots, x_m\} \cap BV(t') = \emptyset$.

例如, 布尔值及其运算的定义可以改写为:

$$\begin{aligned}
 T \ x \ y &= x & \text{(布尔真)} \\
 F \ x \ y &= y & \text{(布尔假)} \\
 \& \ g \ h &= g \ h \ F & \text{(布尔与)} \\
 | \ g \ h &= g \ T \ h & \text{(布尔或)} \\
 ! \ g &= g \ F \ T & \text{(布尔非)}
 \end{aligned}$$

一位布尔值的运算还很弱, 但如果我们能做多位布尔值呢? 这就需要笛卡尔积.

1.5 元组 Tuple

p_1, p_2 起到挑选的功能, 这不仅可以用于建模布尔值, 还可以用于建模笛卡尔积的投影函数:

定义 1.13 (笛卡尔积, Cartesian product)

定义笛卡尔积构造子 \times 为: $\times g h = \lambda p. p g h$

任意两个闭项 s_1, s_2 的笛卡尔积 $(s_1, s_2) = \times s_1 s_2$.



注意区分笛卡尔积 (s_1, s_2) 与表示优先调用的 $(s_1 s_2)$, 不要混淆.

命题 1.4

对于任意两个闭项 s_1, s_2 的笛卡尔积 $(s_1, s_2) = \lambda p. p s_1 s_2$, 且满足:

$$(s_1, s_2)p_1 = s_1 \quad (\text{投影 1})$$

$$(s_1, s_2)p_2 = s_2 \quad (\text{投影 2})$$



证明

$$(s_1, s_2) = \times s_1 s_2 = (\lambda p. p g h) \frac{s_1}{g} \frac{s_2}{h} = \lambda p. p s_1 s_2;$$

$$(s_1, s_2) p_1 = p_1 s_1 s_2 = s_1; (s_2, s_2) p_2 = p_2 s_1 s_2 = s_2.$$

笛卡尔积是一个期待投影函数 p_1 或 p_2 作为参数的函数, 将两个闭项锁在函数体等待投影函数.

进一步地, 我们可以定义 n 元挑选函数 $[m]_n (1 \leq i \leq n)$, 我们也可以定义 n 元笛卡尔积, 也称元组 (tuple):

定义 1.14 (元组 tuple)

n 元组 $(x_1, \dots, x_n) p = p x_1 \dots x_n$;

n 元挑选函数 $[m]_n x_1 \dots x_n = x_i$, 不引起歧义时, 可以简写为 $[m]$.



命题 1.5

$$(x_1, \dots, x_n)[m] = x_m.$$



证明留作练习.

有了元组之后, 我们可以建模数字逻辑电路中的组合逻辑, 有兴趣的读者可以自行玩耍, 此略.

1.6 递归 Recursion

组合逻辑有了, 那么时序逻辑呢? 时序逻辑的关键在循环, 且每次执行循环体内部状态 (变量) 可能发生变化. 而 λ 演算中没有可变量, 我们唯有借助与循环等价的递归, 只要将那些内部状态作为递归函数的参数, 递归调用自身时传入改变后的值即可.


在许多非函数式风格的语言里, λ 表达式指的是匿名函数. 匿名函数, 没有名字, 如何调用自己呢? 如果用我们习惯的 $f x = t$ 记号, 我们当然有名字 f , 可是这并不对劲, 因为 $f = \lambda x. t$, 这只会使得 $f \in FV(t)$. 为了寻找它的名字, 我们需要一个辅助函数 $g f x = t$, 这样, $f \in BV(\lambda f. \lambda x. t)$ 就存在了.

让我们以 32 位有符号整数的乘法为例, 假设已经通过组合逻辑定义了加法 $x + y$, 减法 $x - y$ 与零判断 $? x$, 且零判断返回 T 或 F.

$$g f x y = (? x) 0 (y + (f (x - 1) y))$$

当 $x = 0$ 时, 函数 $(g f)$ 会退出递归, 此时无论 f 是什么, $(g f) 0 y = 0$ 正确. 当 $x \neq 0$ 时, 根据递归的经验, 我们希望 $f = g f$, 就会保证递归函数总正确. 满足这样条件的 f 称为 g 的不动点.


定义 1.15 (不动点, fixpoint)

设函数 $g = \lambda f.t$, 词项 f 称为 g 的一个不动点, 记作 $f : \text{fix } g$, 当且仅当 $f \equiv g f =_{\beta} t$, 其中 \equiv 表示等式语义. 

到此处, 我们似乎没有办法继续走下去了.

递归和循环是等价的, 递归走不通, 不妨试试从循环获取经验. 从简单的死循环入手:


定义 1.16 ([死] 循环, [endless] loop)

称词项 t 为 [死] 循环, 当且仅当 $t =_{\beta} t$. 


注意 β 化简即没有自反性, 也没有对称性, $t =_{\beta} t'$ 意味着 t' 相较 t 一定发生了代换 $t_1 \frac{t_2}{x}$, 但 t' 却与代换前的 t 相同, 这就是循环的意味. 若循环存在, 其需要 β 化简, 则必含有调用.

这样的循环存在吗? 我们直接给出:

定义 1.17 (ω 组合子, ω combinator)

$$\omega x = x x$$


命题 1.6

$\omega \omega$ 是循环, 即 $\omega \omega =_{\beta} \omega \omega$. 


证明 以下的 $=$ 表示定义等同.

$$\begin{aligned} \omega \omega &= (\lambda x. x x) \omega \\ &=_{\beta} (x x) \frac{\omega}{x} \\ &= \omega \omega \end{aligned}$$

我们发现 $x x$ 是产生循环的关键原因, 他可以将自己重复两次.

$t =_{\beta} t$ 与 $f \equiv g f$ 多么相似, 只差一个 g . 仿照 $\omega x = x x$, 有:

引理 1.2


令 g 为自由变量, $h x = g (x x)$, 则 $h h =_{\beta} g (h h)$. 

证明 以下的 $=$ 表示定义等同.


$$\begin{aligned} h h &= (\lambda x. = g (x x)) h \\ &=_{\beta} (g (x x)) \frac{h}{x} \\ &= g (h h) \end{aligned}$$

我们距离不动点只差一步了.

定义 1.18 (Y 组合子, Y combinator)

$$Y g = h h = (\lambda x. g (x x)) (\lambda x. g (x x))$$


定理 1.1

设函数 g , $Y g : \text{fix } g$, 即 $Y g \equiv g(Y g)$, 其中 \equiv 表示等式语义. 

证明 以下 $=$ 表示定义等同, \equiv 表示等式语义. $Y g \equiv g(Y g)$ 是因为:

$$\begin{aligned} Y g & \\ &= h h \\ &=_{\beta} g (h h) \\ &= g (Y g) \end{aligned}$$

我们仅举一个例子来说明这样定义的不动点确实有效.

命题 1.7 (2*2=4)

设 $g f x y = (? x) 0 (y + (f (x - 1) y))$, 令 $f = Y g$, 则 $f 2 2 = 4$.



证明

$$\begin{aligned} f 2 2 & \\ &= Y g 2 2 \\ &= g (Y g) 2 2 \\ &= (? 2) 0 (2 + (Y g (2 - 1) 2)) \\ &= 2 + (Y g (2 - 1) 2) \\ &= 2 + (Y g 1 2) \\ &= 2 + (g (Y g) 1 2) \\ &= 2 + (? 1) 0 (2 + (Y g (1 - 1) 2)) \\ &= 2 + (2 + (Y g 0 2)) \\ &= 2 + (2 + (g (Y g) 0 2)) \\ &= 2 + (2 + (? 0) 0 (2 + (Y g (0 - 1) 2))) \\ &= 2 + (2 + 0) \\ &= 4 \end{aligned}$$

这是尾递归, 却没有进行尾递归优化, 下面给出优化的乘法函数:

命题 1.8 (2*2=4)

设 $g f r x y = (? x) r (f (r + y) (x - 1) y)$, 令 $f = Y g 0$, 则 $f 2 2 = 4$.



证明留作练习.

1.7 列表 List

编程中常常会遇到这么一个问题, $2147483647 + 1 = 0$, 32 位带符号整数运算溢出了.

32 位是固定不变的, 即元组的元数被事先确定了, 我们期望能够有按需改变元数的“笛卡尔积”, 即列表 (list).

定义 1.19 (列表, list)

长度为 n 的列表 $[x_0, \dots, x_{n-1}] = (x_0, [x_1, \dots, x_{n-1}])$; 特别地, 闭项 $[]$ 称为空列表; 我们用元变量 l 表示列表.



上述定义是递归定义的, 或者说归纳定义, 归纳一词来自于 [数学] 归纳法, 以后我们将经常进行归纳定义. 结合例子可能更好理解: $[w, x] = (w, (x, []))$; $[w, x, y] = (w, (x, (y, [])))$; $[w, x, y, z] = (w, (x, (y, (z, []))))$.

实际上, 列表是单链表 (single linked list):

定义 1.20 (列表操作, list operation)

头部添加元素: $\text{add } l \ x = (x, l)$;

删除头部元素: $\text{del } l = l \ p_2$;

获取头部元素: $\text{top } l = l \ p_1$.



列表长度是不确定的, 或者说是要多长有多长的, 我们称列表的长度是潜无穷 (potential infinity). 正因为列表长度不是确定的某个自然数 n , 我们很难像元组那样定义挑选函数 $[m]_n$. 我们直观的想法是将 l 应用到 p_2 上 m 次, 然后再应用到 p_1 上, 正如我们实现链表那样. 应用 m 次的想法, 引出了自然数的定义. 列表定义时下标从 0 开始计数也是因为 0 是第 0 个自然数.

1.8 自然数 Natural Numbers

定义 1.21 (自然数, natural numbers)

自然数零 $0 \ f \ x = x$, 自然数后继 $S \ n \ f \ x = f \ (n \ f \ x)$.

其中自然数 n 的后继 $S \ n$ 指的是 n 的下一个自然数.



顺带一提, 上述定义自然数的方式称为归纳定义, 每一个归纳定义都会导出一个对应的归纳法, 在这里恰好是数学归纳法, 以后我们会见到其他归纳定义及其归纳法.

自然数 n 在 λ 演算中被建模为二元函数, 其语义是将第一个参数 f 作为函数, 应用到第二个参数 x 上 n 次:

$$\begin{aligned} 1 \ f \ x &= S \ 0 \ f \ x = f \ (0 \ f \ x) = f \ x \\ 2 \ f \ x &= S \ 1 \ f \ x = x \ (1 \ f \ x) = f \ (f \ x) \\ 3 \ f \ x &= S \ 2 \ f \ x = x \ (2 \ f \ x) = f \ (f \ (f \ x)) \\ &\vdots \end{aligned}$$

我们可以定义自然数加法:

定义 1.22

自然数加法 $+ \ m \ n \ f \ x = m \ f \ (n \ f \ x)$.

**命题 1.9 (1+1=2)**

$$+ \ 1 \ 1 = 2$$



证明 $+ \ 1 \ 1 \ f \ x = 1 \ f \ (1 \ f \ x) = 1 \ f \ (f \ x) = f \ (f \ x) = 2 \ f \ x$, 由 γ 等价有 $+ \ 1 \ 1 = 2$.

再一次感受: 对于自然数 n , 函数 $f, n \ f$ 是将 f 应用到参数上 n 次的函数. 来定义乘法:

定义 1.23

自然数乘法 $* \ m \ n \ f \ x = m \ (n \ f \ x)$.



让我们稍加解释, $(n \ f)$ 将 f 应用 n 次, 那么 $m \ (n \ f)$ 将 $(n \ f)$ 应用 m 次, 即将 f 应用 mn 次.

列表的挑选函数呼之欲出:

定义 1.24

对于自然数 m , 列表挑选函数 $[m] \ l = m \ (\lambda l. l \ p_2) \ l \ p_1$, 为增强可读性, 不致歧义的情况下, 可以记为 $l[m]$.



我们先来举一个例子直观感受一下:

$$\begin{aligned}
 & [s_0, s_1, s_2, s_3][2] \\
 &= (\lambda f. \lambda x. f (f x)) (\lambda l. l p_2) (s_0, (s_1, (s_2, s_3))) p_1 \\
 &= (\lambda l. l p_2) ((\lambda l. l p_2) (s_0, (s_1, (s_2, s_3)))) p_1 \\
 &= (\lambda l. l p_2) (s_1, (s_2, s_3)) p_1 \\
 &= (s_2, s_3) p_1 \\
 &= s_2
 \end{aligned}$$

命题 1.10

任意自然数 n , 任意自然数 m 满足 $0 \leq m \leq n-1$, 总有

$$[s_0, \dots, s_{n-1}][m] = s_m$$



证明 固定 n , 只要证明任意自然数 m 满足 $0 \leq m \leq n-1$, $m (\lambda l. l p_2) [s_0, \dots, s_{n-1}] = [s_m, \dots, s_{n-1}]$.

方便起见, 记 $l_m = [s_m, \dots, s_{n-1}]$, 即证明 $m (\lambda l. l p_2) l_0 = l_m$

现对 m 做数学归纳法:

(奠基) 验证 $m = 0$ 的情形, $0 (\lambda l. l p_2) l_0 = l_0$ 成立;

(递推) 假设 $m (\lambda l. l p_2) l_0 = l_m$, 要证 $S m (\lambda l. l p_2) l_0 = l_{m+1}$, 根据定义 $S n f x = f (n f x)$,

$$\begin{aligned}
 & m (\lambda l. l p_2) l_0 \\
 &= (\lambda l. l p_2) (m (\lambda l. l p_2) l_0) \\
 &= (\lambda l. l p_2) l_m \\
 &= l_m p_2 = (s_m, [s_{m+1}, \dots, s_{n-1}]) p_2 \\
 &= [s_{m+1}, \dots, s_{n-1}] = l_{m+1}
 \end{aligned}$$