函数式编程——指令与数据外的可能

PB20000296 郑滕飞

一、引言

1、问题来源

作为大三数院的辅修人，在两年多的学习中接触过诸多的语言，无论是 Python、C/C++、JS，还是更加数学化的 MATLAB，整体的思路都是一致的。但是，有一门语言似乎与之不同：Mathematica。

在 Mathematica 中，绝大部分的操作都是通过函数的嵌套来执行的，而在其中定义[函数]时，并不像计算机的函数，进行一系列操作后得到某些结果，而是更类似一个[表达式]，通过输入得到其输出的计算。不仅如此，Mathematica 的函数一般不存在内部的状态，于是确定的输入一定对应确定的输出。

在执行层面，当 Mathematica 中写下 f[x_,y_]:=x+y 时，也并不关心输入输出的类型与加法是否有机会实现，在实际执行时，相当于直接替换。甚至，当在 f[x_,y_]的右侧进行不符合语法的表达(如 Solve 函数要求两个参数，在右侧直接用单参数调用)时，也并不会出现任何反应，只有调用 f 进行计算时才会报错。

上面这些特点与不同，似乎暗示了 Mathematica 中存在一个本质的替换操作。所有的函数映射都是某种替换，而一切“程序”都可以看作一系列函数嵌套得到一个输出。而这也就是函数式编程的基础思想。

2、摘要

本文会先解释一些函数式编程的基本概念与操作与各语言对函数式编程的支持，并以 Mathematica 为例，实现函数式编程定义的简单算术系统，并用函数式编程的思想编写一些程序，简单探讨函数式编程与过程式的优劣。(事实上，严格意义上来说，Mathematica 并不是纯粹的函数式编程软件，因为它支持过程式编程范式。不过，Mathematica 所使用的语言——Wolframe 语言——的内核是基于规则的函数式编程，因此称呼它为函数式编程也并无不可。)

二、从 λ 演算到函数式编程

1、λ演算

就如同图灵发明的图灵机模型在计算机出现之前，λ演算作为一种计算模型的出现也要远早于以它为基础的语言出现：20 世纪 30 年代，数学家邱奇[Alonso Church]发明了λ演算来研究计算系统中函数的抽象化定义方式，并且以此否定了数学系统的可判定性。

λ演算只包含三个基本规则：

1. x 参数[由于只包含可以表示函数]，不具有特定的语义含义
2. (λx.M) 定义匿名函数，调用它的含义是将项 M 中所有的 x 替换为参数
3. (M N) 定义调用，将 M 中的参数以 N 替换。

而之后更多的应用发展，都是建立在基本规则之上的。例如，如果想定义双参数的函数 f(x,y)=xy (这里不假设任何语义，只作为形式记号)，真正的表达是：λx.(λy.xy)，或者省略括号成为λx.λy.xy。从语义的角度理解，它事实上是将参数 x 映射到了一个函数λy.xy，也就是可以看成参数与函数之间的映射。

(值得注意的是，这套想法的来源是数学中的范畴论。范畴之内存在态射，而函子则描

述范畴到范畴之间的作用。这里对于“函数”的计算，可以看作是态射、函子的具体化。)

在简单介绍后，接下来看一个更加复杂的例子。

2、Y 算子

作为一种计算系统，λ演算必须是图灵完备的，可以理解为能实现选择与递归[循环可以通过递归定义]。选择的实现天然可以通过传入多参数、传出其中一个来实现，但递归的做法则并不平凡。

就以阶乘为例，在函数定义时，如果要实现递归，可以通过如下方式:

```
def fact(n):
  return n ? n*fact(n-1) : 1
```

但是，这种方式在λ演算中无法使用，这是因为想要作如此的递归调用，必须给这个函数一个“名字”fact。而将λ表达式作为函数，注定了无法直接采用调用的方式。

在以下以 fact 作为例子的过程中，假设选择操作与数、基本四则运算都已经定义。

一个简单的想法是，将 fact 成为参数，这样在之后的过程中就可以利用了，也就是

λf.λn(n ? n*f(f,n-1) : 1)

由于刚才所述的分割多参数的操作，可以写为:

λf.λn(n ? n*(f f n-1) : 1)

之后调用参数时传入 f 与 n 就可以解决:

λf.λn(n ? n*(f f n-1) : 1) λf.λn(n ? n*(f f n-1) : 1) n

这里给出一个对 n = 2 的具体计算过程:

```
λf.λn(n ? n*(f f n-1) : 1) λf.λn(n ? n*(f f n-1) : 1) 2
2 ? 2*(λf.λn(n ? n*(f f n-1) : 1) λf.λn(n ? n*(f f n-1) : 1) 1) : 1
2 * (λf.λn(n ? n*(f f n-1) : 1) λf.λn(n ? n*(f f n-1) : 1) 1)
2 * (1 ? (λf.λn(n ? n*(f f n-1) : 1) λf.λn(n ? n*(f f n-1) : 1) 0) : 1)
2 * (λf.λn(n ? n*(f f n-1) : 1) λf.λn(n ? n*(f f n-1) : 1) 0)
2 * (0 ? (λf.λn(n ? n*(f f n-1) : 1) λf.λn(n ? n*(f f n-1) : 1) -1) : 1)
2 * 1
2
```

自然，如果定义 w = λf.(f f)，这个式子可以简化为

(w λf.λn(n ? n*(f f n-1) : 1)) n

但是，在这个形式中，依然出现了 f f 的存在，为了达到直接写出递归的效果，希望能够解决这个 f。于是，着重看内部的部分:

λf.λn(n ? n*(f f n-1) : 1)

可以发现，只需要将(f f)替换成 g 即可，进一步调换顺序后得到

λf.(λg.λn. (n ? n*(g n-1) : 1) (f f))

由于中间部分已经成为了想要的λg.(n ? n*(g n-1) : 1 形式，将其称作 fact，则变成:

λf.fact(f f)

由于想要参数后置，将 fact 替换成 h，还原上述的 w 得到:

(λh.λf.h(f f)) fact) (λh.λf.h(f f)) fact)

这时，只需要把 fact 提到外层就可以结束，由于公共性可以得到上式等价于:

λh.(λf.h(f f) λf.h(f f)) fact

于是，我们最终得到了与递归目标函数无关的 Y 组合子，通常替换符号后写作:

Y = λf.(λx.f(x x) λx.f(x x))

可以发现，利用这个组合子即能将任何递归函数成功定义，也即，要想递归定义 f(x)，只需要按照递归方式写出 rec(f x)，(Y rec)即为想要的函数。

3、函数式编程

刚才的讨论给出了λ演算的定义与一个复杂的操作例子。本节的最后，说一说如何从演算到函数式编程语言。

与图灵演算和现代计算机的冯诺依曼架构密切相关不同，上方的这些基于数学推导的操作并不是那么容易通过编译器实现的，尤其是加入类型后对函数定义、调用形式的自由要求。而又由于"判断两个λ表达式是否相等"已经超出了可计算问题的范畴，作为基本元素的函数相同都是难以判定的。不过，这并不是本文讨论的重点。

严格的函数式语言不存在变量与真正的赋值语句，也没有"状态"的概念，只有不同的名字用于不同的函数规则。这样的好处是保证了引用过程的透明性，也即相同的输入一定对应相同的输出(由于函数不可能存在内部的状态)。

此外，其应当具有惰性求值的特性，也就是在需要用时在计算结果，中间过程不进行计算。这个思想不仅对优化代码有很重要的意义，也能引导出一些新的特性，例如过程可以构建无穷的递归(因为实际上需要用时才会确定递归到哪步结束)，也可以构造无限长的列表。关于函数式编程的更多例子，下面在 Mathematica 中实现，并说明其图灵完备性：

二、编程实现

1、语法简述

Mathematica 中，纯函数有三种定义方式，下划线、Function 与#和&表示的匿名函数，例如下面三句是完全相同的：

(这里涉及:=与=的含义差别，:=可以理解为一定意义的惰性求值，但 Mathematica 并不存在真正的完全惰性)

```
f[x_] := x + 1
f = Function[x, x+1]
f = (# + 1)&
```

在函数式编程的视角，Function 并不是真正的函数，而是一个与λ等同的记号。

函数的调用可以使用中括号，也可以直接写@，但注意其为右结合，如下面三句意思相同：

```
f[x[a]]
f @ x @ a
f @ x[a]
```

而以下两句意思相同，但与上面不同：

```
(f @ x) @ a
f[x][a]
```

事实上，将(λx.M)写作 Function[x,M]，(M N)写作(M @ N)，就差不多得到了 Mathematica 中函数式编程的效果。

由于需要验证无类型函数式编程的图灵完备性，需要实现算术可判断函数与算术可递归函数，这里简单理解为任意多元参数下的条件与递归。此外，为了说明其能表示数据，必须能够表示自然数系统(事实上其在理论上即为递归的基础，因此能实现递归必然能表示，不过还是需要具象化)。下面的程序即一一验证。

2、柯里化

首先，为了能从基本规则出发定义多参数函数，需要先构造到函数的函数，即λx.λy.xy的操作。在此之后只需要先后传入 x、y，就能实现多参数的效果，例如双参数函数实现如下：

```
In[1]:= g[y_] := (y + #) &

In[2]:= g[1]

Out[2]= 1 + #1 &

In[3]:= g[1][2]

Out[3]= 3
```

这里将简单的加法函数拆分成了两步操作：传入 1 得到函数，传入 2 得到最终数值，这个过程即叫做多参数函数的柯里化。

对于更多的参数，构造是完全类似的：

```
In[4]:= f[t_] := Function[m, Function[x, t * m + x]]
                 纯函数          纯函数

In[5]:= f[2]

Out[5]= Function[m$, Function[x$, 2 m$ + x$]]

In[6]:= f[2][3]

Out[6]= Function[x$, 2 × 3 + x$]

In[7]:= f[2][3][5]

Out[7]= 11
```

需要注意的是，如上一段所说，这里的 Function 并不是真正的“双参数函数”，只是代表纯函数的规则，类似λ作为标识符。

当然，原式也可以写成 f = Function[t,Function[m,Function[x,t+m*x]]]，这就和λ演算的原始形式更加接近了。事实上，Mathematica 也自带了将一般函数进行柯里化的函数 Curry[]，具体效果是将多参数函数转化为一个一个调用的形式，也即 t[x,y]与 Curry[t][x][y]一致，而实现的过程理论来说就是进行如上的纯函数转化。

3、条件选择

要实现条件选择，必须先有真假的布尔值。出于函数式编程的思想，真假都需要是一个函数，而且必须是不同的函数。为了避免引入其他运算，一个很自然的想法是：

```
In[12]:= TRUE[x_, y_] := x;
         FALSE[x_, y_] := y;
         {TRUE[1, 0], FALSE[1, 0]}

Out[14]= {1, 0}
```

假设有引入了数，这样的“选第一个”和“选第二个”在数里就可以映射为 1 与 0，与我们熟悉的逻辑表示一致。这样的另一个好处是，条件跳转也会变得非常容易书写(这里的条件跳转要求输入三个表达式，第一个为真采用第二个，否则采用第三个)：

```
In[20]:= IF[bool_, x_, y_] := bool[x, y]
```

(这里有趣的事情是，从这个角度看，函数式编程是不允许不带 else 的 if 的，因为每个表达式都必须返回一个值，不可能第二个分支不存在值。)

当然，布尔值还必须有逻辑运算才能构建出布尔代数，这利用定义也是不难实现的：

```
In[45]:= NOT[x_] := x[FALSE, TRUE];
         {NOT[TRUE], NOT[FALSE]}

Out[46]= {FALSE, TRUE}

In[47]:= OR[x_, y_] := x[TRUE, y];
         AND[x_, y_] := x[y, FALSE];
         {MatrixForm[Table[OR[i, j], {i, 2}, {j, 2}] /. {1 → TRUE, 2 → FALSE}], \
          矩阵格式      表格
           MatrixForm[Table[AND[i, j], {i, 2}, {j, 2}] /. {1 → TRUE, 2 → FALSE}]}
          矩阵格式      表格

Out[49]= {(TRUE  TRUE ), (TRUE  FALSE)}
         (TRUE  FALSE)  (FALSE FALSE)
```

这里的结果都具有很清晰的语义性质，例如 OR 就是，假如第一个条件为真，直接为真，否则返回第二个条件的真假。

利用上面这些结果，就可以在函数式编程中构建逻辑运算和条件跳转。

4、递归

在上一节中已经证明了理论来说用 Y 组合子如何实现递归，不过，在 Mathematica 中实现并不是直接定义这么简单：

```
In[51]:= rec = Function[g, If[# == 0, 1, # * g[# - 1]] &]
           纯函数       如果

Out[51]= Function[g, If[#1 == 0, 1, #1 g[#1 - 1]] &]

In[52]:= Yfail = Function[f, Function[x, f@x@x]@Function[x, f@x@x]]
           纯函数       纯函数                纯函数

Out[52]= Function[f, Function[x, f[x[x]]][Function[x, f[x[x]]]]]

In[53]:= (Yfail@rec)[10]

         ··· $RecursionLimit: Recursion depth of 1024 exceeded during evaluation of Function[g, If[#1 == 0, 1, #1 g[#1 - 1]] &].

Out[53]= Hold[Yfail[rec][10]]
```

这里 Yfail 的定义与上面推导出的 Y 组合子形式完全相同，最后却报了栈溢出。究其原因，是 Yfail[rec]表示一个递归函数，而它的展开本就是无限的过程，只有接收到后面的参数才知道具体展开的结果。

Yfail 的失败，暗示了 Mathematica 并不是完全的惰性求值，也会将中间表达式计算出来。为了在非惰性求值中正常得到结果，需要用一个名为η化(Eta 化)的小技巧，把后面的 f@x@x 替换为 f[Function[y, (x@x)@y]]。乍一看，这似乎在说一句“废话“，x@x 与 Function[y, (x@x)@y]根本就是一个东西。但是，后者在没有接收到 y 时不会自动展开，也就在过程中实现了惰性。最后得到可以运行的版本是：

```
In[54]:= Y = Function[f, Function[x, f@x@x]@Function[x, f[Function[y, (x@x)@y]]]]
           纯函数       纯函数                纯函数          纯函数
```

事实上，利用 Mathematica 的#&表示匿名函数的方式，还可以作进一步的简写：

```
In[56]:= Y0 = Function[f, f@#@# &@Function[x, f[(x@x)@# &]]]
          纯函数                    纯函数

Out[56]= Function[f, (f[#1[#1]] &)[Function[x, f[x[x][#1] &]]]]

In[57]:= (Y0@rec)[10]

Out[57]= 3628800
```

由于左侧两个#和右侧的#嵌套在不同层中，解析时并不会有误，这就得到了相对简洁的最终形式。容易发现，这里不止可以接收 rec，而是可以接收任何一元递归函数。与柯里化类似，多元递归函数可以由一元递归函数表示，从而得到了递归的表示。

5、邱奇数

最后，来看看自然数在函数式编程中的表示方式。

在皮亚诺公理体系中，自然数就是装配了零元素与后继运算且满足某些公理的一个形式系统。一个比较自然的想法是，用恒等表示零，s 表示后继，则 s 嵌套的层数就代表数。也即

```
0 = λx.x
1 = λx.s(x)
2 = λx.s(s(x))
```

但是，这样的表示导致需要显式地定义一个标识符 s，只从 0 出发并不知道下一个数是多少，为了避免这点，将 s 也作为参数：

```
In[111]:= ZERO[s_] := # &;
          ONE[s_] := s@# &;
          TWO[s_] := s@s@# &;
          {ZERO[# + 1 &][0], ONE[# + 1 &][0], TWO[# + 1 &][0]}

Out[114]= {0, 1, 2}
```

这里利用了柯里化的技巧，将其看作 s 映射到函数的函数。想要从这里得到我们印象里的“自然数”，只需要把后继设置成+1，第二个初始参数设置成 0 即可：

但是，我们并不希望每次都多写一遍。而是仍然希望能有一个“后继函数”出现，用来推导到下一个。后继函数的形式也并不复杂：

```
In[123]:= SUC[num_] := Function[s, s@num[s]@# &]
                       纯函数

          THREE = SUC[TWO];
          THREE[# + 1 &][0]

Out[125]= 3
```

事实上，这里只是自然地多复合了一次就得到了后继。从后继出发，构造加法也并不困难，因为它相当于直接将后继中的“+1”变成增加更多：

```
In[166]:= ADD[num1_, num2_] := Function[s, num1[s]@num2[s]@# &];
                               纯函数

          Table[ADD[i, j][# + 1 &][0], {i, 5, 8}, {j, 5, 8}] /. {5 → ZERO, 6 → ONE, 7 → TWO, 8 → THREE}
          表格

Out[167]= {{0, 1, 2, 3}, {1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}}
```

(这里之所以用 5678 去替换，是为了避免+1 中的 1 被替换掉)

更有趣的是乘法，比起加法事实上只需要一个微小的改动：

```
In[168]:= MUL[num1_, num2_] := Function[s, (num1@num2[s])@# &];
                               纯函数
          Table[MUL[i, j][# + 1 &][0], {i, 5, 8}, {j, 5, 8}] /. {5 → ZERO, 6 → ONE, 7 → TWO, 8 → THREE}
          表格
Out[169]= {{0, 0, 0, 0}, {0, 1, 2, 3}, {0, 2, 4, 6}, {0, 3, 6, 9}}
```

由于 num[s]代表 s 的 num 次迭代，加法就是先进行 num2 次，再进行 num1 次，到了乘法，则需要对 num2 进行 num1 次。

于是，装配有加法乘法的自然数形成了，自然数数据类型可以表示，也可以从此出发得到更多的数据类型。而这样得到的自然数，正是以λ演算的发明人邱奇命名的。

三、对比与讨论

1、程序结构分析

在进行了上面的图灵完备分析后，我们最后来看一个简单的函数式编程语言实例——将高斯消去解方程利用函数式编程的思想写出代码。首先看看过程式写出的 C++代码：

```cpp
void gauss_elim(vector<vector<double>>& A)
{
    int n = (int)A.size();
    for (int j = 0; j < n - 1; j++) {
        if (A[j][j] == 0) return;
        for (int k = j + 1; k < n; k++) A[k][j] /= A[j][j];
        for (int k = j + 1; k < n; k++)
            for (int i = j + 1; i < n; i++)
                A[k][i] -= A[k][j] * A[j][i];
    }
}
```

```cpp
void forward_subs1(vector<vector<double>>& L, vector<double>& b)
{
    int n = (int)L.size();
    for (int j = 0; j < n; j++)
        for (int k = j + 1; k < n; k++)
            b[k] -= b[j] * L[k][j];
}
```

```cpp
void back_subs(vector<vector<double>>& U, vector<double>& y)
{
    int n = (int)U.size();
    for (int j = n - 1; j >= 0; j--) {
        y[j] /= U[j][j];
        for (int k = j - 1; k >= 0; k--) y[k] -= y[j] * U[k][j];
    }
}
```

这三个函数分别代表将矩阵 A 作 LU 分解，使得 L 的对角元为 1，并把 L 与 U 存在 A 中。接着，通过前代和回代分别求解两个三角方程组得到方程的解，调用时也就是：

```
gauss_elim(A);
forward_subs1(A, b);
back_subs(A, b);
```

注意到，这里函数模块化的思想是很适合函数式编程的。具体的过程仍然可以分为三个模块。而由于 gauss_elim 最终得到的是形式正确的 A，剩下是计算 b，具体的调用可以写为

```
res = backsubs[gauss_elim[A],forward_subs1[gauss_elim[A],b]]
```

但是，这会导致高斯消元的重复调用，因此最好把第一个参数传回来，方便 backsubs 直接调用。

而每个函数体内的过程，事实上都可以看作若干递归进行嵌套。通过这个方法，即可写出函数式编程的代码。

2、函数式编程化

首先，是高斯消元的构造：

```
In[8]:= e[A_, i_, j_] := A[[i]][[j]];

        ELIM[A_, size_, J_] := Table[If[i > J, If[j > J, e[A, i, j] - e[A, i, J] / e[A, J, J] * e[A, J, j], \
                                表格   如果       如果
                If[j == J, e[A, i, j] / e[A, j, j], e[A, i, j]]], e[A, i, j]], {i, size}, {j, size}];
                如果

In[64]:= GAUSS[A_, size_, now_] := If[now > 0, ELIM[GAUSS[A, size, now - 1], size, now], A];
                                   如果

In[65]:= A = {{3, 1, 1}, {1, 3, 1}, {1, 1, 3}};
          MatrixForm[GAUSS[A, 3, 2]]
          矩阵格式

Out[66]//MatrixForm=
```

$$\begin{pmatrix} 3 & 1 & 1 \\ \frac{1}{3} & \frac{8}{3} & \frac{2}{3} \\ \frac{1}{3} & \frac{1}{4} & \frac{5}{2} \end{pmatrix}$$

这里的整体思路是以 now 进行递归来实现循环[Mathematica 事实上有自带的循环，不过此处为了保证函数式性质不使用]，在循环的每一步作一次消去。消去的细节与 C 语言代码是相同的，不过由于 Mathematica 的数组索引下标是 1 开始，具体的实现有一些细微区别。

实际调用时，size 应该传入 A 的阶数，而 now 初始则为 size 减 1。

高斯消元完成后，剩下的前代法和回代法是相对简单的，不过有几件事需要注意：减法的顺序不同导致的递归方向不同、传参时产生的二维列表的处理与 size、now 初始值的确定方式：

```
In[29]:= FSUBONCE[{A_, b_}, size_, J_] := Table[If[i > J, b[[i]] - b[[J]] * e[A, i, J], b[[i]]], {i, size}]
                                          表格   如果

In[52]:= FSUB1[{A_, b_}, size_, now_] := {A, If[now > 0, FSUBONCE[FSUB1[{A, b}, size, now - 1], size, now], b]}
                                         如果

In[54]:= FSUB1[{A, {x, y, z}}, 3, 2]

Out[54]= {{{3, 1, 1}, {1, 3, 1}, {1, 1, 3}}, {x, -x + y, -y + z}}
```

```
In[79]:= BSUBONCE[{A_, b_}, size_, J_] := Table[If[i ≤ J, If[i == J, b[[i]] / e[A, J, J], \
                                           |表格  |如果      |如果
              b[[i]] - b[[J]] / e[A, J, J] * e[A, i, J]], b[[i]]], {i, size}]

In[86]:= BSUB[{A_, b_}, size_, now_] := {A, If[now ≤ size, BSUBONCE[BSUB[{A, b}, size, now + 1], size, now], b]}
                                            |如果

In[89]:= Simplify[BSUB[{A, {x, y, z}}, 3, 1]]
         |化简
```

Out[89]= $\left\{\{\{3,1,1\},\{1,3,1\},\{1,1,3\}\},\left\{\frac{1}{27}(9x-3y-2z),\ \frac{1}{9}(3y-z),\ \frac{z}{3}\right\}\right\}$

最后，得到最终的函数：

```
In[92]:= MYSOLVE[A_, b_] := BSUB[FSUB1[{GAUSS[A, Length[b], Length[b] - 1], b}, Length[b], Length[b] - 1], Length[b], 1][[2]]
                                            |长度       |长度                  |长度       |长度            |长度

In[93]:= MYSOLVE[{{4, 1, 2}, {2, 5, 2}, {3, 1, 7}}, {7, 1, 16}]

Out[93]= {1, -1, 2}
```

3、结果讨论

对比函数式编程的程序与过程式编程的程序，有如下区别：

a. 涉及循环时，过程式编程所得到的程序的逻辑是更可读的，而函数式编程用递归处理就会显得更加复杂。不过，如果封装顺序递归为循环，也可以更加简化。

b. 函数式编程的程序的编写过程会相对困难，减少的中间状态都要通过嵌套实现，如果不利用 Mathematica 自带的各种强大计算功能(如求解、LU 分解事实上都有自带的)，会更加难以书写

c. 在上面两条可以说是缺点的性质之后，也有显著的优点：其几乎不会出现引起程序中止的错误(在上面的过程中，唯一发生的中断是由于递归无终止)。在区分出函数后，逐个函数进行调试也更加方便。

d. 此外，函数式编程事实上是更加简洁的，利用重复减少(重复部分可以提取为函数)来提升代码效率。

4、总结

这篇文章中，我在学习的过程中用 Mathematica 进行了函数式编程的简单尝试。诚然，这和真正的函数式语言，如 Haskell，还有不小的距离，不过足以窥见函数式编程的一点特性。对于当中的数学理论，其实也有更深的研究，而对于数学中函子的具象化也带来了函数式编程中的重要功能，不过，限于篇幅，此处不再赘述。

作为一种编程范式，函数式编程对代码结构的改进也有不少启示，例如，其中的数据类型实现能带来更多对嵌套的理解，懒求值写法则可以较大幅度提升代码的性能。作为计算数学专业的学生，我期待未来能对此进行更深入的理解与研究。

参考资料

baike.baidu.com/item/λ 演算

blog.csdn.net/Henzox/article/details/70197232

blog.csdn.net/universsky2015/article/details/100891288

www.w3cschool.cn/ytnjs8/ujvf3ozt.html

zhuanlan.zhihu.com/p/190210560

zhuanlan.zhihu.com/p/20616683

zhuanlan.zhihu.com/p/578991429

zhuanlan.zhihu.com/p/57972301