关于 λ 演算的报告

物理学院　林旭辰　1800011324

2020 年 2 月 25 日

摘要

函数式编程是一种应用十分广泛的编程范式，其主要思想是将任务拆分成一系列函数，仅通过输入与输出进行通信，而不依赖于其他的全局数据结构。

这一编程范式背后是 Church 在 1932 年首次提出的 λ 演算系统。虽然该系统当时尚有些许瑕疵，但它仍推进了对决定性问题的研究，并在后世的改进中逐渐完善。

λ 演算的语法较为简单，但仍可以仅通过匿名函数的构建来表达类似自然数或是逻辑运算的概念。本文以此为例子做了一些讨论。

关键词 λ 演算 函数式编程 理论发展 决定性问题

1　函数式编程与 λ 演算

在编程实践中，存在着众多的编程风格（“编程范式”），其中比较著名且被广泛使用的有 • 过程式程序设计 • 面向对象程序设计 • 泛型程序设计 • 函数式程序设计 等等。当然，实际程序设计中都会结合需求综合使用各种风格，很多编程语言也对多种编程范式有所支持。(Stroustrup, 2014)

其中，**函数式编程设计**的主要思想是将待解决的问题拆分为一系列函数。在其理想中，整个问题解决过程仅是各函数间进行输入输出，而没有其他任何不表现为函数返回值的变化（所谓“纯函数”）。函数式编程通过将问题分解为一系列小函数，达到分治的目的。(Kuchling, 2020) 这一“分治”思想对编程实践具有很大的指导作用。著名的以函数式编程为主的语言有 ML 和 Haskell 等。

函数式编程最重要的理论基础是最先由 Church (1932) 提出的 λ **演算**。λ 演算最简单的应用就是所谓匿名函数——Python、C++ 等语言都对此有相应支持。(Kuchling, 2020; Stroustrup, 2014) 此外，将函数也视为对象的思想也是受到 λ 演算的影响。

下面，本文将介绍 λ 演算的发展、其大致内容与语法并讨论一些字面量和逻辑运算如何仅通过函数对象实现。

2 λ 演算的思想发展

λ 运算的引入 Church (1932) 在他的论文中试图通过1. 禁止自由变量使用，并 2. 去除排中律中“若 p 可导出矛盾，则 p 为假”的部分 来改造形式逻辑，以避免诸如罗素悖论的矛盾出现。为表示他的思想，Church 引入了 $\lambda\bullet[\bullet]$ 等一系列符号，并将所有命题用此表示。这是此思想的首次发表。

Church 使用的符号系统与本文使用的较现代的记号有所不同。参见§ 3。

Kleene–Rosser 悖论 Kleene and Rosser (1935) 发现，Church (1932) 的逻辑系统仍然会产生与最原始的形式逻辑类似的矛盾（**Kleene–Rosser 悖论**）。Curry (1941, 1942) 对Kleene and Rosser (1935) 的结果做出改进，得到一个简化版本的悖论（**Curry 悖论**）。

设有函数 m，其对参量 A, B 依次作用后的结果 $((mA)B)$ 等价于 $A \to B$。引入一个函数 $N := \lambda p.((mp)Z)$，并记 N 的不动点为 $X = (\mathsf{Y}N)$（Y 为**不动点算子**）。那么，$(\mathsf{Y}N) = (N(\mathsf{Y}N)); X = (NX) = ((mX)Z)$。

由于 $X \to X$ 恒真，$((mX)X)$ 恒真。做代换，可得 $((mX)((mX)Z))$ 恒真。由公理 $(A \to (A \to B)) \vdash (A \to B)$，可得 $((mX)Z)$。再做代换，可得 X 恒真，从而证得 Z 为真。对于任意的 Z 都可如此论证其为真——这说明原始的 λ 演算体系具有漏洞。(Wikipedia contributors, 2019a)

Curry 悖论的一种解决 一种方案是在 λ 演算体系中引入简单类型。(Church, 1940; Wikipedia contributors, 2020a, 2019b)

在 λ 演算的体系下，所有的变量都是用 λ 符号定义的函数。变量 A 有类型 T 记作 $A : T$。引入简单类型要求在系统中确定几个基本类型，其他所有变量的类型都需要用有限长的基本类型组合表示。递归形式的定义有

$$\text{变量} : \text{参量类型} \to \text{返回值类型}. \tag{1}$$

不动点算子 Y 对任意 g 都要满足 $(\mathsf{Y}g) = (g(\mathsf{Y}g))$，那么我们当然可以把 g 取为 Y，即

$$(\mathsf{YY}) = (\mathsf{Y}(\mathsf{YY})) \tag{2}$$

设 Y 的返回值类型为 x，则 (YY) 的类型为 x。那么，Y 接受 (YY)，输出 (YY)，类型为 $x \to x$。又根据 (YY) 的类型为 x，Y 的类型又可表示为 $(x \to x) \to x$。故有

$$(x \to x) = (x \to x) \to x, \tag{3}$$

这是一个无限自我嵌套的类型，与简单类型这一要求矛盾。所以，在这一系统下，Curry 悖论不会出现。

λ 演算中的不可解决定性问题　**决定性问题**指的是对于某个特定命题，是否存在一个算法，能够对于任意输入给出一个“是”或“否”的答案。一个十分著名的不可解决定性问题是停机问题 (Turing, 1937b)。Church (1936) 在 Turing 之前论证了 λ 演算中一个类似决定性问题的不可行性：无法找到一个通用算法判断两个 λ 表达式是否等价。Turing (1937a) 证明，这个问题与停机问题是等价的。同时，λ 演算系统和图灵机是等价的。

3　λ 演算的符号表示

本节将对 λ 演算的语义语法做简单介绍。(Barendregt and Barendsen, 2000)

非形式化地，**变量**用小写字母表示。λ **项**则或者是一个变量，或者是用“()”括起的一对 λ 项，或者是用“()”括起的“λ”、变量以及一个 λ 项；用大写字母表示。

```
λ-term ::=
variable | `(' λ-term λ-term `)' | `(λ' variable λ-term `)'
```

语义上，两个接连的 λ 项表示后者作为参量传给前者后得到的值；λ 项的第三种形式 λxM 则是定义了一个函数，将变量 x 绑定到 λ 项 M 上。

最外层括号可省略。一些常用的缩写是

$$FM_1 \cdots M_n \equiv (\cdots((FM_1)M_2)\cdots M_n), \tag{4}$$

$$\lambda x_1 \cdots x_n.M \equiv \lambda x_1(\lambda x_2(\cdots(\lambda x_n M)\cdots)). \tag{5}$$

注意到，λ 演算中函数是左结合的。

3.1　用 λ 演算表示逻辑运算

接下来展示一些用 λ 演算表示与或非等逻辑运算的方法并加以解释，以作为演算语法的例子。(Wikipedia contributors, 2020b)

`TRUE` 与 `FALSE`　习惯上，将这对布尔值定义为 `TRUE` := $\lambda xy.x$，`FALSE` := $\lambda xy.y$。可以理解为，若向它们各传入一对参数，那么 `TRUE` 会返回前者，`FALSE` 会返回后者。

个人认为，这个定义的精妙之处在于很好地体现了**排中律思想**。上面定义的 `TRUE` 与 `FALSE` 都是接收两个变量，返回其中一个变量的函数。那么，两个参量必被返回一个，不可能都不被返回或都被返回。这与逻辑值的性质是一致的。

`AND` := $\lambda pq.pqp$　可以将 `AND` 理解为这样一种运算：若前件为真，则返回后件的布尔值；反之返回前件的布尔值。（这正是 Python、C++ 等语言中对逻辑运算的实现方式）假设 p 为真，由上一段对 `TRUE` 的讨论，传给 p 的参数 qp 中的 q 会被返回；反之若 p 为假，第二个 p 会被返回。这个 `AND` 给出了正确结果。

`OR` := $\lambda pq.ppq$　类似的，可以将 `OR` 理解为：若前件为真，则返回前件的布尔值；反之返回后件的布尔值。假设 p 为真，传给 p 的参数 qp 中的 p 会被返回；反之若 p 为假，q 会被返回，给出了正确结果。

`NOT` := $\lambda p.p$ `FALSE TRUE`　由前文讨论，p `FALSE TRUE` 会根据 p 的真假返回后两个变量中的一个。由于设置了这两个变量为 `FALSE`, `TRUE`，函数返回的结果是与 p 的真假相反的。

`IFTHENELSE` := $\lambda pab.pab$　可以将 `NOT` p 理解为 `IFTHENELSE` p `FALSE TRUE`。所以，`IFTHENELSE` 只是将 `NOT` 返回值中的布尔值换成其他值。

此外，还可以在 λ 演算系统中构造等价于于自然数的类，称为 Church 数。(Church, 1936; Turing, 1937a) 其构造方法也是非常巧妙的。特别地，前文中 `FALSE` 的定义是与 0 的 Church 数表示一致的。

4　总结

本人对 λ 演算思想最早的接触是 Python 语言中的 `lambda` 匿名函数。虽然有人对使用 `lambda` 匿名函数表示反对 (Kuchling, 2020)，我确实觉得对于一些只会被使用一次的函数，匿名函数是非常有用且清晰的。

λ 演算系统可以说是“万物皆函数”，将一切用 λ 表达式表达。这一思想比较深刻，甚至可以说颇难以在直观上理解。Python 中可以与之相比的大概就是面向对象的“万物皆 `object`”——但，`Python` 能将函数和其他对象等同地处理也恰说明其对函数式编程是有所支持的。

Church (1932) 最初的想法便是构造一个优美的逻辑系统。他确实做到了，而且这个简洁而不简单的系统对计算机科学产生了深远的影响。这一影响既体现在对决定性问题的研究的贡献，也体现在催生了函数式编程范式以及相关的编程语言。回顾这段历史，看到的是前辈的智慧闪耀的光芒。

参考文献

Henk Barendregt and Erik Barendsen. Introduction to lambda calculus. [Unpublished; online; accessed 24-February-2020], March 2000.

Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, pages 346–366, 1932.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

Haskell B Curry. The paradox of kleene and rosser. *Transactions of the American Mathematical Society*, 50(3):454–516, 1941.

Haskell B Curry. The inconsistency of certain formal logics. *The Journal of Symbolic Logic*, 7(3):115–117, 1942.

Stephen C Kleene and J Barkley Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, pages 630–636, 1935.

Andrew M Kuchling. `Functional Programming HOWTO —Python 3.8.2rc2 documentation`, 2020. URL `https://docs.python.org/3/howto/functional.html`. [Online; version 0.32; accessed 24-February-2020].

Bjarne Stroustrup. *Programming: principles and practice using C++*, pages 528–529, 815–818. Pearson Education, 2014.

Alan M Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163, 1937a.

Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1): 230–265, 1937b.

Wikipedia contributors. `Curry's paradox - Wikipedia, The Free Encyclopedia`, 2019a. URL `https://en.wikipedia.org/w/index.php?title=Curry%27s_paradox&oldid=912968347`. [Online; accessed 24-February-2020].

Wikipedia contributors. `Simply typed lambda calculus - Wikipedia, The Free Encyclopedia`, 2019b. URL `https://en.wikipedia.org/w/index.php?title=Simply_typed_lambda_calculus&oldid=925240360`. [Online; accessed 24-February-2020].

Wikipedia contributors. `Fixed-point combinator - Wikipedia, The Free Encyclopedia`, 2020a. URL `https://en.wikipedia.org/w/index.php?title=Fixed-point_combinator&oldid=940923163`. [Online; accessed 24-February-2020].

Wikipedia contributors. `Lambda calculus - Wikipedia, The Free Encyclopedia`, 2020b. URL `https://en.wikipedia.org/w/index.php?title=Lambda_calculus&oldid=940497171`. [Online; accessed 24-February-2020].