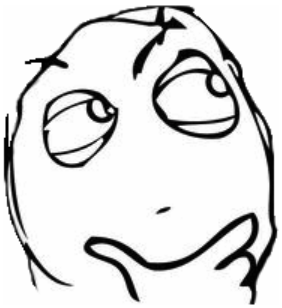
函数式编程
FUNCTIONAL PROGRAMMING

郭家寶
byvoid@byvoid.com

什么是函数式编程？

你肯定听说过「面向对象」

- 封装
- 继承
- 多态
- 总之好像很厉害的样子



函数式?

- 听起来简直弱爆了

真相是……

函数式编程是一种完全不同的编程范式，与之相对的是「指令式编程」。

它的特点是：

- 不可变量
- 惰性求值
- 高阶函数
- 无副作用
- 一切皆函数

从停机问题开始

调程序的时候经常会遇到死循环的Bug，聪明的你有没有想过「发明」一个自动检查程序里面有没有死循环的工具呢？

不管你有没有过这种想法，反正我有过，可惜答案是，没有！

停机问题：给定任意一个程序及其输入，判断该程序是否能够在有限次计算以内结束。

假设有停机判定算法

假设真的做出了这个算法，你只要给它任意一个函数和这个函数的输入，它就能告诉你这个函数会不会运行结束。

我们用下列伪代码描述：

```
function halting(func, input) {
    return if_func_will_halt_on_input;
}
```

充分利用停机判定

我们构造另一函数:

```
function ni_ma(func) {
	if (halting(func, func)) {
		for(;;) //死循环
	}
}
```

接下来，调用:

```
ni_ma(ni_ma)
```

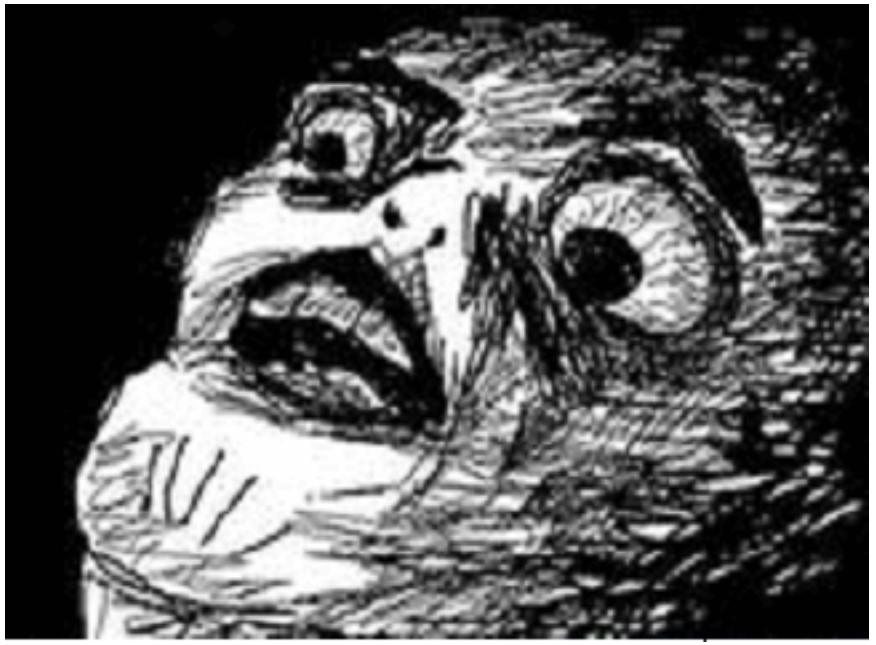
尼玛，悖论！

函数ni_ma以ni_ma为输入时

到底停机还是不停机？

所以……

停机问题不可判定



LAMBDA演算语法

停机问题只是个引子，接下来让我们步入正题

λ演算的语法只有三条：

- <表达式> ::= <标识符>
- <表达式> ::= λ <标识符+>. <表达式>
- <表达式> ::= (<表达式> <表达式>)

例如：

- λ x y. x + y

LAMBDA演算语法

前面定义的三条语法中，前两条用于产生「函数」，第三条用于函数「调用」，例如：

- `((λ x y. x + y) 2 3)`

简便起见：

- `let add = λ x y. x + y`

则

- `(add 2 3)`

LAMBDA演算公理

置换公理

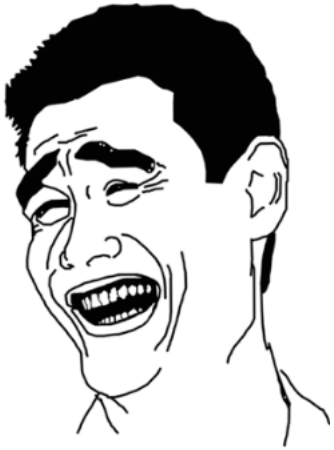
- λ x y. x + y => λa b. a + b

代入公理

- (λ x y. x + y) a b => a + b

以上就是λ演算公理系统。

简单吧?



函数生成器

λ演算相当于一个函数生成器

- `let mul = λ x y. x * y`
- `let con = λ x y. xy`

代入

- `mul 3 5 -> 3 * 5`
- `con ‘BYV’ ‘oid’ -> ‘BYVoid’`

基本表达式

考虑not函数，定义

```
let not =
```

- `false -> true`
- `true -> false`

基本表达式

如何定义and?

```
let and =
```

- `true true -> true`
- `true false -> false`
- `false true -> false`
- `false false -> false`

广义AND

```
let and =
```

- `true value -> value`
- `false value -> false`
- `value true -> value`
- `value false -> false`

定义IF

请大家考虑，如何定义if

```
let if =
λ cond tvalue fvalue .
(cond and tvalue) or (not cond and fvalue)
```

则 `if true a b`

- `-> (true and a) or (not true and b)`
- `-> a or false`
- `-> a`

递归？

下面让我们来点有意思的，计算n的阶乘。

- `let fact = λ n. if (n == 0) 1 (n * fact n-1)`

问题出现了，我们在定义fact的时候引用到了自身。

虽然在实际的编程中编译器都可以识别这种方式的定义，但这不符合严格的数学公理体系。

如何表示递归

重新审视

- `let fact = λ n. if (n == 0) 1 (n * fact n-1)`

不是无法引用自身吗？我们把「自身」参数化

```
let P = λ self n. if (n==0) 1 (n * self(self n-1))
```

然后再令

```
let fact n = P (P n)
```

如此一来

```
fact 4
-> P (P 4)
-> if (4==0) (1) (4 * P(P n-1))
-> 4 * P(P 3)
-> 4 * 3 * P(P 2)
-> 4 * 3 * 2 * P(P 1)
-> 4 * 3 * 2 * 1
```

成功了！

可惜……

这并不是真正的递归，只是每次传入了一个额外的参数，反复调用而已。

那么我们的目的是什么呢？我想要一个真正的递归函数，就是

- `let fact = λ n. if (n == 0) 1 (n * fact n-1)`

但λ演算没有这样一条公理可以使你在定义函数的时候引用本身，怎么办？

大胆的想法

不管三七二十一，我们认为真正的fact是存在的。

然后再让我们回到那个带参数的「伪递归」函数:

- `let P = λ self n. if (n==0) 1 (n * self n-1)`

P接收两个参数，但我们可以「部分求值」:

- `P(fact) -> λ n. if (n==0) 1 (n * fact n-1)`

神奇的现象出现了!

- `P(fact) = fact`

不动点

我们发现了P的一个「不动点」：

- `P(fact) = fact`

什么是不动点呢？就是一个点（广义）在一个函数的映射下，得到的结果仍然是这个点。

想象一下墙上的一张中国地图掉在了地上，地图上面肯定有且只有一个点与它实际的位置是重合的。

找到不动点

现在，我们只有找到这个不动点，就能把「伪递归」函数

- `let P = λ self n. if (n==0) 1 (n * self n-1)`

转换为真正的递归函数了。

神奇的Y

所以，让我们继续假设有个神奇的函数Y，它可以找到这个伪递归函数的不动点，即：

- `Y(F) = f = F(Y(F))`

其中`F(f) = f`，那么就有

- `Y(P) = fact`

只有我们有了Y，就可以把伪递归函数变幻成我们要的真递归函数了。

构造Y组合子

让我们一睹Y组合子的尊容吧

- `let Y = λ F. G(G)`
- 其中`G = λ self. F(self(self))`

验证一下

Y(P)

= G(G)，其中 G = λ self. P(self(self))

= P(G(G))

= λ n. if (n==0) 1 (n * G(G) n-1)

假设Y(P) = fact，那么

Y(P) = fact = λ n. if (n==0) 1 (n * fact n-1)

这就是我们梦寐以求的真正的递归函数！

终于有了Y组合子

现在当我们想定义递归函数的时候，只需增加一个self参数，按伪递归的方法定义，然后再用Y组合子一套用，就变成我们想要的真递归了。

题外话，硅谷有一个著名的Y组合子（Y-Combinator）公司，它是一个天使投资公司，专门为早期创业团队投资第一桶金。

其创始人Paul Graham写过一本著名的书《Hackers and Painters》，中文译本叫做《黑客与画家》。

图灵等价

我们已经成功地推导出了Y组合子，就相当于在λ演算公理体系中推导出了一条定理：

- 可以在定义函数的过程中引用自身

这条定理是证明λ演算图灵等价的一个重要步骤。

λ演算是图灵等价的意味着什么呢？

意味着它的计算能力与我们的计算机是一致的。也就是说任何程序都能用λ演算描述，同时λ演算描述的函数一定可以由计算机计算。

停机问题的等价命题

回想我们刚才讲到的停机问题，即不可判定一个图灵机在给定任意输入的时候是否可以停机。

这个命题在λ演算中的等价命题是：

「不存在一个算法能够判定任意两个λ函数是否等价，即对于所有的n，有f(n)=g(n)。」

请大家思考，证明出来有奖！

真实世界中的函数式编程

前面我讲的这一大堆都是λ演算的理论，完全是用数学家的思维在考虑问题。

下面我们来转换头脑，用工程师的头脑来看看真实世界中的函数式编程。

HASKELL

Haskell是一个纯函数式编程语言，它的名字是为了纪念Haskell Curry而命名的。

我们刚才讲的Y组合子就是他发现的，此外他还提出了函数的柯里化(Currying)，即部分求值。

Haskell中一切都是函数，甚至没有指令式编程中变量的概念，它的变量全部都是只允许一次赋值，然后不可改变，就像数学推导中对变量的赋值一样。

HASKELL

Haskell还没有一般意义上的控制流结构，如for循环，取而代之的是递归。

Haskell还有两个重要的特性，即无副作用和惰性求值。

无副作用指的是任何函数在给定同样输入的情况下每次调用的结果都一样，而惰性求值指的是函数除非需要，否则不会立即计算。

第一个HASKELL程序

在运行Haskell程序之前，首先你需要安装一个GHC编译器，然后运行ghci即可进入交互模式。

输入let max a b = if a>b then a else b

```
Prelude> max 3 4
4
Prelude> max 1.0001 1
1.0001
Prelude> max "BYVoid" "CmYkRgB123"
"CmYkRgB123"
```

列表

Haskell中的列表是这样定义的:

- `list X :: = [] | elem : (list X)`

即:

- 空列表 = `[]`
- `[1] = 1:[]`
- `[1, 2, 3] = 1:2:3:[]`

输入`2:1:3:7:8:[]`可以看到

`[2,1,3,7,8]`

模式匹配

定义:

- `let first (elem:rest) = elem`

输入 `first [1,3]` 可以看到结果1

`elem:rest`是Haskell的函数参数模式匹配。

对于列表`[1,3]`，实质上是`1:3:[]`，`elem`匹配了1，`rest`匹配了`3:[]`，也就是`[3]`。

列表求和

把以下内容保存为acc.hs

```
accumulate [] = 0

accumulate (elem:rest) = elem + accumulate
rest

main = print (accumulate [1,2,3])
```

运行runghc acc.hs，可以看到结果

6

判断回文

```
palindrome []  = True
palindrome [_] = True
palindrome (elem:rest) = (elem == last rest) && (palindrome(init rest))
>palindrome [1, 2, 3, 2, 1]
True
>palindrome [1, 1, 2]
False
>palindrome "madam"
True
```

删除连续重复元素

```
cut cond [] = []
cut cond (elem:rest) = if cond elem then
cut cond rest else elem:rest
compress [] = []
compress (elem:rest) = elem : compress
(cut (== elem) rest)
>compress [1, 2, 2, 2, 3, 3]
[1, 2, 3]
>compress "aaabbaccc"
"abac"
```

惰性求值

Haskell中可以定义无穷列表，例如：

- [1..]表示所有的正整数
- [1,3..]表示所有的奇数

这在大多数编程语言中都是不可思议的，因为大多数语言都是及早求值(eager evaluation)的，Haskell的惰性求值(lazy evaluation)特性可以让列表按需取用。

例如[1,3..] !! 42 可以返回结果85

FIBONACCI数列

如何用Haskell实现Fibonacci数列？最符合数学描述的方法是：

```
fib 0 = 1

fib 1 = 1

fib a = fib (a - 1) + fib (a - 2)
```

不巧的是，这个算法是$O(2^N)$的，Haskell编译器还没有聪明到可以实现递归记忆化。

线性算法

让我们利用无穷列表来实现线性算法！一行代码即可解决：

- `fib = 1:1:zipWith (+) fib (tail fib)`

```
fib !! 4是5，fib !! 42是433494437

fib !! 1000

70330367711422815821835254877183549770181269836358732742604905087154537118196933579742249494562611733487750449241765991088186363265450223647106012053374121273867339111198139373125598767690091902245245323403501
```

解释一下

```
fib = 1:1:zipWith (+) fib (tail fib)
```

tail返回列表除了第一项以外后面的内容，例如

- `tail[1..]`返回`[2..]`

zipWith功能是两个列表每个元素通过一个函数分别计算，并返回结果的列表，例如

- `zipWith (*) [2,3,5] [1,2,3]`返回`[2,6,15]`

于是

```
fib = 1:1:zipWith (+) fib (tail fib)
```

生成了一个无穷列表，前两个元素都是1，后面的元素由现有列表错位相加而成，即

```
  [1, 1, 2, 3, 5, 8, 13, 21…] //fib
+ [1, 2, 3, 5, 8, 13, 21, 34…] //tail fib
= [2, 3, 5, 8, 13, 21, 34, 55…]
```

由于惰性求值，列表不会被立即计算，只有当我们用到其中元素的时候才会算。

快速排序

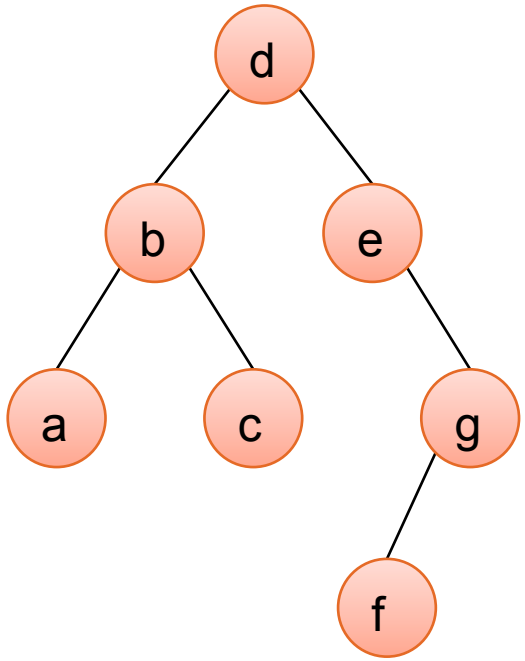
```
qsort (elem:rest) = (qsort lesser) ++
[elem] ++ (qsort greater)
 where
   lesser  = filter (< elem) rest
   greater = filter (>= elem) rest
```

++用于列表连接

filter返回列表中满足条件的元素组成的列表

二叉树表示

```
data Tree a = Empty | Node a (Tree a) (Tree a)
tree = Node 'd'
  (Node 'b'
    (Node 'a' Empty Empty)
    (Node 'c' Empty Empty)
  )
  (Node 'e'
    Empty
    (Node 'g'
      (Node 'f' Empty Empty)
      Empty
    )
  )
```



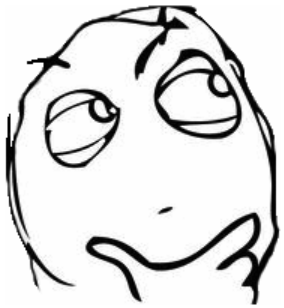
中序遍历

```
inorder Empty = []

inorder (Node value left right) =
inorder left ++ [value] ++ inorder right
```

```
>inorder tree

"abcdefg"
```



树的高度

```
height Empty = 0

height (Node value left right) =
max (height left) (height right) + 1

>height tree

4
```

高阶函数

高阶函数的参数是函数，通过部分求值可以返回函数。

```
traverse func zero Empty = zero

traverse func zero (Node value left right) =
func value
(traverse func zero left)
(traverse func zero right)

height_func _ a b = max a b + 1

>traverse height_func 0 tree

4
```

高阶函数

中序遍历函数:

```
inorder_func value left right = left ++ [value] ++ right
```

通过部分求值产生低阶函数:

```
inorder = traverse inorder_func []
```

```
>inorder tree
"abcdefg"
```

工程中的函数式编程

函数式编程已经不仅限于像Haskell这样非主流的语言了，其思想已经被几乎所有主流的编程语言所接受。

支持函数式编程特性的主流语言有:

- Python
- JavaScript
- Ruby
- C++ 11
- C#
- Scala

匿名函数

- Python
  - `lambda x : x**2`
- C#
  - `x => x**2`
- JavaScript
  - `function (x) {return x * x}`
- C++ 11
  - `[](int x) -> int {return x * x;}`

闭包

```
function make_closure() {
  var inner_varible = 0;
  return function () {
    return inner_varible++;
  }
}
var counter = make_closure();
counter(); // 0
counter(); // 1
```

用闭包实现柯里化

多数语言无法部分求值，原因是柯里化（部分求值）与参数表机制冲突。但可以用闭包实现。

例如对pow部分求值：

```
function pow5(x) {
  return Math.pow(x, 5);
}
pow5(2); //输出 32
```

谢谢大家

郭家寶 byvoid@byvoid.com

欢迎访问我的个人网站

HTTP://WWW.BYVOID.COM/BLOG

参考资料

- http://zh.wikipedia.org/wiki/不动点组合子
- http://mindhacks.cn/2006/10/15/cantor-godel-turing-an-eternal-golden-diagonal/
- http://www.byvoid.com/blog/godel-incompleteness-theorems-agnosticism/
- http://www.haskell.org/haskellwiki/99_questions
- https://developer.mozilla.org/en/JavaScript/Guide/Closures
- http://www.cprogramming.com/c++11/c++11-lambda-closures.html