函数式, 宏, lambda calculus

陈小羽

2019 年 9 月 8 日

1 Background

当我开始使用 LaTeX 的一些比较高级的宏包时, 我发现了一件很不可思议的事情. 在我的认知中, 像 TeX 这样的纯粹用宏编写的系统是做不出来很复杂的逻辑的. 因为宏的执行顺序和人的思考顺序是严重不符合的, 所以通过宏编写复杂的程序是一件很复杂的工作. 并且, 宏的展开只是简单的模板匹配, 缺乏严谨性, 所以很容易写出错误的代码.

然而, 当我开始使用 tikz 这一类宏包的时候, 我被震撼了. tikz/pgf 包内, 实现了表达式求值和复杂的控制流语句. 这里面甚至有 *foreach* 语句

```
\foreach \x in \{1, 2, 3\} \{
    \x
\}
```

当时我在想, 如果我的认识是正确的话, 那么一定存在某种系统性 的方法, 这种方法能够支持有条理的宏编程. 而这种方法恰好是我比较感兴趣的, 所以我对此进行了一些研究. 这项研究进行了大约一周, 在这之间, 我对很多的概念进行了探索, 并找到了他们之间的一些联系.

本文将以 lambda 表达式为线索, 讨论宏和 lambda 表达式, 函数式编程和 lambda 表达式的关系. 在这两个基础上, 再来探究一个系统性的宏程序的实现思想, *Continuation Passing Style* (一般简写为 CPS). 这种思想在多个不同的领域被反复的发现, 不仅使得系统性宏编程成为可能, 还成为了函数式编程语言中处理副作用的一种常用的方法 (又被称为 Monad).

2 lambda 演算

我们首先来定义 lambda 表达式.

define: lambda expression

```
<lambda expression> -> <constant>
                     |  <variable>
                     |  <lambda apply>
                     |  <lambda abstraction>
<constant> -> <number>
<variable> -> x | y | z | ...
<lambda apply> -> <lambda expression> <lambda expression>
<lambda abstraction> -> lambda x . <lambda expression>
```

lambda 演算建立在变量替换的基础之上, 下面我们来定义这个替换过程:

define: substitution

$$
\begin{aligned}
c[y/x] &= c \\
z[y/x] &= z \\
x[y/x] &= y \\
(s\ t)[y/x] &= s[y/x]\ t[y/x] \\
(\lambda x.t)[y/x] &= \lambda x.t \\
(\lambda z.t)[y/x] &= \lambda z.t[y/x], \text{if } z \notin FV(y) \text{ (避免 y 的 Free Vars 被 z Capture)} \\
(\lambda z.t)[y/x] &= \lambda w.t[w/z][y/x], \text{ otherwise, find } w \notin FV(t) \cup FV(y) \\
& \quad w \notin FV(y) \text{: 避免 y 的 Free Vars 被 w Capture} \\
& \quad w \notin FV(t) \text{: 避免 t 的 Free Vars 被 w Capture}
\end{aligned}
$$

可以很容易的发现, 上面的替换过程的定义, 是定义在 lambda 表达式递归结构的基础上, 它涵盖了在替换过程中可能遇到的所有情况.

define: conversions

- α-conversions(变量换名): $\lambda x.t \xrightarrow{\alpha} \lambda y.t[y/x]$. (这里的 y 只能是一个变量或者常数)
- β-conversion(函数调用): $(\lambda x.t)\ y \xrightarrow{\beta} t[y/x]$.
- η-conversion(去调用): $(\lambda x.t)\ x \xrightarrow{\eta} t$ where $x \notin FV(t)$.

不难看出,η-conversion 只是 β-conversion 的一种特殊情况 (貌似在逻辑学中常用). 这里, 从程序的角度上来说, 最值得我们关注的是 β-conversion. 通过反复的对一个式子进行 β-conversion, 我们便可以模拟计算的过程.

TODO: 这里先放一下 lambda 演算, 将具体的比较放到后面

3 lambda 演算和函数式编程

lambda 演算和函数式编程几乎就是同一个东西. 几乎可以讲函数式编程语言理解为一种 "特殊记法" 下的 lambda 演算. 这种 "特殊的记法" , 很多时候都可以使用宏来实现.

3.1 lambda 表达式的结果与求值顺序无关

lambda 表达式有一个很重要的性质 (这个性质听说不太好证明, 是 Church-Rosser theorem 的一个推论)

同一个 lambda 表达式, 通过顺序进行计算 (conversion), 最终如果能够停机的话, 那么这些所有可能的执行结果在 α-conversion 下面是等价的. 因为这个性质, 我们通常使用

$$t = s$$

表示 t 可以通过一些列 conversion 转换为 s, 因为这样的话, 他们最终执行的结果一定是相同的.

回忆: 在 α-conversion 下等价表示我们只允许变量名不同, 所以这些表达式的意义最终都是一样的.

其实, 通过 lambda 表达式, 我们可以构造函数式编程语言, 像 haskell 之类的函数式编程语言中的 lazy evaluation 的概念, 就来自与这里. 因为表达式的执行顺序可以是任意的, 编译器就可以任意的优化其求值顺序, lazy evaluation 指的就是将求求值的时机尽可能的退后. 另外一门函数式语言 Ocaml 则走上了完全相反的道路, Ocaml 采用 eager evaluation, 是能求值, 就先求值的一种语言. 可以说, 不同的函数式编程语言, 因为内核都是 lambda 演算, 所以他们之间的不同往往体现在编译器的求值顺序上.

John Harrison 的 «Introduction to Functional Programming» 这本书中, 有一章专门讲了如何通过 lambda 演算来构造一个函数式编程语言. 简单来说:

$$\begin{aligned}
\text{True} &:= \lambda x\ y\ .\ x \\
\text{False} &:= \lambda x\ y\ .\ y \\
(x, y) &:= \lambda f. f x y \\
\text{Fst Pair} &:= \text{P True} \\
\text{Snd Pair} &:= \text{P False} \\
\text{If E Then E1 Else E2} &:= E\ E1\ E2
\end{aligned}$$

当我们有了 If 语句之后, 我们可以通过 If 语句很快的构造出 And, Or, Xor 之类的逻辑运算. 我们甚至可以构造出自然数

$$\begin{aligned}
n &:= \lambda f\ x. f^n\ x \\
\text{SUC } n &:= \lambda n\ f\ x. n\ f\ (f\ x) \\
\text{ISZERO } n &:= n(\lambda x.\text{False})\text{True} \\
m + n &:= \lambda f\ x.\ m\ f\ (n\ f\ x) \\
m \times n &:= \lambda f\ x.\ m\ (n\ f)\ x \\
\text{PREFN} &= \lambda f\ p.\ (\text{False, If Fst P Then Snd P Else } f(\text{Snd P})) \\
\text{PRE } n &= \lambda f\ x.\ \text{Snd(n (PREFN } f) \text{ (True, x))}
\end{aligned}$$

我们甚至还可以构造出递归函数, 下面这个东西叫做 Y 组合子 (是 haskell 这个人发明的)

$$Y := \lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))$$

它有一个性质: 对任意的 F, 都有 $YF = F(YF)$, 于是我们可以设计递归函数. 比如, 如果我们希望计算阶乘可以这样来:

$$\begin{aligned} fact(n) &:= \text{If ISZERO } n \text{ Then 1 Else } n \times fact(n-1) \\ fact &:= \lambda n.\ \text{If ISZERO } n \text{ Then 1 Else } n \times fact(n-1) \\ fact &:= (\lambda f\ n.\ \text{If ISZERO } n \text{ Then 1 Else } n \times f(n-1))\ fact \end{aligned}$$

令

$$F := \lambda f\ n.\ \text{If ISZERO } n \text{ Then 1 Else } n \times f(n-1)$$

则

$$\begin{aligned} fact &= F\ fact \\ Y\ F &= F\ (Y\ F) \end{aligned}$$

所以, 我们只需令 $fact = Y\ F$ 即可实现这个递归函数.

最后一步, 我们还可以很简单的实现 local-binding:

$$\text{Let } x = s \text{ in } t := (\lambda x.\ t)\ s$$

这样之后, 我们的 lambda 演算作为一个函数式编程语言, 可以说是非常完善了. 很多函数式语言, 其实就是在 lambda 演算的基础之上实现了类型. 我们这里讨论的 lambda 演算被称为 untyped-lambda-calculus, 带类型的 lambda 演算被称为 typed-lambda-calculus

这里写的非常的简略, 细节还是要参考 «Introduction to Functional Programming» 这本书才行.

4 Macro 宏与 lambda 演算

不难发现, lambda 演算最深刻, 最核心的部分在其替换过程 (substitution) 的定义上, 其他东西只是形式而不太重要. 很多语言的宏, 在形式和替换过程的定义上, 都和 lambda 演算不同. 虽然都不一样, 但是很相似的一点是: 宏和 lambda 演算都使用 "替换" 这个概念来表达计算的过程.

4.1 C++ 宏

```
#define FOO BAR // FOO --> BAR
#define ID(a) a // ID(a) --> a
#define FUN(a, b) a + b // FUN(a, b) --> a + b
#define CAT(a, b) a ## b // CAT(a, b) --> ab
#define STR(a) #a // STR(a) --> "a"
#define CDR(a, ...) __VA_ARGS__ // CDR(a, ...) --> ...
```

不难发现, 在形式上, C++ 的宏对 lambda 演算做了扩展, 增加了像 `#a`, `a##b`还有 `__VA_ARGS__`这样的语法, 为程序开发增加了便利. 但是在另外一个方面, C++ 的宏执行替换的机制和 lambda 演算不太一样. C++ 的宏的替换是基于模板匹配的, 可以通过如下的规则来的描述:

- 编译器会从左往右扫描所有的 TOKEN, 如果发现这个 TOKEN 和某个宏名相同的话 (比如 `CAT`), 就去匹配这个宏名剩下的括号部分 (比如 `(a,b)`) 和这个 TOKEN 的右边部分. 匹配好之后, 编译器会将整个宏展开称宏定义的样子. 如果匹配不成功, 则编译器考虑下一个 token, 以后都不会考虑这个 TOKEN 了.
- 编译器展开了一个宏之后, 会立即将扫描的指针置于展开的这个串的头部, 重新检查这个新展开的串里面是否有 TOKEN 可以继续展开.
- 在展开一个宏的时候, 编译器会先将所有括号中的式子展开, 然后再来展开这个宏.

比如:

```
#define IF_1(a, b) a
#define IF_0(a, b) b
#define IF(val) IF_ ## val

IF(1) (this, that)
==> IF_1 (this, that)  /* IF(1) --> IF_1 */
==> this  /* IF_1(this, that) --> this */
```

从使用的角度上来看, C++ 的宏比 lambda 演算要更加的灵活. 然而, C++ 的这种基于 pattern matching 的替换策略是不太严谨的.

C++ 的宏的基本操作单位是字符串, 不关心数学意义, 而 lambda 表达式的基本单位是变量, 常数, 等, 是数学元素, 在运算的时候会考虑其数学意义.

```
#define MUL(a, b) a * b
MUL(2 + 3, 4 + 5) ==> 2 + 3 * 4 + 5
```

这种现象产生的原因是: C++ 的宏操作的单位是 TOKEN, 也就是字符串, C++ 编译器不会去考虑这些东西的数学意义. 而 lambda 演算中, 带入的参数只能是变量, 常数, 或者另外一个 lambda 表达式 (相当于会自动加括号, 而添加括号不会影响 lambda 表达式的意义). 比如

$$(\lambda x\ y.x \times y)\ (1+2)\ (4+5) \xrightarrow{\beta} (1+2) \times (4+5)$$

这里必须加上括号, 否则不是一个合法的 lambda 表达式.

C++ 的替换策略是局部的, 而 lambda 演算的替换策略是全局的.

这一点可以通过下面的例子来说明

```
#define ADD(a, b) a + b + x
#define FUN(x) ADD
#define EVAL(...) __VA_ARGS__

EVAL(FUN(y) (a, b)) ==> EVAL(ADD(a, b))
                    ==> ADD(a, b)
                    ==> a + b + x
/* 其实EVAL只是利用C++宏的规则,
   让嵌套的宏能够继续展开下去的一种技巧
   便于理解的话, 上面的展开过程完全可以写成下面这样 */
FUN(y) (a, b) ==> ADD(a, b) ==> a + b + x
```

而这个东西用 lambda 表达式来表示应该是这个样子的:

$$\begin{aligned}(\lambda x.\lambda a\ b.a+b+x)\ y\ a\ b &\xrightarrow{\beta} (\lambda a\ b.a+b+y)\ a\ b \\ &\xrightarrow{\beta} a+b+y\end{aligned}$$

为啥会导致这个问题呢? 因为 C++ 不允许在宏内部定义宏, 导致 ADD(a, b) 中的 x, 不能被 bind 到 FUN 上面, 或者说, 不能被 FUN(x) 捕获到.

C++ 的宏是不卫生的

```
#define X_ADD_Y x + y
#define SUB(z) X_ADD_Y - z

SUB(x) ==> X_ADD_Y - x
       ==> x + y - x
       ==  y
```

可以看到, X_ADD_Y 内部的 x, y 原本都是自由变量, 现在加到了 SUB(z) 下面, 因为 $z \neq x, z \neq y$, 所以 x, y 理应还是自由变量才对, 但是当调用 $SUB(x)$ 时, 因为 $x = x$, 所以外部的 x 捕获了内部 $x + y$ 中的 x, 迫使内部的 x 变成了被 bounded varialbe, 这就导致了问题. 这种问题有个专门术语, 叫做不卫生 (non-hygienic).

和 C++ 形成对比的是, 有些语言中是实现了卫生宏的, 比如 scheme.

```
(let-syntax
    ((insert-binding
      (syntax-rules ()
        ((_ x)
         (let ((a 1))
           (+ x a))))))
  (let ((a 5))
    (insert-binding (+ a 3))))

(let ((a 5))
  (insert-binding (+ a 3)))
==>
(let ((a 5))
  (let ((a:1 1))
    (+ (+ a 3) a:1)))
==>
(+ (+ 5 3) 1)
==>
9
```

容易注意到, 上面这段代码中, 对内部的自由变量 a 做了处理, 使得它没有被外部的 bound-variable a 捕获, 解决了 C++ 中宏的问题.

4.2 宏的结果和求值顺序相关

考虑如下例子

```
#define ADD(a, b) a + b
#define RR )

ADD (1, 2 RR
```

这个例子, 如果先展开 RR, 那么最后就能展开为 1+2, 否则就只能展开为 ADD (1, 2) 然后停下来. 因为这个原因, 所以很多时候, 宏编程非常容易出错, 难以进行系统性的编程.

4.3 Q: 为什么不直接把宏设计成 lambda 演算那样呢?

宏和 lambda 演算之间各有优劣, lambda 演算注重的是计算, 所以它很严谨. 宏注重的是修改自己的这种能力, 所以它在严谨这个层面上做了妥协. 在下一节里面, 我们将继续讨论这个问题.

5 如何系统性的编写宏程序

通常我们会发现自己无法实现复杂的宏程序, 或者很多在其他语言中能够轻易的实现的函数, 难以想到如何简洁的通过宏来实现.

5.1 编写宏程序的时候遇到的障碍

无法保存中间结果 我们在编写 C++ 程序的时候, 经常会有如下的操作

```
int a = func1();
int b = func2(a);
int c = func3(a, b);
int d = func4(a, b, c);
```

我们想要使用之前的程序运行的中间结果. 不幸的是: 宏展开几乎不提供这样的能力. 不仅不能提供这样的能力, 宏甚至连像这样顺序执行的能力都很难提供.

如果我们想把上面的程序用宏来写, 下面是两种 naive 的尝试.

```
#define func1() ret_func1
#define func2(a) ret_func2
#define func3(a, b) ret_func3
#define func4(a, b, c) ret_func4

/* 写法一 */
func4(func1(), func2(func1()), func3(func1(), func2(func1())))

/* 写法二 */
#define a func1()
#define b func2(a)
```

```
#define c func3(a, b)
#define d func4(a, b, c)
```

不过如果程序发生递归, 就显得不太适用了.

有一种最通用的写法是 continuation passing style (CPS), 也是本文的重点.

5.2 CPS

CPS 风格, 要求我们在写所有的程序的时候, 统一使用这样的格式来写:

$$< func, args, cont >$$

其中, $func$ 表示宏名, 或者函数名, $args$ 表示参数列表, $cont$ 是一个回调函数, 表示 $func$ 执行完之后要执行的内容. CPS 要求 $cont$ 也具有 $< func, args, cont >$ 的形式. 有点像尾递归的写法.

宏编程中, 很多时候 cont 不是一开始就在的, 而是在展开 func 的时候动态构造出来的. 通过在执行上一条宏 (语句) 的时候, 生成下一跳需要执行的宏 (语句), 这样就可以编写出能够正常运行的程序.

比如有人通过 CPS, 实现了匿名函数 (又叫 lambda 函数, 和之前我们提到的 lambda 演算不是同一个东西).

```
(define-syntax ??!apply
  (syntax-rules (??!lambda)
    ((_ (??!lambda (bound-var . other-bound-vars) body) oval . other-ovals)
    (letrec-syntax
        ((subs
          (syntax-rules (??! bound-var ??!lambda)
            ((_ val k (??! bound-var)) (appl k val))
            ((_ val k (??!lambda bvars int-body))
             (subs-in-lambda val bvars (k bvars) int-body))
            ((_ val k (x))
             (subs val (recon-pair val k ()) x))
            ((_ val k (x . y))
             (subs val (subsed-cdr val k x) y))
            ((_ val k x)
             (appl k x))))
         (subsed-cdr
          (syntax-rules ()
            ((_ val k x new-y)
             (subs val (recon-pair val k new-y) x))))
         (recon-pair
          (syntax-rules ()
            ((_ val k new-y new-x) (appl k (new-x . new-y)))))
         (subs-in-lambda
          (syntax-rules (bound-var)
            ((_ val () kp int-body)
             (subs val (recon-l kp ()) int-body))
            ((_ val (bound-var . obvars) (k bvars) int-body)
             (appl k (??!lambda bvars int-body)))
            ((_ val (obvar . obvars) kp int-body)
             (subs-in-lambda val obvars kp int-body))))
         (recon-l
          (syntax-rules ()
            ((_ (k bvars) () result)
             (appl k (??!lambda bvars result)))))
         (appl
          (syntax-rules ()
            ((_ (a b c d) result) (a b c d result))
            ((_ (a b c) result) (a b c result))))
         (finish
          (syntax-rules ()
            ((_ () () exp) exp)
            ((_ rem-bvars rem-ovals exps)
             (??!apply (??!lambda rem-bvars exps) . rem-ovals))))
         )
         (subs oval (finish other-bound-vars other-ovals) body)))))
```

实现匿名函数顺便还有一个好处, 就是可以利用函数的参数来模拟变量的赋值. 这个好处一般要通过使用 CPS 才能够实现. 我们可以从下面这一段伪代码中, 感受到这一点:

```
((lambda (x)
  ((lambda (y)
    ((lambda (z) body)
     y + 3))
   x + 2))
 6)
```

上面这段代码相当于下面的代码

```
x = 6;
y = x + 2;
z = y + 3;
body;
```

用宏来实现这个东西的原理很简单, 就是当运行最外侧的 lambda 函数的时候, 将 6 一层层的替换进去, 最终将 body 里面能够替换的 x 全部换成 6. 通过这种思想, 我们就可以模拟函数的传参了.

上面的代码虽然能够模拟赋值语句, 但是陷入了回调地狱. 我们将其做如下改写:

```
(define-syntax CHG
  (syntax-rules ()
    ((_ args func)
     (func args))))

(CHG 6
  (lambda (x)
    (CHG x + 2
      (lambda (y)
        (CHG y + 3
          (lambda (z) body))))))
```

这样就顺眼多了, 通过这种思想, 我们甚至可以直接用宏实现出 let 语句. 关于这个东西更加具体的细节可以参考 Oleg Kiselyov 的文章 «Macros that Compose: Systematic Macro Programming»

5.3 CPS 在函数式编程中的应用 (Monad)

同样是为了模拟赋值语句, haskell 中常用一种叫做 Monad 的数学概念. Monad 的数学定义可以参考 wikibook. 或者

`http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html`

这篇帖子对为什么 Moand 可以模拟赋值语句做了清楚的阐述:

`https://en.wikibooks.org/wiki/Haskell/Understanding_monads`

Haskell 选用 Moand 处理副作用, 还有一个重要的原因是 Monad 中的 (»=) 算符中, 有一个从 m a 到 a 的过渡过程, 所以在实现编译器的时候可以实现顺序. 比如一个用了 monad 的程序长这样

```
type Maybe = Just x | Nothing

father: a -> Maybe a
mather: a -> Maybe a

bothGrandfathers p =
       father p >>=
           (\dad -> father dad >>=
               (\gf1 -> mother p >>=   -- gf1 is only used in the final return
                   (\mom -> father mom >>=
                       (\gf2 -> return (gf1,gf2) ))))
```

这个式子中的 (»=) 算符是编译器内部实现的 (可能是用 C). 所以从 lambda 演算的角度来说, 这个式子已经没办法化简了. 但是从编译器的角度来说, 要想要执行

`(\dad -> ...)`

这个函数, 必须先求出 dad 这个变量的值. 这里面有一个 m a 到 a 的过程是蕴含在`(>>=)`中的, 而 m a 到 a 的这个过程中很多时候被编译器隐藏了很多

有状态的, 不纯洁的代码在其中. 所以 Monad 在这个时候还起到了限制执行顺序的作用. 保证有副作用的代码的执行顺序. haskell 还专门给 monad 提供了一个 do-syntax 来简化 monad 的使用.