Untyped λ Calculus

Introduction

roife

Beihang University

Dec 2nd 2021

1 Syntax
   - Syntax of λ calculus
   - Free Variables and Substitution

2 Reduction and Evaluation

3 Programming in λ Calculus

4 EOF

什么是 λ?

匿名函数?

Python:

$$\texttt{lambda}\ x : x + 1$$

JavaScript:

$$\texttt{function}(x)\ \{\ x + 1;\ \}$$

λ calculus 是函数式编程语言基础，类似于图灵机，是一个计算模型。

λ terms

$$\lambda x.f(f(x))$$

从上面可以看到一个 term 的组成部分包括：

- λ
- x
- $.$
- $f(f(x))$

Syntax

$$
\begin{aligned}
t ::= & & (\text{项，terms})\\
& x & (\text{变量，variable})\\
& \lambda x.t & (\text{抽象，abstraction})\\
& t\ t & (\text{应用，application})
\end{aligned}
$$

直观理解：

$$
\begin{aligned}
t ::= & & (\text{terms})\\
& x & (\text{variable})\\
& \text{function}(x)\{\ t\ \} & (\text{abstraction})\\
& t(t) & (\text{application})
\end{aligned}
$$

Examples

Abstraction:

$$f(x) = x + 2 \Rightarrow \lambda x.x + 2$$

Application:

$$f(2) \Rightarrow (\lambda x.x + 2)\ 2$$

自由变量 - Free Variables

In $\lambda x.t$, all occurences of x in t are said to be bound. Otherwise, it is said to be free。
The set of free variables of a term t, written $FV(t)$, is defined as follows:

$$
\begin{aligned}
FV(x) &= x \\
FV(\lambda x.t_1) &= FV(t_1) \setminus x \\
FV(t_1\ t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

Examples: $\lambda x.x + y$： y is free.

直观理解：没有被捕获的变量（即非参数的变量）被称为自由变量。

替换 - Substitution

Substitution replaces such free variables with other λ-terms.

$$[\mathrm{x} \mapsto \mathrm{y}]\ \mathrm{t}$$

replaces all free occurences of x in t with y.

Examples:

$$[\mathrm{y} \mapsto \mathrm{z}](\lambda \mathrm{x}.\mathrm{x} + \mathrm{y}) \Rightarrow \lambda \mathrm{x}.\mathrm{x} + \mathrm{z}$$

Exercises

- $[x \mapsto y](\lambda x.x)$
- $[x \mapsto z](\lambda z.x)$
- $[x \mapsto y\ z](\lambda y.x\ y)$

Answers

- $[x \mapsto y](\lambda x.x) = \cancel{\lambda x.y} \quad \lambda x.x$
- $[x \mapsto z](\lambda z.x) = \cancel{\lambda z.z} \quad \lambda z.x$
- $[x \mapsto y\ z](\lambda y.x\ y) = \cancel{\lambda y.y\ z\ y} \quad \lambda w.y\ z\ w$ (alpha-conversion)

Alpha Conversion:

$$\lambda y.t \Leftrightarrow \lambda z.([z \mapsto y]\ t)$$

Substitution - Formal Definition

- $[x \mapsto s]x = s$
- $[x \mapsto s]y = y \qquad \text{if } y \neq x$
- $[x \mapsto s](\lambda y.t_1) = \lambda y.[x \mapsto s]t_1 \qquad \text{if } y \neq x \text{ and } y \notin FV(s)$
- $[x \mapsto s](t_1\ t_2) = [x \mapsto s]t_1\ [x \mapsto s]t_2$

Beta 规约

$$((\lambda x.t)M) \to ([M \mapsto x]\ t)$$

Examples:

$$\begin{aligned}
&\underline{(\lambda x.x)\ ((\lambda x.x)\ (\lambda z.(\lambda x.x)\ z))}\\
\to\ &\underline{(\lambda x.x)\ (\lambda z.(\lambda x.x)\ z)}\\
\to\ &\lambda z.\ \underline{(\lambda x.x)\ z}\\
\to\ &\lambda z.z
\end{aligned}$$

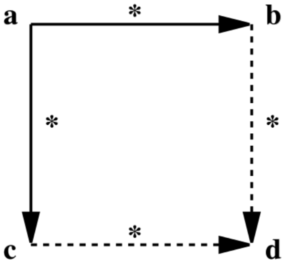
Alpha Conversion + Beta Reduction + λ terms = λ calculus

Church-Rosser Theorem

Church-Rosser Theorem

Will different reduction orders lead to different terms?

Church-Rosser Theorem: λ-calculus is confluent.



Introduction

目前我们看到的 λ 演算中，只有 λ terms。那么如何用 λ terms 表达程序控制结构和数据？

Boole algebra

$$\texttt{tru} = \lambda t.\lambda f.t;$$

$$\texttt{fls} = \lambda t.\lambda f.f;$$

$$\texttt{test} = \lambda l.\lambda m.\lambda n.l\ m\ n;$$

Example for test

Example:

$$
\begin{aligned}
& \mathtt{test}\ \mathtt{tru}\ \mathrm{v}\ \mathrm{w} \\
= \ & \underline{(\lambda \mathrm{l}.\lambda \mathrm{m}.\lambda \mathrm{n}.\mathrm{l}\ \mathrm{m}\ \mathrm{n})\ \mathtt{tru}}\ \mathrm{v}\ \mathrm{w} \\
\rightarrow \ & \underline{(\lambda \mathrm{m}.\lambda \mathrm{n}.\mathtt{tru}\ \mathrm{m}\ \mathrm{n})\ \mathrm{v}}\ \mathrm{w} \\
\rightarrow \ & \underline{(\lambda \mathrm{n}.\mathtt{tru}\ \mathrm{v}\ \mathrm{n})}\ \mathrm{w} \\
\rightarrow \ & \mathtt{tru}\ \mathrm{v}\ \mathrm{w} \\
= \ & \underline{(\lambda \mathrm{t}.\lambda \mathrm{f}.\mathrm{t})\ \mathrm{v}}\ \mathrm{w} \\
\rightarrow \ & \underline{(\lambda \mathrm{f}.\mathrm{v})\ \mathrm{w}} \\
\rightarrow \ & \mathrm{v}
\end{aligned}
$$

and, or, not

$$\text{and} = \lambda b.\lambda c.b\ c\ \texttt{fls};$$

$$\text{or} = \lambda b.\lambda c.b\ \texttt{tru}\ c;$$

$$\text{not} = \lambda b.b\ \texttt{fls}\ \texttt{tru}$$

Pair

$$\texttt{pair} = \lambda f.\lambda s.\lambda b.b\ f\ s;$$

$$\texttt{fst} = \lambda p.p\ \texttt{tru};$$

$$\texttt{snd} = \lambda p.p\ \texttt{fls};$$

直观理解：pair 接受三个参数，其中 f 和 s 分别为组成 pair 的两个元素，b 是一个函数，可以从 pair 中提取出元素（一般被称为 eliminator）。

fst 的作用是将 tru 作为 eliminator 传入，其中 tru 的作用是取第一个元素。因此 fst 实现了提取第一个元素的效果。

Example

$$
\begin{aligned}
& \mathsf{fst}\ (\mathsf{pair}\ \mathrm{v}\ \mathrm{w}) \\
= {} & \mathsf{fst}\ (\lambda \mathrm{b}.\mathrm{b}\ \mathrm{v}\ \mathrm{w}) \\
= {} & (\lambda \mathrm{p}.\ \mathrm{p}\ \mathsf{tru})(\lambda \mathrm{b}.\mathrm{b}\ \mathrm{v}\ \mathrm{w}) \\
\to {} & (\lambda \mathrm{b}.\mathrm{b}\ \mathrm{v}\ \mathrm{w})\ \mathsf{tru} \\
\to {} & \mathsf{tru}\ \mathrm{v}\ \mathrm{w} \\
\to^{*} & \mathrm{v}
\end{aligned}
$$

1 Syntax

2 Reduction and Evaluation

3 Programming in λ Calculus
- Introduction
- Boole algebra
- Cartesian product
- **Church Numerals**
- Substraction in Church Numerals
- Fix-point Operator

4 EOF

Church Numerals

λ 演算中，自然数用函数运算表示。其中，s 和 z 分别代表 succ 和 zero。其意义为递归对于 z 调用 n 次 s，即 $s^n(z)$。

$$
\begin{aligned}
c_0 &= \lambda s.\lambda z.z; \\
c_1 &= \lambda s.\lambda z.s\ z; \\
c_2 &= \lambda s.\lambda z.s\ (s\ z); \\
c_3 &= \lambda s.\lambda z.s\ (s\ (sz));
\end{aligned}
$$

一个有趣的巧合：在 C 语言中，`false` $== 0$。在 church numerals 中，$c_0 ==$ `fls`。

Church Numerals

如何理解 church numerals?

$$
\begin{aligned}
c_0 &= \lambda s.\lambda z.z; \\
c_1 &= \lambda s.\lambda z.s\ z; \\
c_2 &= \lambda s.\lambda z.s\ (s\ z); \\
c_3 &= \lambda s.\lambda z.s\ (s\ (sz));
\end{aligned}
$$

（个人理解）在 λ 演算中，对于数据强调的不是如何存储，而是如何去使用它们。所以 `tru` 和 `fls` 对应了程序的选择结构；自然数对应了程序的归纳结构（类似于有终止的循环或递归）。

Arithmetic in Church Numerals

求后继：（注意，结果还是 $\lambda s.\lambda z.t$）

$$\mathtt{scc} = \lambda n.\lambda s.\lambda z.s\ (n\ s\ z);$$

求和：即 $s^{n+m}(z) = s^n(s^m(z))$

$$\mathtt{plus} = \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z);$$

Exercise

怎么在 λ 演算中表达乘法?

tips：$m * n$ 表示执行 m 次 $+n$

Answer

乘法:

$$
\begin{aligned}
\texttt{times} &= \lambda m.\lambda n.\lambda s.\lambda z.\lambda .m\ (n\ s)\ z; \\
\texttt{times}' &= \lambda m.\lambda n.\lambda s.m\ (n\ s);
\end{aligned}
$$

其中 $(n\ \ s)$ 的基数部分 (z) 接受的是上一次加法的结果，这样调用 m 次，即执行 m 次加法:

$$(n\,s\,(n\,s\,(n\,s\ \ldots)))$$

`times'` 是 `times` 的化简形式（η-conversion）。

η-conversion

$$\lambda x.M\,x \to M$$

直观理解：
左边的 λ term 接受一个参数，然后将这个参数应用于 M，等价于直接将参数应用于 M

Exercise

怎么在 λ 演算中表达减法?

前面介绍的运算都是通过【组合】的方式“构建”。但是减法显然无法通过【组合】实现。

一个巧妙的思路

我们已经知道了如何构建 pair，那么利用 pair 可以构建出下面的“合成”路线：

$$\mathtt{zz} = \underbrace{(0,0) \xrightarrow{ss} (0,1) \xrightarrow{ss} (1,2) \xrightarrow{ss} \cdots \xrightarrow{ss}}_{n \text{ times}} (n-1,n)$$

观察这个路线，发现 pair 第一个元素是上一个 pair 的第二个元素；pair 的第二个元素是上一个 pair 第二个元素加一。

$$\begin{aligned}
\mathtt{zz} \quad &= \mathtt{pair}\ c_0\ c_0; \\
\mathtt{ss} \quad &= \lambda p.\mathtt{pair}\ (\mathtt{snd}\ p)\ (\mathtt{plus}\ c_1\ (\mathtt{snd}\ p)); \\
\mathtt{prd} \quad &= \lambda m.\mathtt{fst}\ (m\ ss\ zz);
\end{aligned}$$

How to construct RECURSIONs

我们知道，church numerals 可以表达指定次数的递归。但是 church numerals 无法表示未知次数的递归。

不难想到，未知次数的递归和不终止的递归是等价的。（未知次数的递归相当于在一个不终止的递归中，判断满足某个条件就跳出）

所以我们先尝试构造不会终止的递归。

Recursion

【不会终止的递归】

首先，它必须是一个可以进行 beta-reduction 的东西（不然不能被求值）

其次，它 reduce 之后的结果要和原来相同

黄金体验镇魂曲 - Divergent Combinator

omega 是一个 divergent combinator：

$$\texttt{omega} = (\lambda x.x\ x)\ (\lambda x.x\ x);$$

虽然 omega 可以进行 reduce，但结果还是一个 omega，永远无法达到「Normal Form」的真实

$$\texttt{omega} \to ([x \mapsto (\lambda x.x\ x)](x\ x)) \to (\lambda x.x\ x)\ (\lambda x.x\ x) = \texttt{omega};$$

Fix-point Combinator

omega 有一个 generalized 的形式，被称为 fixed-point combinator，也叫 call-by-value Y-combinator 或 Applicative-order Y-combinator 或 Z combinator。

$$\texttt{fix} = \lambda f.((\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y)));$$

进行一次 reduce：

$$\texttt{fix} \to \lambda f.f\ (\lambda y.\underline{(\lambda x.f\ (\lambda y.x\ x\ y))}\ \underline{(\lambda x.f\ (\lambda y.x\ x\ y))}\ y); \to (\lambda f.f\ (\lambda y.(\texttt{fix}\ f)\ y));$$

$$\texttt{fix}\ f\ v = f\ (\texttt{fix}\ f)\ v;$$

Other forms of fix-point combinator

上面展示的 Z combinator 是 fix-point combinator 的一种形式，它通过一个参数 v 来阻止 fix-point combinator 在非 lazy 的语言里产生无穷递归。

而在 lazy 的语言里，还有一种更简单的形式：

$$\mathrm{Y} = \lambda \mathrm{f}.(\lambda \mathrm{x}.\mathrm{f}\ (\mathrm{x}\ \mathrm{x}))(\lambda \mathrm{x}.\mathrm{f}\ (\mathrm{x}\ \mathrm{x}))$$

1. Syntax
2. Reduction and Evaluation
3. Programming in λ Calculus
4. EOF

Homework

- 使用 λ 演算构建比较运算符（easy）
- 使用 λ 演算构建阶乘（easy）
- 在 λ 演算中表达幂次（normal）
- 尝试用 λ 演算构建 List（normal）
- 尝试用 λ 演算构建 Binary Tree（normal）
- 想出另一种表达减法的构造（hard）

答案可以从 TaPL 或我的博客（其实间接来源还是 TaPL）中找。

More about λ calculus

其他的和 λ 演算相关但是这次没有提到的主题（可能以后会讲?):

- Equivalence of Turing machine and λ calculus
- Evaluation strategies for λ calculus
- Normal forms in λ calculus
- Operational Semantics for λ calculus
- Simply Typed Lambda Calculus (STLC)
- ……

References

- Benjamin C. Pierce. 2002. Types and Programming Languages.
- `https://lambdacalc.io`

Q & A

欢迎提问

EOF

Thanks