民科眼中的 Y 组合子

荀涧林

二〇二三年九月七日

摘要

给过去的我一个小小震撼。

1 前言

Y 组合子（Y Combinator）是 lambda 演算中比较关键的理论。是个“懂的都懂”“不懂看了还是不懂”，更还是“我以为我懂了，实际没懂”，我就是后者。经过这几天小小折腾，感觉我又行了。

因为 lambda 演算中一切皆为函数，没有变量，没有具名函数可以调用，Y 组合子解决了匿名函数的递归调用问题。

2 推荐配置

- racket 基本语法
- lambda 演算基础定义

3 lambda 演算定义

在开始前，我们需要简单讲一下 lambda 演算的定义，它的定义十分简单，也许比你见过的编程语言还要简单。它的定义如下：

$$\begin{aligned}\langle express\rangle &= \langle name\rangle|\langle function\rangle|\langle application\rangle\\ \langle function\rangle &= \lambda\langle name\rangle.\langle express\rangle\\ \langle application\rangle &= \langle expression\rangle\langle expression\rangle\end{aligned}$$

lambda 演算核心就是 $\langle express\rangle$，它可以是普通的名字（$name$），也可以是一个函数定义（$function$），也可以是函数的应用。

lambda 演算的 $name$ 不能表示常规语言（如 C 类语言）上的变量定义，它仅仅是指代某个函数，它相当于是占位符，可以被它所指代的函数完全替换。$function$ 表示如下：

```
(λ (x) x)
```

application 表示如下

```
((λ (x) x) (λ (x) x))
```

这段代码看起来有点不好懂，括号太多了，于是我们用 id 指代它：

```
(define (id x) x)
(id id)
; 与下面式子等价
((λ (x) x) (λ (x) x))
```

4 递归

我们都会递归，例如这么一个求和递归函数。

```
(define (sum n)
  (match n
    [0 0]
    [_ (+ n (sum (sub1 n)))]))
```

可以看到 sum 之中调用了自身 sum。我们知道了 sum 这个函数的名字，所以顺理成章地调用了它，如果我们不知道它的名字，又或者说，它本身就是一个匿名函数，我们还是继续使用递归吗？上面代码变成了：

```
(λ (n)
  (match n
    [0 0]
    [_ (+ n (? (sub1 n)))]))
```

？该填什么？好像什么也做不了。

5 传递自身

因为匿名缘故，我们无法得知自身函数，无法直接调用。但是从函数定义上看，有一个 $\langle name \rangle$ 参数，这个 *name* 是允许直接调用，那么我们能否把自己传进去呢？让它变成：

```
(λ (sum n)
  (match n
    [0 0]
    [_ (+ n (sum (sub1 n)))]))
```

看起来不错，行得通的样子，这样一来，我们需要实现一个函数（先称之为 sum-factor），能接受上面这样的函数：

```
(define (sum-factor f n)
  (match n
    [0 0]
    [_ (+ n (sum-factor f (sub1 n)))]))
```

```
(sum-factor
 (λ (sum n)
   (match n
     [0 0]
     [_ (+ n (sum (sub1 n)))]))
 10)
```

可以看到`(λ (sum n) ... )`与`(sum-factor)`定义十分相近，仅相差一个参数，如果它们本身是一样的话会如何？或者说`(sum-factor sum-factor)`调用自身的话，是不是就能解决这个问题了？

```
  (sum-factor sum-factor 10)
```

结果十分令人震惊！答案竟然是正确的！

6 Y 组合子定义

说了半天，Y 组合子到底是啥？跟上面有啥关系？

我们先看 Y 组合子的定义：

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

可以看到 $\lambda x.f(xx)$ 是重复的，这跟`(sum-factor sum-factor n)`很相像：应用自身，将自身传递下去：

$$\begin{aligned} YR &= (\lambda x.R(xx))(\lambda x.R(xx)) \\ &= R((\lambda x.R(xx))(\lambda x.R(xx))) \end{aligned}$$

可以看到最后一步，R 括号中的值等于 (YR)，也就是 $YR = R(YR)$，往后推导出来 $YR = R(YR) = R(R(YR)) = R...R(YR)$。

于是我们也写一个 racket 版本：

```
(define (Y f n)
  (f f n))
(Y sum-factor 10)
```

结果正确了，但离 Y 的定义有点距离，为了让 sum-factor 往下传递，且不用显式应用自己，我们将 f 重新包装：

```
(define/curry (Y f n)
  (define (g k)
    (f (λ (x) ((k k) x))))
  ((g g) n))
```

我们把 f 包装在 g 里，g 的 k 参数同样指向 g，经过这种变形，我们就可以把 sum-factor 改成：

```
(define/curry (sum-factor f n)
  (match n
    [0 0]
    [_ (+ n (f (sub1 n)))]))

(Y sum-factor 10)
```

f 已经完全指向 sum-factor 了。

7 Fix 不动点

我们再回到 Y 组合子，已知 $YR = R(YR) = ...$，如果我们将它转成 Haskell 代码的话：

```
fix :: (a -> a) -> a
fix f = let x = f x in x
```

这是一个无穷递归调用，$x = fx = f(fx) = f(f(fx)) = ...$，永无止境。如果存在一个 x，使得 $f(x) = x$，我们可以从任意公式中再推回到 x：

$$
\begin{aligned}
f(f(fx)) &= f(f(x')) && x' = x \\
&= f(f(x)) \\
&= f(x') && x' = x \\
&= f(x) \\
&= x
\end{aligned}
$$

一旦确定 x，整条递归的结果也随之确定，这个点称之为不动点，例如上面我们一直使用 sum-factor，它的不动点就是 0：

```
sumFactor :: (Int -> Int) -> Int -> Int
sumFactor _ 0 = 0
sumFactor k n = n + (k (n - 1))

fix sumFactor 10 -- 55
```

8 小结

我们费了许多周折，不仅为了让匿名函数也能递归，同样也该看到函数的无限可能。事实上，lambda 演算已证明跟图灵机等价，既然等价，那么是可以相互换化的，例如循环与递归。

lambda 演算仅靠函数就能实现复杂的逻辑运算，不是很美妙吗？