Lambda演算

部分内容修改自南京大学计算机系梁红瑾老师的《程序设计语言的形式语义》课件，感谢梁老师的分享！

这部分的内容可能比上次的容易一点点

提纲

- 函数式编程示例
- Lambda演算
  - 语法
  - 语义
  - 性质
  - 编程

其他形式的计算？

- 通过图灵机的讲解，我们给出了一个可以描述人类计算的模型：通过"变换"0和1来表达计算。
  - 这也启发了我们用"命令"来让机器完成想要做的事情。
- 然而这样的操作过于繁琐，是否存在其他形式的符号变换方式来表达计算，但是计算能力和图灵机等同的模型呢？

函数式（Functional）

函数式编程（Functional programming）

- 难道我们没学过函数吗？

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}
```

函数式编程不是指函数定义和调用

函数式编程（Functional programming）

- 是一种编程范型，其通过函数作用和复合来构造程序。
  - 它是声明式范型（ Declarative programming ）的一种。
  - 与命令式的通过语句串改变状态不同，它使用一系列的函数表达式完成值到值的映射来表达计算。

一等公民（ First-class citizens ）

- 在函数式编程里，函数式一等公民（和其他类型如int、char一样），其可以被命名绑定、可以作为参数传递、作为返回值。

例子

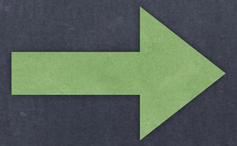
```
// Java program to demonstrate
// anonymous method
import java.util.Arrays;
import java.util.List;
public class GFG {
  public static void main(String[] args)
  {

    // Defining an anonymous method
    Runnable r = new Runnable() {
      public void run()
      {
        System.out.println(
          "Running in Runnable thread");
      }
    };

    r.run();
    System.out.println(
      "Running in main thread");
  }
}
```

As a return

```
// Java 8 program to demonstrate
// a lambda expression
import java.util.Arrays;
import java.util.List;
public class GFG {
  public static void main(String[] args)
  {
    Runnable r
      = ()
      -> System.out.println(
        "Running in Runnable thread");

    r.run();

    System.out.println(
      "Running in main thread");
  }
}
```

例子

```
// Java program to demonstrate an
// external iterator
import java.util.Arrays;
import java.util.List;
public class GFG {
  public static void main(String[]
args)
  {
    List<Integer> numbers
      = Arrays.asList(11, 22, 33, 44,
                      55, 66, 77, 88,
                      99, 100);

    // External iterator, for Each loop
    for (Integer n : numbers) {
      System.out.print(n + " ");
    }
  }
}
```



```
// Java 8 program to demonstrate
// an internal iterator

import java.util.Arrays;
import java.util.List;

public class GFG {
  public static void main(String[] args)
  {
    List<Integer> numbers
      = Arrays.asList(11, 22, 33, 44,
                      55, 66, 77, 88,
                      99, 100);

    // Internal iterator
    numbers.forEach(number
                    -> System.out.print(
                      number + " "));
  }
}
```

As an argument

例子

```java
// Java program to find the sum
// using imperative style of coding
import java.util.Arrays;
import java.util.List;
public class GFG {
  public static void main(String[] args)
  {
    List<Integer> numbers
      = Arrays.asList(11, 22, 33, 44,
                      55, 66, 77, 88,
                      99, 100);

    int result = 0;
    for (Integer n : numbers) {
      if (n % 2 == 0) {
        result += n * 2;
      }
    }
    System.out.println(result);
  }
}
```

repeatedly mutating

functions Composition

```java
// Java program to find the sum
// using declarative style of coding
import java.util.Arrays;
import java.util.List;
public class GFG {
  public static void main(String[] args)
  {
    List<Integer> numbers
      = Arrays.asList(11, 22, 33, 44,
                      55, 66, 77, 88,
                      99, 100);

    System.out.println(
      numbers.stream()
        .filter(number -> number % 2 == 0)
        .mapToInt(e -> e * 2)
        .sum());
  }
}
```

函数式编程的根基

- Lambda演算（Lambda Calculus）！

Lambda演算（Lambda Calculus）

- Lambda演算是由Alonzo Church教授在1930s提出的一个形式系统，可以用来表达计算（能力等价于图灵机）。
- Lambda演算引入了函数抽象、作用、变量绑定和替换等核心概念。

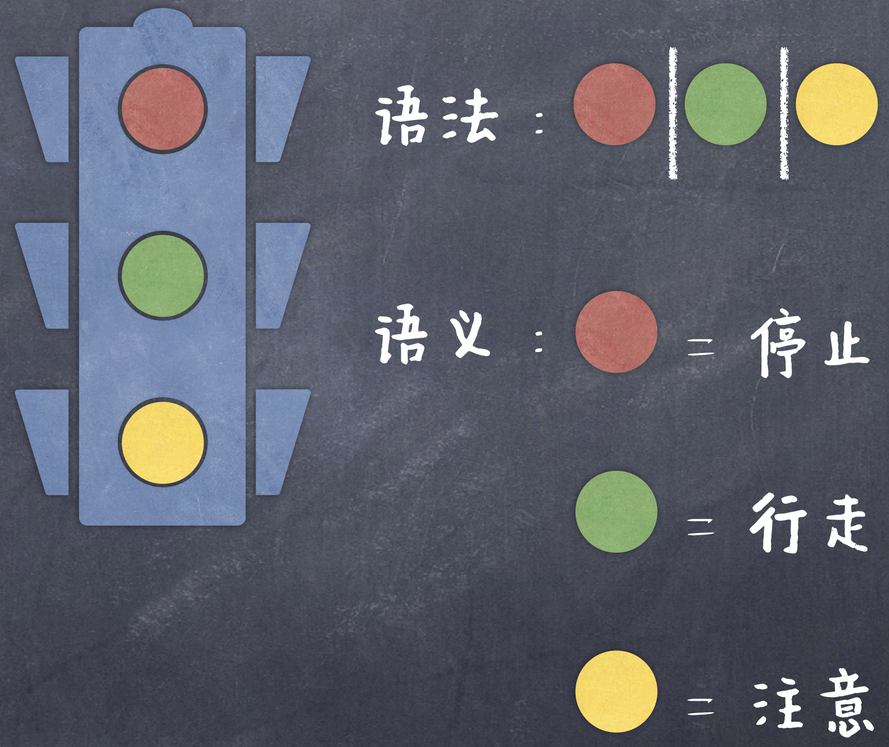
Alonzo Church

Lambda演算的意义

- Lambda演算是函数式编程语言的基础。
  - Lisp, Meta language, Haskell...
  - 除了函数式语言，很多命令式语言也加入了函数特性，如JAVA、 Python
- 并且其往往作为研究程序语言理论的内核。
  - 类型系统（Type system）
  - 作用域和绑定 （Scope and binding）
  - 高阶函数 （Higher-order functions）
  - 。。。

基本定义

- 语法（Syntax）
  - 构成合法程序的写法规则
- 语义（Semantics）
  - 描述所写程序的运行行为



Lambda演算的语法

E.g. BNF for C programming language

- λ项或者λ表达式的BNF范式（Backus-Naur Form）：
  - $M, N ::= x \mid (\lambda x\,.\,M) \mid (MN)$

λ项　变量

John Backus

Peter Naur

Lambda演算的语法

- λ项或者λ表达式的BNF范式（Backus-Naur Form）：
  - $M, N ::= x \mid (\lambda x . M) \mid (MN)$

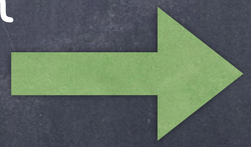
具体解释：给定一个无限的变量集合V，让A为一个字符，其要么是V中的一个元素，要么是"("，要么是")"，要么是"λ"，要么是"."。让A^*为有限的字符串。那么隶属于λ表达式的字符串符$\wedge \in A^*$合如下标准：

1. 如果 $x \in V$，那么 $x \in \wedge$。
2. 如果 $x \in V$且$M \in \wedge$，那么 $(\lambda x . M) \in \wedge$。
3. 如果 $M, N \in \wedge$，那么 $(MN) \in \wedge$。

Lambda演算的语法

- $(\lambda x\,.\,M)$：即为 Lambda抽象（Lambda abstraction）
- 比如：

```
int increment (int x) {
    return x+1;
}
```



$$(\lambda x\,.\,x+1)$$

匿名函数（anonymous），
无法通过函数名进行调用

Lambda演算的语法

- (MN)：即为 Lambda作用（Lambda application）
  - 比如：

```
int increment (int x) {
    return x+1;
}
//...
increment(3);
```

将$(\lambda x\,.\,x+1)$作用于3结果为4

$$((\lambda x\,.\,x+1)3)$$

一些约定（Conventions）

- 一般最外层的括号是省略的。
  - 比如我们一般写MN而不是(MN)
  - 一般写$\lambda x\,.\,M$而不是$(\lambda x\,.\,M)$

一些约定

- λ主体部分($\lambda x\,.\,M$中的M)是尽可能向右延伸的
  - 即$\lambda x\,.\,MN$意味着$\lambda x\,.\,(MN)$，而不是$(\lambda x\,.\,M)N$
  - 比如$\lambda x\,.\,\lambda y\,.\,x - y = \lambda x\,.\,(\lambda y\,.\,x - y)$

一些约定

- Lambda作用是左结合的
  - MNP意味着$(MN)P$，而不是$M(NP)$。
  - 比如$(\lambda x\,.\,\lambda y\,.\,x-y)5\ 3 = ((\lambda x\,.\,\lambda y\,.\,x-y)5)3$

一些约定

- 让我们看一个例子：

$$(\lambda f\,.\,\lambda x\,.\,f\ x)(\lambda x\,.\,x+1)2$$

$$\rightarrow ((\lambda f\,.\,\lambda x\,.\,f\ x)(\lambda x\,.\,x+1))2$$

$$\rightarrow (\lambda x\,.\,(\lambda x'\,.\,x'+1)\ x)2$$

$$\rightarrow (\lambda x\,.\,x+1)2$$

$$\rightarrow 3$$

Apply a function?

高阶函数（High-order functions）

- 函数可以作为返回值返回
  - $\lambda x\,.\,\lambda y\,.\,x - y$
- 函数可以作为参数传递
  - $(\lambda f\,.\,\lambda x\,.\,f\ x)(\lambda x\,.\,x + 1)2$

高阶函数

- 可以做如下类比（函数指针）

```
#include <stdio.h>
#include <stdlib.h>
typedef int (*VAL)(void);
int getNextRandomValue(void) {return rand();}
VAL myFun(){
  return getNextRandomValue ;
}

int main()
{
   printf("myValue: %d\n",myFun());
}
```

函数作为返回值

高阶函数

- 可以做如下类比（函数指针）

函数作为参数

```
#include <stdio.h>
#include <stdlib.h>
void myFun(int (*getNextValue)(void)){
  printf("myValue: %d\n”, getNextValue());
}
int getNextRandomValue(void) {return rand();}
int main()
{
   myFun(getNextRandomValue);
}
```

高阶函数

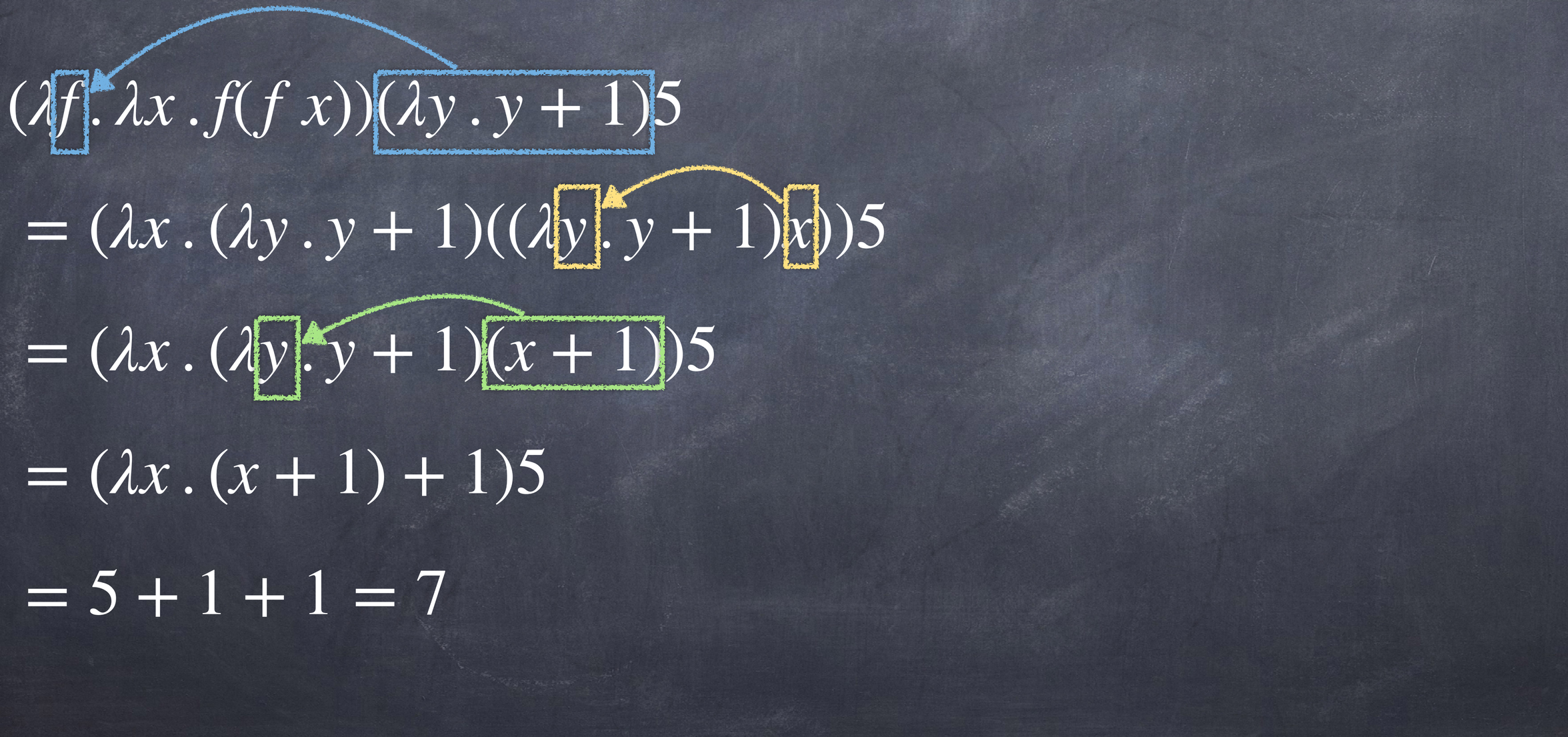
函数复合（composition）

- 给定函数f，返回$f \circ f$
  - $\lambda f . \lambda x . f(f\ x)$

高阶函数

- 练习：
  - $(\lambda f\,.\,\lambda x\,.\,f(f\ x))(\lambda y\,.\,y+1)5$

高阶函数


$(\lambda f . \lambda x . f(f\ x))(\lambda y . y + 1)5$
$= (\lambda x . (\lambda y . y + 1)((\lambda y . y + 1)\ x))5$
$= (\lambda x . (\lambda y . y + 1)(x + 1))5$
$= (\lambda x . (x + 1) + 1)5$
$= 5 + 1 + 1 = 7$

高阶函数

$$(\lambda f\,.\,\lambda x\,.\,f(f\ x))(\lambda y\,.\,y+1)5$$

$$=(\lambda x\,.\,(\lambda y\,.\,y+1)((\lambda y\,.\,y+1)x))5$$

$$=(\lambda x\,.\,(\lambda y\,.\,y+1)(x+1))5$$

$$=(\lambda x\,.\,(x+1)+1)5$$

$$=5+1+1=7$$

柯里化（Curry）

- 注意以下两者区别：
- $\lambda x.\lambda y.x - y$
- `int f(int x, int y) {return x - y;}`
- λ抽象是**单参数**函数！
- 但是两种形式在计算上是等价的！

柯里化（Curry）

- 柯里化
  - 将$\lambda(x, y) . x - y$转化为$\lambda x . \lambda y . x - y$
- 逆柯里化（Uncurry）
  - 逆向转化，即将$\lambda x . \lambda y . x - y$转化为$\lambda(x, y) . x - y$

Haskell Brooks Curry

柯里化（Curry）

- 如果非要用C语言来对比

```c
#include <stdio.h>

typedef int (*one_var_func) (int);

int minus_int (int x, int y) {
    return x-y;
}

one_var_func partial (int x) {
    int g (int y) {
        return x - y ;
    }
    return g;
}

int main (void) {
    int a = 1;
    int b = 2;
    printf ("%d\n", minus_int (a, b));
    printf ("%d\n", partial (a) (b));
}
```

Cannot define functions here!

自由和绑定变量

- 在C语言里，我们经常看到这样的代码

全局变量

```
int y;
…
…
int add(int x) {
    return x + y;
}

x = 0;
```

局部变量

out of scope

自由和绑定变量

- 对应地，在一个λ表达式中，如$\lambda x.x+y$中
  - 紧挨着λ后的就是绑定变量（bound varaible），如上述式子中的x。
  - 除此之外的变量就是自由变量（free varaible），如上述式子中的y。
  - 而$\lambda x.M$中M就是x的绑定域（scope），如上述式子中的$x+y$。

自由和绑定变量

- 绑定的变量可以重命名，而不会改变表达式的语义
  - 比如$\lambda x . x + y$等价于$\lambda z . z + y$
  - 记为$\lambda x . x + y =_\alpha \lambda z . z + y$

α-等价(α-equivalent)

自由和绑定变量

- 自由变量不可重命名！
  - 比如$\lambda x.x+y$不等价于$\lambda x.x+z$

```
int y = 10;
int z = 20;
int add(int x) {  return x + y;  }
```

```
int y = 10;
int z = 20;
int add(int x) {  return x + z;  }
```

```
int y = 10;
int z = 20;
int add(int m) {  return m + y;  }
```

自由和绑定变量

- 出现：
- $(\lambda x.x + y)(x + 1)$ ：x 同时有一个自由 和 绑定的出现

```
int x = 10;
int add(int x) {return x+y;}
add(x+1);
```

对自由和绑定变量的形式化定义

- 回顾一下 $M, N ::= x \mid (\lambda x\,.\,M) \mid (MN)$
- 令 $fv(M)$: 为 M 中的自由变量集合，则
  - $fv(x) = \{x\}$
  - $fv(\lambda x\,.\,M) = fv(M) \backslash \{x\}$
  - $fv(MN) = fv(M) \cup fv(N)$
- 比如
  - $fv((\lambda x\,.\,x)x) = \{x\}$
  - $fv((\lambda x\,.\,x + y)x) = \{x, y\}$

Lambda演算的语义

- Lambda演算语义的核心是β-规约（β-reduction）

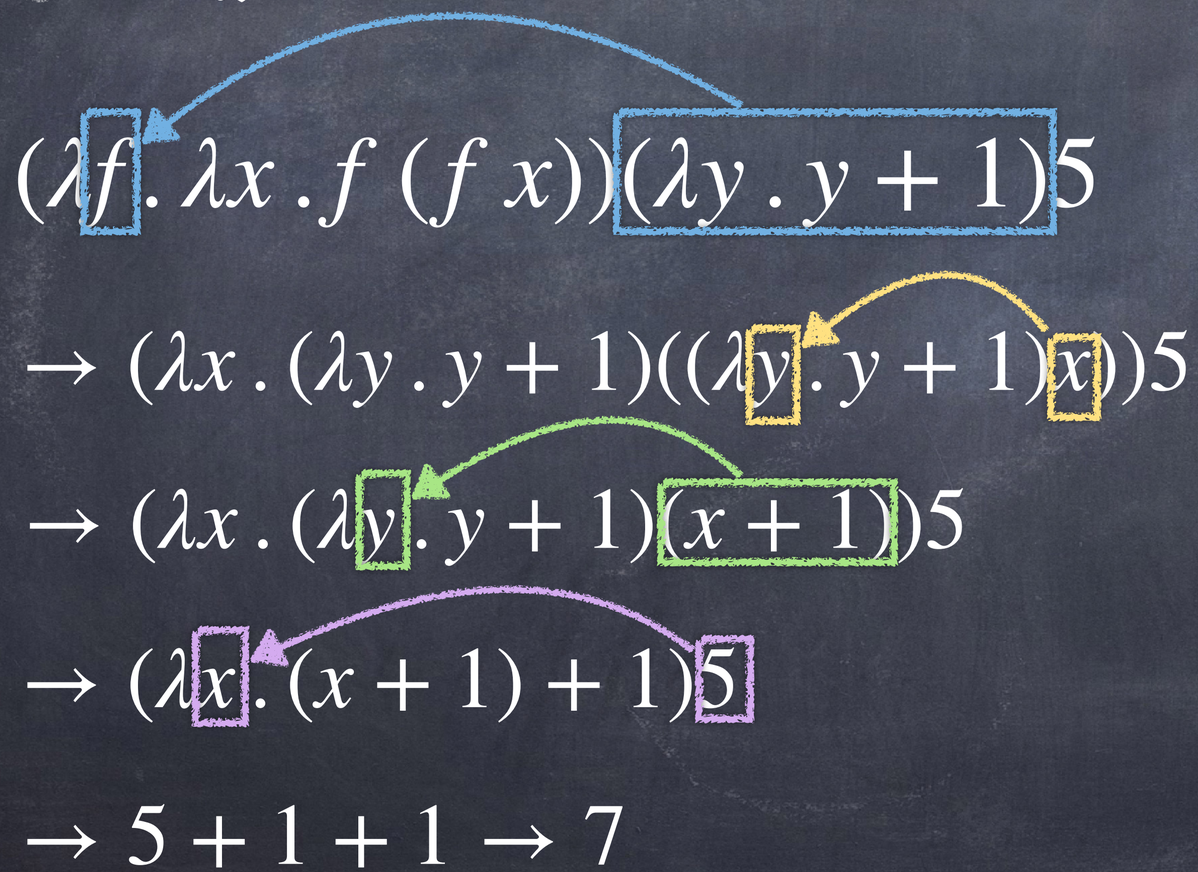
β-规约

- 基本规则：
  - $(\lambda x\,.\,M)N \rightarrow M[N/x]$
  - 即用N来替换参数x
  - 改规则可以一直持续应用于任何λ子项。

替换(Substitution)

β-规约

- 比如：


$(\lambda f.\ \lambda x\ .f\ (f\ x))(\lambda y\ .\ y+1)5$
$\rightarrow (\lambda x\ .\ (\lambda y\ .\ y+1)((\lambda y.\ y+1)\ x))5$
$\rightarrow (\lambda x\ .\ (\lambda y.\ y+1)\ (x+1))5$
$\rightarrow (\lambda x.\ (x+1)+1)5$
$\rightarrow 5+1+1 \rightarrow 7$

β-规约

- 比如：

$$(\lambda f\,.\,\lambda x\,.f\ (f\ x))(\lambda y\,.\,y+1)5$$

$$\rightarrow (\lambda x\,.\,(\lambda y\,.\,y+1)((\lambda y\,.\,y+1)x))5$$

$$\rightarrow (\lambda x\,.\,(\lambda y\,.\,y+1)(x+1))5$$

$$\rightarrow (\lambda x\,.\,(x+1)+1)5$$

$$\rightarrow 5+1+1\rightarrow 7$$

替换

- $M[N/x]$：将M中的x替换为N，下面我们通过递归给出替换的严格定义：

$$x[N/x] = N$$

$$y[N/x] = y$$

$$(M\ P)[N/x] = (M[N/x])(P[N/x])$$

$$(\lambda x\,.\,M)[N/x] = \lambda x\,.\,M$$

只替换自由变量，绑定变量的名字不重要！

$$(\lambda y\,.\,M)[N/x] = ?$$

避免命名捕捉

- 考虑如下式子：$(\lambda x\,.\,x - y)[x/y]$

$\lambda x\,.\,x - x$ ✗

- x的名字被λ的绑定变量捕捉了

解决方案：重命名

$(\lambda x\,.\,x - y)[x/y] = (\lambda z\,.\,z - y)[x/y] = \lambda z\,.\,z - x$

替换

- $M[N/x]$：将M中的x替换为N，下面我们通过递归给出替换的严格定义：

$$x[N/x] = N$$

$$y[N/x] = y$$

$$(M\ P)[N/x] = (M[N/x])(P[N/x])$$

$$(\lambda x\,.\,M)[N/x] = \lambda x\,.\,M$$

$$(\lambda y\,.\,M)[N/x] = \lambda y\,.\,(M[N/x]), \text{如果} y \notin fv(N)$$

$$(\lambda y\,.\,M)[N/x] = \lambda z\,.\,(M[z/y][N/x]), \text{如果} y \in fv(N) \text{且} z \text{是fresh的}$$

z没被使用过

β-规约的规则

$$\frac{}{(\lambda x\,.\,M)N \to M[N/x]}\ (\beta)$$

$$\frac{M \to M'}{MN \to M'N}$$

$$\frac{N \to N'}{MN \to MN'}$$

$$\frac{M \to M'}{\lambda x\,.\,M \to \lambda x\,.\,M'}$$

不断应用(β)规约到子项上

练习

$$\frac{}{(\lambda x\,.\,M)N \to M[N/x]}\ (\beta)$$

$$\frac{M \to M'}{MN \to M'N}$$

$$\frac{N \to N'}{MN \to MN'}$$

$$\frac{M \to M'}{\lambda x\,.\,M \to \lambda x\,.\,M'}$$

- $(\lambda f\,.\,f\ x)(\lambda y\,.\,y)$

应用β

$\to (f\ x)[(\lambda y\,.\,y)/f]$

应用β

$= (\lambda y\,.\,y)x$

$\to y[x/y]$

$= x$

练习

$$\frac{}{(\lambda x\,.\,M)N \to M[N/x]}\ (\beta)$$

$$\frac{M \to M'}{MN \to M'N}$$

$$\frac{N \to N'}{MN \to MN'}$$

$$\frac{M \to M'}{\lambda x\,.\,M \to \lambda x\,.\,M'}$$

- $(\lambda y\,.\,\lambda x\,.\,x - y)x$

应用β

$\to (\lambda x\,.\,x - y)[x/y]$

$= \lambda z\,.\,((x - y)[z/x][x/y])$

$= \lambda z\,.\,((z - y)[x/y])$

$= \lambda z\,.\,z - x$

练习

$$\frac{}{(\lambda x\,.\,M)N \to M[N/x]}\ (\beta)$$

$$\frac{M \to M'}{MN \to M'N}$$

$$\frac{N \to N'}{MN \to MN'}$$

$$\frac{M \to M'}{\lambda x\,.\,M \to \lambda x\,.\,M'}$$

- $\lambda x\,.\,(\lambda y\,.\,y+1)x$

应用第4规则

$$(\lambda y\,.\,y+1)x$$
$$\to (y+1)[x/y]$$
$$= x+1$$

应用β

$$\to \lambda x\,.\,x+1$$

练习

$$\frac{}{(\lambda x\,.\,M)N \to M[N/x]}\ (\beta)$$

$$\frac{M \to M'}{MN \to M'N}$$

$$\frac{N \to N'}{MN \to MN'}$$

$$\frac{M \to M'}{\lambda x\,.\,M \to \lambda x\,.\,M'}$$

- $(\lambda f\,.\,\lambda z\,.\,f(f\ z))(\lambda y\,.\,y + x)$

应用β

$\to \lambda z\,.\,(\lambda y\,.\,y + x)((\lambda y\,.\,y + x)z)$

应用β和第3、4规则

$\to \lambda z\,.\,(\lambda y\,.\,y + x)(z + x)$

应用β和第4规则

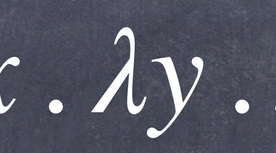
$\to \lambda z\,.\,z + x + x$

范式（Normal form）

reducible expression

- β-可约项（β-redex）：以$(\lambda x . M)N$形式出现的λ项
- β-范式（β-normal form）：没有β-redex的λ项
  - 其实就是λ表达式的终点，无法再运用任何β-规约的规则了。

范式（Normal form）

- 例子：
  - $\lambda x\,.\,\lambda y\,.\,x$ 
  - $\lambda x\,.\,(\lambda y\,.\,x)(\lambda z\,.\,z)$ 

汇聚（Church-Rosser theorem）

- 定理：
- λ项可以以任何顺序进行规约（应用规则的顺序），只要规约能得到β-范式结果，结果都是一致的。

汇聚（Church-Rosser theorem）

- 比如：

$$(\lambda f\,.\,\lambda x\,.\,f(f\ x))(\lambda y\,.\,y+1)2$$
$$\to (\lambda x\,.\,(\lambda y\,.\,y+1)((\lambda y\,.\,y+1)x))2$$
$$\to (\lambda x\,.\,(\lambda y\,.\,y+1)(x+1))2$$
$$\to (\lambda x\,.\,x+1+1)2$$
$$\to 2+1+1$$

$$(\lambda f\,.\,\lambda x\,.\,f(f\ x))(\lambda y\,.\,y+1)2$$
$$\to (\lambda x\,.\,(\lambda y\,.\,y+1)((\lambda y\,.\,y+1)x))2$$
$$\to (\lambda x\,.\,(\lambda y\,.\,y+1)(x+1))2$$
$$\to (\lambda y\,.\,y+1)(2+1)$$
$$\to 2+1+1$$

Church-Rosser theorem的形式化表达

- $M \to^* M'$，即0或者多步的$\to$，定义如下：

$M \to^0 M'$ 当且仅当 $M = M'$

$M \to^{k+1} M'$ 当且仅当 $\exists M''$，使得 $M \to M'' \wedge M'' \to^k M'$

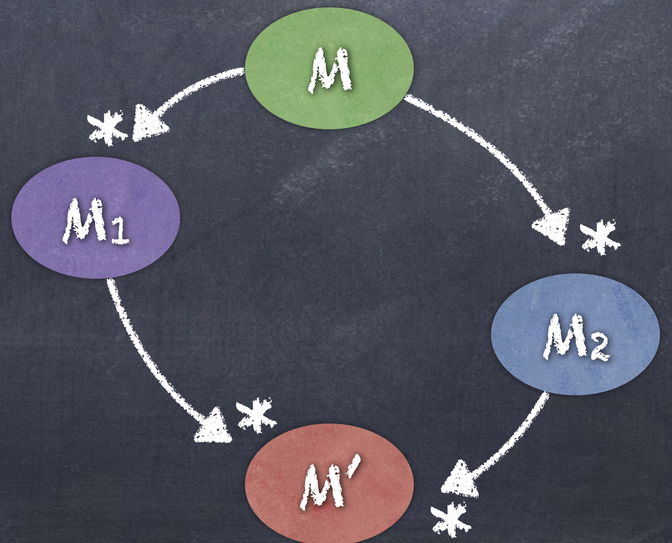
$M \to^* M'$ 当且仅当 $\exists k$，使得 $M \to^k M'$

} 递归定义

Church-Rosser theorem的形式化表达

Church-Rosser theorem:

- 如果$M \to^* M_1$#且$M \to^* M_2$，那么存在一个M'，使得$M_1 \to^* M'$#且$M_2 \to^* M'$



Church-Rosser theorem的证明

- ~~我找到了一个证明，但是这里太小了，放不下~~

https://student.cs.uwaterloo.ca/~cs442/W22/extras/c-r-thm-proof.pdf

推论

- 在α-等价的基础上，任何λ表达式最多只有一个β-范式。
- 那么一个问题是：如果一个λ表达式有很多β-可约项，先选择哪个进行规约呢？

好消息：无论选择哪个进行规约，最终最多只有一个β-范式

坏消息：有一些挑选策略可能会导致最终得不到β-范式

有些项是无法终止规约的

- $(\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x)$

  $\to (\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x)$

  $\to \ldots$

- $(\lambda x\,.\,x\ x\ y)(\lambda x\,.\,x\ x\ y)$

  $\to (\lambda x\,.\,x\ x\ y)(\lambda x\,.\,x\ x\ y)y$

  $\to \ldots$

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

  $\to f\left((\lambda x\,.f\ (x\ x))(\lambda x\,.f\ (x\ x))\right)$

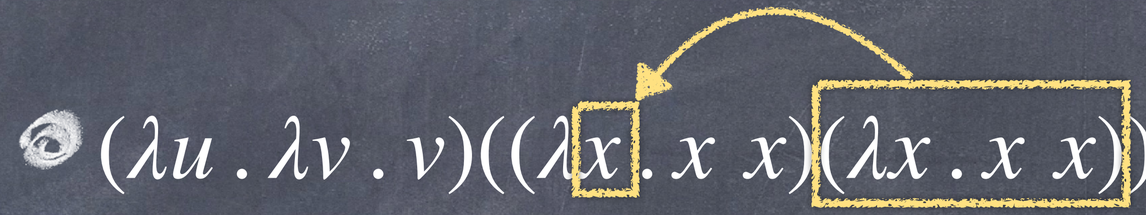
  $\to \ldots$

一些reduction策略可能会导致非终止

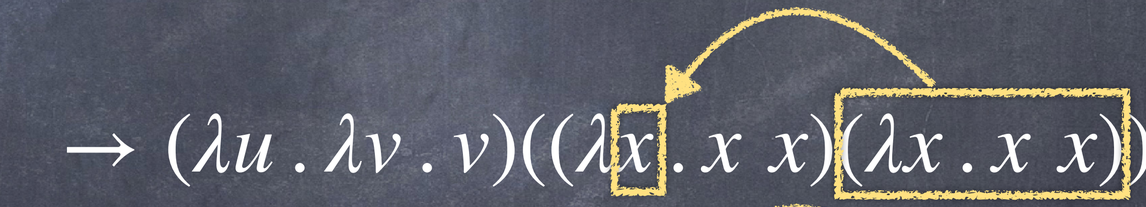
- $(\lambda u\,.\,\lambda v\,.\,v)((\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x))$

$\rightarrow \lambda v\,.\,v$

- $(\lambda u\,.\,\lambda v\,.\,v)((\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x))$



$\rightarrow (\lambda u\,.\,\lambda v\,.\,v)((\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x))$

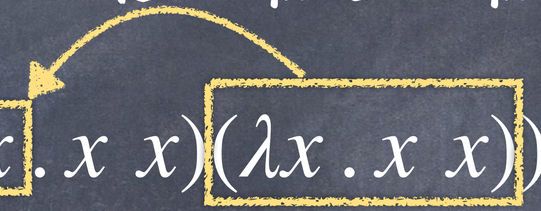


$\rightarrow (\lambda u\,.\,\lambda v\,.\,v)((\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x))$

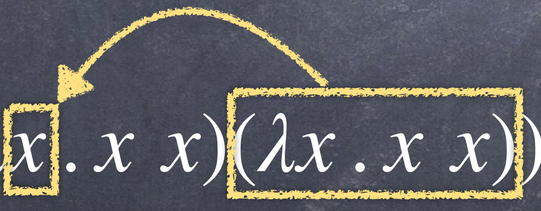
$\ldots$

规约策略

- 应用序(Applicative Order)：总是先规约最左、最内的可规约项。
  - $(\lambda u\,.\,\lambda v\,.\,v)((\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x))$



  - $\rightarrow (\lambda u\,.\,\lambda v\,.\,v)((\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x))$



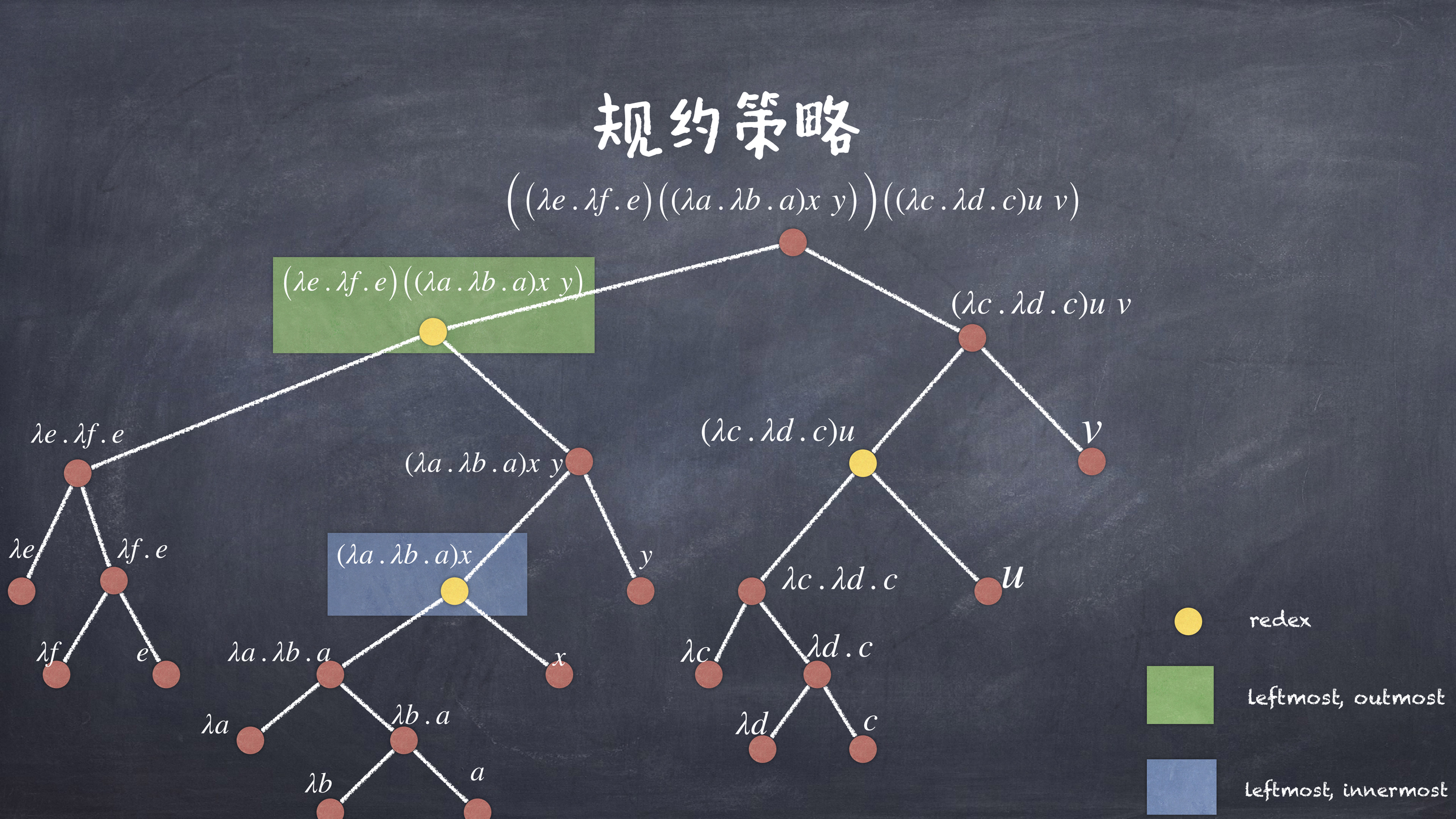
  - $\ldots$

May fail to find normal form even if it exists.

规约策略

- 一个最外（outmost）的可约项是一个不被其他可约项包含的可约项。
- 一个最内（innermost）的可约项是一个不包含其他可约项的可约项。

规约策略


$\big((\lambda e\,.\,\lambda f\,.\,e)((\lambda a\,.\,\lambda b\,.\,a)x\ y)\big)((\lambda c\,.\,\lambda d\,.\,c)u\ v)$
$(\lambda e\,.\,\lambda f\,.\,e)((\lambda a\,.\,\lambda b\,.\,a)x\ y)$
$(\lambda c\,.\,\lambda d\,.\,c)u\ v$
$\lambda e\,.\,\lambda f\,.\,e$
$(\lambda a\,.\,\lambda b\,.\,a)x\ y$
$(\lambda c\,.\,\lambda d\,.\,c)u$
v
λe
$\lambda f\,.\,e$
$(\lambda a\,.\,\lambda b\,.\,a)x$
y
$\lambda c\,.\,\lambda d\,.\,c$
u
λf
e
$\lambda a\,.\,\lambda b\,.\,a$
x
λc
$\lambda d\,.\,c$
λa
$\lambda b\,.\,a$
λd
c
λb
a
redex
leftmost, outmost
leftmost, innermost

规约策略

- 应用序(Applicative Order)：总是先规约最左、最内的可规约项。

leftmost, outmost

leftmost, innermost

$$(\lambda e\,.\,\lambda f\,.\,e)((\lambda a\,.\,\lambda b\,.\,a)x\ y)((\lambda c\,.\,\lambda d\,.\,c)u\ v)$$

$$\rightarrow (\lambda e\,.\,\lambda f\,.\,e)((\lambda b\,.\,x)\ y)((\lambda c\,.\,\lambda d\,.\,c)u\ v)$$

$$\rightarrow (\lambda e\,.\,\lambda f\,.\,e)\ x\ ((\lambda c\,.\,\lambda d\,.\,c)u\ v)$$

$$\rightarrow (\lambda f\,.\,x)((\lambda c\,.\,\lambda d\,.\,c)u\ v)$$

$$\rightarrow (\lambda f\,.\,x)\ u\ \rightarrow x$$

规约策略

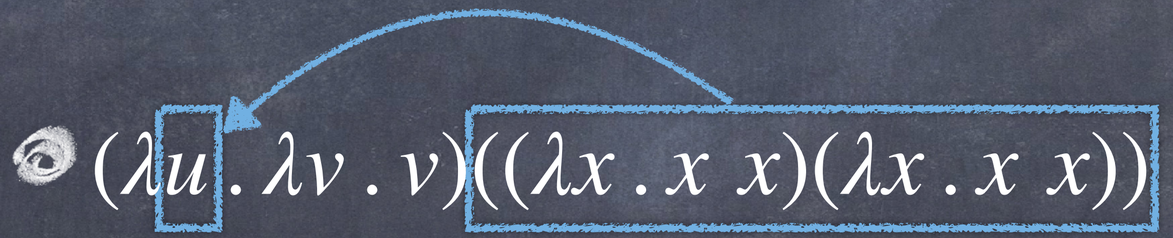
- 应用序(Applicative Order)：总是先规约最左、最内的可规约项。

- 直观地类比，有点像一个函数的参数永远先于函数自身进行规约。

"Call by value" in C

规约策略

- 正则序(Normal Order)：总是先规约最左、最外的可规约项。



$$(\lambda u\,.\,\lambda v\,.\,v)((\lambda x\,.\,x\ x)(\lambda x\,.\,x\ x))$$

$$\rightarrow \lambda v\,.\,v$$

定理[Standardization]：如果λ表达式存在β-范式，那么按正则序规约总能规约到该β-范式。

规约策略

- 正则序(Normal Order)：总是先规约最左、最外的可规约项。

leftmost, outmost

leftmost, innermost

$$(\lambda e\,.\,\lambda f\,.\,e)((\lambda a\,.\,\lambda b\,.\,a)x\ y)((\lambda c\,.\,\lambda d\,.\,c)u\ v)$$

$$\rightarrow (\lambda f\,.\,(\lambda a\,.\,\lambda b\,.\,a)x\ y)((\lambda c\,.\,\lambda d\,.\,c)u\ v)$$

$$\rightarrow (\lambda a\,.\,\lambda b\,.\,a)x\ y$$

$$\rightarrow (\lambda b\,.\,x)\ y\ \rightarrow x$$

规约策略

- 正则序(Normal Order)：总是先规约最左、最外的可规约项。

- 直观地类比，有点像一个函数的参数没有被规约（求值），而是优先被替换进函数体里。

"Call by name", e.g., ALGOL 60

在参数使用不到的情况下，这种策略会比applicative order要高效，然而如果参数会用到，而且会重复的话，applicative order 要比 normal order 要更加高效

各种规约策略类比

square(x) = x * x, 现在我们想计算square(1+2)

带记忆的 by name 的版本

Call by value

1. square (1+2)
2. square (3)
3. 3*3
4. 9

Call by name

1. square (1+2)
2. (1+2)*(1+2)
3. 3*(1+2)
4. 3*3
5. 9

Call by need

1. square (1+2)
2. Let z = 1+2 in z*z
3. Let z = 3 in z*z
4. 3*3
5. 9

基于Lambda演算的编程

- 我们已经介绍了Lambda演算的语法和语义。然而我们还不能解决实际的计算问题。
- 我们还缺少能够表达自然数、布尔数值、分支、递归这些操作的方法。
- 下面让我们来基于Lambda演算自身来定义和实现他们。

Church 编码（Church encoding）

- 对布尔值的编码
  - True $\triangleq \lambda x\,.\,\lambda y\,.\,x$
  - False $\triangleq \lambda x\,.\,\lambda y\,.\,y$

Church 编码（Church encoding）

- True $\triangleq \lambda x\,.\,\lambda y\,.\,x$
- False $\triangleq \lambda x\,.\,\lambda y\,.\,y$

相应的操作

- if b then M else N $\triangleq b\ M\ N$

这是理解下面定义的核心

True M N = $(\lambda x\,.\,\lambda y\,.\,x)M\ N$
$\rightarrow (\lambda y\,.\,M)N$
$\rightarrow M$

False M N = $(\lambda x\,.\,\lambda y\,.\,y)M\ N$
$\rightarrow (\lambda y\,.\,y)N$
$\rightarrow N$

Church 编码（Church encoding）

- True $\triangleq \lambda x . \lambda y . x$
- False $\triangleq \lambda x . \lambda y . y$

相应的操作

- Not $\triangleq \lambda b . b$ False True
- Not' $\triangleq \lambda b . \lambda x . \lambda y . b\ y\ x$

Not True = $(\lambda b . b$ False True $)$(True)
→ True False True
→ False
Not False = $(\lambda b . b$ False True)(False)
→ False False True
→ True

Not' True = $(\lambda b . \lambda x . \lambda y .$ b y x)(True)
→ $\lambda x . \lambda y .$ True y x
→ $\lambda x . \lambda y . y$ = False
Not' False = $(\lambda b . \lambda x . \lambda y .$ b y x)(False)
→ $\lambda x . \lambda y .$ False y x
→ $\lambda x . \lambda y . x$ = True

Church 编码（Church encoding）

- 相应的操作
  - And $\triangleq \lambda b\,.\,\lambda b'\,.\,b\ b'$ False
  - Or $\triangleq \lambda b\,.\,\lambda b'\,.\,b$ True b'

And True b
→* True b False
→ b
And False b
→* False b False
→ False

Or True b
→* True True b
→ True
Or False b
→* False True b
→ b

- True $\triangleq \lambda x\,.\,\lambda y\,.\,x$
- False $\triangleq \lambda x\,.\,\lambda y\,.\,y$

Church 编码（Church encoding）

- 对自然数的编码
  - 我们应该。。。。。
  - 等等，什么是自然数？

Church 编码（Church encoding）

- 皮亚诺公理

1. 0是自然数
2. 每个确定的自然数a，都有一个确定的后继数a'，a'也是自然数
3. 对于每个自然数b, c，$b = c$，当且仅当，b的后继数 $= c$的后继数
4. 0不是任何自然数的后继数
5. 任意关于自然数的命题，如果证明，它对0是真的，且假定它对自然数a为真时，可以证明对其后继a'也真。那么，命题对所有自然数都真。

Giuseppe Peano

Church 编码（Church encoding）

- 对自然数的编码
  - 对0的定义
  - 对后继函数S的定义
  - 这样我们就有了 $1 = S(0),\quad 2 = S(S(0)),\ \ldots,\quad n = S^n(0)$

Church 编码（Church encoding）

- 对自然数的编码
  - 对于上述思想的一种拓展，我们可以想象一下一个lambda演算下的自然数可以定义为如下形式：

$$\lambda f.\lambda x.\text{一些表达式}$$

后继函数

代表0

Church 编码（Church encoding）

- Church数(Church numeral)
  - $0 \triangleq \lambda f . \lambda x . x$
  - $1 \triangleq \lambda f . \lambda x . f\ x$
  - $2 \triangleq \lambda f . \lambda x . f\ (f\ x)$
  - ...
  - $n \triangleq \lambda f . \lambda x . f^n\ x$

不难看出church数用application次数表达自然数

Church 编码（Church encoding）

- $0 \triangleq \lambda f.\lambda x.x$
- $1 \triangleq \lambda f.\lambda x.f\ x$
- $2 \triangleq \lambda f.\lambda x.f\ (f\ x)$
- ...
- $n \triangleq \lambda f.\lambda x.f^n\ x$

当我们把函数和变量作用于church数上时

- $3\ f'\ x' = (\lambda f.\lambda x.f(f(f(x))))f'\ x'$

$$\rightarrow f'(f'(f'(x')))$$

apply 3 times based on x'

- $n\ f'\ x' = (\lambda f.\lambda x.f^n(x))f'\ x'$

$$\rightarrow f'^n(x')$$

apply n times based on x'

Church 编码（Church encoding）

- 操作 自增
  - Inc $\triangleq \lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\ (n\ f\ x)$
  - Inc' $\triangleq \lambda n\,.\,\lambda f\,.\,\lambda x\,.\,n\ f\ (f\ x)$

apply n times based on (f x)

Inc n = $(\lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\ (n\ f\ x))n$
$\rightarrow \lambda f\,.\,\lambda x\,.\,f\ (n\ f\ x)$
$= \lambda f\,.\,\lambda x\,.\,f\ ((\lambda f\,.\,\lambda x\,.\,f^n\ x)\ f\ x)$
$\rightarrow \lambda f\,.\,\lambda x\,.\,f(f^n\ x)$
$= \lambda f\,.\,\lambda x\,.\,f^{n+1}\ x$
$= n+1$

- 0 $\triangleq \lambda f\,.\,\lambda x\,.\,x$
- 1 $\triangleq \lambda f\,.\,\lambda x\,.\,f\ x$
- 2 $\triangleq \lambda f\,.\,\lambda x\,.\,f\ (f\ x)$
- ...
- n $\triangleq \lambda f\,.\,\lambda x\,.\,f^n\ x$

Church 编码（Church encoding）

- 操作 加法

apply n times based on (m f x)

apply m times based on x

- Add $\triangleq \lambda n\,.\,\lambda m\,.\,\lambda f\,.\,\lambda x\,.\,n\ f\ (m\ f\ x)$

Add n m = $(\lambda n\,.\,\lambda m\,.\,\lambda f\,.\,\lambda x\,.\,n\ f\ (m\ f\ x))\ n\ m$
$\rightarrow \lambda f\,.\,\lambda x\,.\,n\ f\ (m\ f\ x)$
$= \lambda f\,.\,\lambda x\,.\,n\ f\ (f^m(x))$
$= \lambda f\,.\,\lambda x\,.\,f^n(f^m(x))$
$= \lambda f\,.\,\lambda x\,.\,f^{n+m}\ x$
$= n + m$

- $0 \triangleq \lambda f\,.\,\lambda x\,.\,x$
- $1 \triangleq \lambda f\,.\,\lambda x\,.\,f\ x$
- $2 \triangleq \lambda f\,.\,\lambda x\,.\,f\ (f\ x)$
- ...
- $n \triangleq \lambda f\,.\,\lambda x\,.\,f^n\ x$

Church 编码（Church encoding）

- $0 \triangleq \lambda f.\lambda x.x$
- $1 \triangleq \lambda f.\lambda x.f\ x$
- $2 \triangleq \lambda f.\lambda x.f\ (f\ x)$
- ...
- $n \triangleq \lambda f.\lambda x.f^n\ x$

- 操作 乘法
  - Mult ≜ ?

回顾：$n\ f'\ x' = (\lambda f.\lambda x.f^n(x))f'\ x'$

$\rightarrow f'^n(x')$

如果f' = m f -> f' x = m f x

Church 编码（Church encoding）

- $0 \triangleq \lambda f\,.\,\lambda x\,.\,x$
- $1 \triangleq \lambda f\,.\,\lambda x\,.\,f\ x$
- $2 \triangleq \lambda f\,.\,\lambda x\,.\,f\ (f\ x)$
- ...
- $n \triangleq \lambda f\,.\,\lambda x\,.\,f^n\ x$

- 操作 乘法
  - Mult $\triangleq \lambda n\,.\,\lambda m\,.\,\lambda f\,.\,\lambda x\,.\,n\ (m\ f)\ x$

Mult n m $= (\lambda n\,.\,\lambda m\,.\,\lambda f\,.\,\lambda x\,.\,n\ (m\ f)\ x)\ n\ m$
$\rightarrow \lambda f\,.\,\lambda x\,.\,n\ (m\ f)\ x$
$\rightarrow \lambda f\,.\,\lambda x\,.\,(m\ f)^n\ x$
$\rightarrow \lambda f\,.\,\lambda x\,.\,(\lambda x'\,.\,f^m\ x')^n\ x$
$\rightarrow \lambda f\,.\,\lambda x\,.\,(f^m)^n\ x$
$= \lambda f\,.\,\lambda x\,.\,f^{m*n}\ x = \ = m*n$

Church 编码（Church encoding）

- 习题
  - 怎么定义幂运算？
    - Pow n m = n^m

习题

- 前驱
  - Pred $= \lambda n\,.\,\lambda f\,.\,\lambda x\,.\,n\ (\lambda g\,.\,\lambda h\,.\,h\ (g\ f))(\lambda u\,.\,x)(\lambda u\,.\,u)$
- 减法
  - Sub $= \lambda m\,.\,\lambda n\,.\,n$ Pred m

计算 Pred n 和 Sub n m，并思考可能出现的问题

Church 编码（Church encoding）

- Church数(Church numeral)
  - $0 \triangleq \lambda f\,.\,\lambda x\,.\,x$
  - $1 \triangleq \lambda f\,.\,\lambda x\,.\,f\ x$
  - $2 \triangleq \lambda f\,.\,\lambda x\,.\,f\ (f\ x)$
  - ...
  - $n \triangleq \lambda f\,.\,\lambda x\,.\,f^n\ x$

- 操作 是否0
  - $\text{Iszero} \triangleq \lambda n\,.\,n\ (\lambda x\,.\,\text{False})\ \text{True}$

Iszero 0=$(\lambda n\,.\,n\ (\lambda x\,.\,\text{False})\ \text{True})\ 0$
→ $0\ (\lambda x\,.\,\text{False})\ \text{True}$
→ True

Iszero 1=$(\lambda n\,.\,n\ (\lambda x\,.\,\text{False})\ \text{True})\ 1$
→ $1\ (\lambda x\,.\,\text{False})\ \text{True}$
→ $(\lambda x\,.\,\text{False})\ \text{True}$
→ False

Iszero n=$(\lambda n\,.\,n\ (\lambda x\,.\,\text{False})\ \text{True})\ n$
→ $n\ (\lambda x\,.\,\text{False})\ \text{True}$
→ $(\lambda x\,.\,\text{False})^n\ \text{True}$
→ False

习题

- 习题
  - 怎么定义小于等于？
    - Leq n m = if $n \leq m$ then True else False

Church 编码（Church encoding）

- 对有序对（Pairs）的编码
  - Pair $\triangleq \lambda x\,.\,\lambda y\,.\,\lambda f\,.\,f\ x\ y$
  - $\pi_0 \triangleq \lambda p\,.\,p$ True·
  - $\pi_1 \triangleq \lambda p\,.\,p$ False

$\pi_0(\text{Pair}\ M\ N) \rightarrow^* M$

$\pi_1(\text{Pair}\ M\ N) \rightarrow^* N$

- True $\triangleq \lambda x\,.\,\lambda y\,.\,x$
- False $\triangleq \lambda x\,.\,\lambda y\,.\,y$

Church 编码（Church encoding）

- 对元组（Tuples）的编码
  - Tuple $\triangleq \lambda x_1 . \lambda x_2 . \ldots . \lambda x_n . \lambda f . f\ M_1\ M_2 \ldots M_n$
  - $\pi_i \triangleq \lambda p . p\ (\lambda x_1 . \lambda x_2 . \ldots . \lambda . x_n . x_i)$

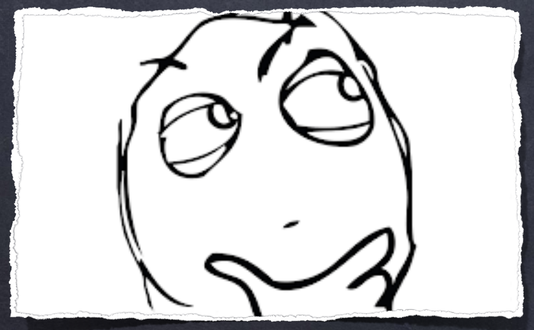
基于Lambda演算的编程

- 我们已经完全具备了通用编程的能力了吗？

不，我们还没有递归！

递归（Recursion）

- 考虑阶乘（Factorial）的求解：
  - Fact(n) = if (n == 0) then 1 else n* Fact(n-1)



Fact是什么？

我们为什么要知道Fact是什么？

递归

- 我们先来看第二个问题，即我们为什么需要知道Fact
  - 这是因为Lambda演算不允许我们对函数命名，然后调用它！
  - Lambda演算是匿名函数（anonymously）

递归

⚠️这里需要注意的是，之前定义的函数

- 如Inc、Add、Mult、Not、And、Iszero等，这些名字只是它们所代表的Lambda 表达式的简便记法，它与我们学习的函数名不一样。
- 即这些名字本质上是宏（macros），它们在使用的时候会扩展成相应的Lambda 表达式。
- 因此，对于诸如Fact(n) = if (n == 0) then 1 else n* Fact(n-1)形式的表达式，在不知道Fact是什么的话，我们需要将其拓展无限次！

递归

那么问题来了，为什么Lambda演算这么设计呢？

历史原因：Alonzo Church想避免自指（自己调用自己），从而陷入Curry's paradox。因此Church希望这样的设计可以让Lambda演算成为数学的基础。 -然而这个尝试失败了！

这样设计的好处：更加简单。因为要能够实现依据名字进行函数调用，我们必须构造"环境"，使得名字和具体的表达联系起来。

递归

- 现在让我们回到第一个问题，Fact 是什么？
  - Fact(n) = if (n == 0) then 1 else n* Fact(n-1)

我们要求解一个方程！

递归

- 让我们先考虑一个简单的例子 $X = 4/X$：
  - 如果x'是上述方程的解话，
  - 那么$x' = (\lambda y.4/y)(x')$

递归

- 因此，可以将阶乘问题转化为如下形式：
  - Fact(n) = if (n == 0) then 1 else n* Fact(n-1)
  - Fact = λn . if (n == 0) then 1 else n* Fact(n-1)
  - Fact = (λf . λn . if (n == 0) then 1 else n* f(n-1)) Fact
  - Fact = (λf . λn . (Iszero n) 1 (n* f(n-1))) Fact

不动点（Fixpoint）

- x是函数f的一个不动点，当且仅当$f(x) = x$
- 不是所有函数都有不动点
  - $f(x) = x * x$, 该函数有两个不动点0和1
  - $f(x) = x + 1$, 该函数么没有不动点
  - $f(x) = x$, 该函数有无穷多个不动点

不动点

- 在Lambda演算中，任何Lambda项都有不动点
- 不动点组合子（Fixpoint combinator）是一个高阶的函数 h，满足：
  - 对于任何f，$(h\,f)$是f的一个不动点，即$h\,f = f\,(h\,f)$

不动点

- 图灵（Turing）不动点组合子Θ：
  - 让$A = \lambda x\,.\,\lambda y\,.\,y(x\ x\ y)$，则$\Theta = A\ A$
- 丘奇（Church）不动点组合子$\mathbf{Y}$：
  - $\mathbf{Y} = \lambda f\,.\,(\lambda x\,.f\ (x\ x))(\lambda x\,.f\ (x\ x))$
- 事实上有无穷多的可能的组合子

图灵不动点

- 让$A = \lambda x\,.\,\lambda y\,.\,y(x\ x\ y)$，则$\Theta = A\ A$
- 让我们证对任意f，有$\Theta\ f = f\ (\Theta\ f)$

$$\Theta\ f = A\ A\ f = (\lambda x\,.\,\lambda y\,.\,y\ (x\ x\ y))\ A\ f$$
$$\rightarrow (\lambda y\,.\,y\ (A\ A\ y))\ f$$
$$\rightarrow f\ (A\ A\ f)$$
$$= f\ (\Theta\ f)$$

丘奇不动点

- $\mathbf{Y} = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$
- 让我们证对任意f，有$\mathbf{Y}\ f = f\ (\mathbf{Y}\ f)$

$$\begin{aligned}
\mathbf{Y}\ f &= \left(\lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))\right)\ f \\
&\to (\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x)) \\
&\to f\ \left((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))\right) \\
&= f\ (\mathbf{Y}\ f)
\end{aligned}$$

不动点

- 为什么$f(x) = x + 1$, 该函数没有不动点，而对应的Lambda表达式$\lambda x\,.\,x + 1$就有不动点？

$$\Theta\ (\lambda x\,.\,x + 1) = A\ A\ (\lambda x\,.\,x + 1)$$

$$\rightarrow^*\ (\lambda x\,.\,x + 1)\ (\Theta\ (\lambda x\,.\,x + 1))$$

$$\rightarrow\ \Theta\ (\lambda x\,.\,x + 1) + 1$$

不动点

- 如果采用之前的自增来算，即$\lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\ (n\ f\ x)$

$$\Theta\ (\lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\ (n\ f\ x))$$

$$\rightarrow^*\ (\lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\ (n\ f\ x))\big(\Theta\ (\lambda n\,.\,\lambda f\,.\,\lambda x\,.\,f\ (n\ f\ x))\big)$$

回到阶乘问题

- 之前我们已经进行了转化：
  - Fact(n) = if (n == 0) then 1 else n* Fact(n-1)
  - Fact = $(\lambda f\,.\,\lambda n\,.$ (Iszero n) 1 (n* f(n-1))) Fact
- 因此，Fact是$(\lambda f\,.\,\lambda n\,.$ (Iszero n) 1 (n* f(n-1)))的不动点：
  - Fact = Θ $(\lambda f\,.\,\lambda n\,.$ (Iszero n) 1 (n* f(n-1)))

习题

- 给出斐波那契数列
  - $F(0) = 0$
  - $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$

求出F

基于Lambda演算的编程

- 布尔值
- 自然数
- 有序对
- 递归函数
- 列表、树…

想更加深入了解Lambda演算，可以参考
"Lecture Notes on the Lambda Calculus"
by Peter Selinger

此外，也可以查看http://worrydream.com/AlligatorEggs/
其用图形的形式介绍lambda calculus

总结

- λ表达式
  - 自由和绑定变量
- β-规约
  - 替换
- 编程
  - 自然数和布尔值的定义
  - 不动点

Any questions ?