康托尔、哥德尔、图灵——永恒的金色对角线

康托尔、哥德尔、图灵——永恒的金色对角线

转载自：
http://blog.csdn.net/pongba/archive/2006/10/15/1336028.aspx

我看到了它，却不敢相信它[1]。

——*康托尔*

计算机是数学家一次失败思考的产物。

——*无名氏*

哥德尔的不完备性定理震撼了20世纪数学界的天空，其数学意义颠覆了希尔伯特的形式化数学的宏伟计划，其哲学意义直到21世纪的今天仍然不断被延伸到各个自然学科，深刻影响着人们的思维。图灵为了解决希尔伯特著名的第十问题而提出有效计算模型，进而作出了可计算理论和现代计算机的奠基性工作，著名的停机问题给出了机械计算模型的能力极限，其深刻的意义和漂亮的证明使它成为可计算理论中的标志性定理之一。丘齐，跟图灵同时代的天才，则从另一个抽象角度提出了lambda算子的思想，与图灵机抽象的倾向于硬件性不同，丘齐的lambda算子理论是从数学的角度进行抽象，不关心运算的机械过程而只关心运算的抽象性质，只用最简洁的几条公理便建立起了与图灵机完全等价的计算模型，其体现出来的数学抽象美开出了函数式编程语言这朵奇葩，Lisp、Scheme、Haskell... 这些以抽象性和简洁美为特点的语言至今仍然活跃在计算机科学界，虽然由于其本质上源于lambda算子理论的抽象方式不符合人的思维习惯从而注定无法成为主流的编程语言[2]，然而这仍然无法妨碍它们成为编程理论乃至计算机学科的最佳教本。而诞生于函数式编程语言的神奇的Y combinator至今仍然让人们陷入深沉的震撼和反思当中...

然而，这一切的一切，看似不很相关却又有点相关，认真思考其关系却又有点一头雾水的背后，其实隐隐藏着一条线，这条线把它们从本质上串到了一起，而顺着时光的河流逆流而上，我们将会看到，这条线的尽头，不是别人，正是只手拨开被不严密性问题困扰的19世纪数学界阴沉天空的天才数学家康托尔，康托尔创造性地将一一对应和对角线方法运用到无穷集合理论的建立当中，这个被希尔伯特称为“谁也无法将我们从康托尔为我们创造的乐园中驱逐出去”、被罗素称为“19世纪最伟大的智者之一”的人，他在集合论方面的工作终于驱散了不严密性问题带来的阴霾，仿佛一道金色的阳光刺破乌云，19世纪的数学终于看到了真正严格化的曙光，数学终于得以站在了前所未有的坚固的基础之上；集合论至今仍是数学里最基础和最重要的理论之一。而康托尔当初在研究无穷集合时最具天才的方法之一——对角线方法——则带来了极其深远的影响，其纯粹而直指事物本质的思想如洪钟大吕般响彻数学和哲学的每一个角落[3]。随着本文的展开，你将会看到，刚才提到的一切，歌德尔的不完备性定理，图灵的停机问题，lambda算子理论中神奇的Y combinator、乃至著名的罗素悖论、理发师悖论等等，其实都源自这个简洁、纯粹而同时又是最优美的数学方法，反过来说，从康托尔的对角线方法出发，我们可以轻而易举地推导出哥德尔的不完备性定理，而由后者又可以轻易导出停机问题和Y combinator，实际上，我们将会看到，后两者也可以直接由康托尔的对角线方法导出。尤其是Y combinator，这个形式上绕来绕去，本质上捉摸不透，看上去神秘莫测的算子，其实只是一个非常自然而然的推论，如果从哥德尔的不完备性定理出发，它甚至比停机问题还要来得直接简单。总之，你将会看到这些看似深奥的理论是如何由一个至为简单而又至为深刻的数学方法得出的，你将会看到最纯粹的数学美。

图灵的停机问题(The Halting Problem)

了解停机问题的可以直接跳过这一节，到下一节“Y Combinator”，了解后者的再跳到下一节“哥德尔的不完备性定理”

我们还是从图灵著名的停机问题说起，一来它相对来说是我们要说的几个定理当中最简单的，二来它也最贴近程序员。实际上，我以前曾写过一篇

关于图灵机的文章，有兴趣的读者可以从那篇开始，那篇主要是从理论上阐述，所以这里我们打算避开抽象的理论，换一种符合程序员思维习惯的直观方式来加以解释。

停机问题
不存在这样一个程序（算法），它能够计算任何程序（算法）在给定输入上是否会结束（停机）。

那么，如何来证明这个停机问题呢？反证。假设我们某一天真做出了这么一个极度聪明的万能算法（就叫God_algo吧），你只要给它一段程序（二进制描述），再给它这段程序的输入，它就能告诉你这段程序在这个输入上会不会结束（停机），我们来编写一下我们的这个算法吧：

```
bool God_algo(char* program, char* input)
{
if(<program> halts on <input>)
return true;
return false;
}
```

这里我们假设if的判断语句里面是你天才思考的结晶，它能够像上帝一样洞察一切程序的宿命。现在，我们从这个God_algo出发导出一个新的算法：

```
bool Satan_algo(char* program)
{
if( God_algo(program, program) ){
  while(1); // loop forever!
  return false; // can never get here!
}
else
  return true;
```

}

正如它的名字所暗示的那样，这个算法便是一切邪恶的根源了。当我们把这个算法运用到它自身身上时，会发生什么呢?

Satan_algo(Satan_algo);

我们来分析一下这行简单的调用:

显然，Satan_algo(Satan_algo)这个调用要么能够运行结束返回（停机），要么不能返回（loop forever）。

如果它能够结束，那么Santa_algo算法里面的那个if判断就会成立（因为God_algo(Santa_algo,Santa_algo)将会返回true），从而程序便进入那个包含一个无穷循环while(1);的if分支，于是这个Satan_algo(Satan_algo)调用便永远不会返回（结束）了。

而如果Satan_algo(Satan_algo)不能结束（停机）呢，则if判断就会失败，从而选择另一个if分支并返回true，即Satan_algo(Satan_algo)又能够返回（停机）。

总之，我们有:

Satan_algo(Satan_algo)能够停机=> 它不能停机
Satan_algo(Satan_algo)不能停机=> 它能够停机

所以它停也不是，不停也不是。左右矛盾。

于是，我们的假设，即God_algo算法的存在性，便不成立了。正如拉格朗日所说：“陛下，我们不需要（上帝）这个假设”[4]。

这个证明相信每个程序员都能够容易的看懂。然而，这个看似不可捉摸的

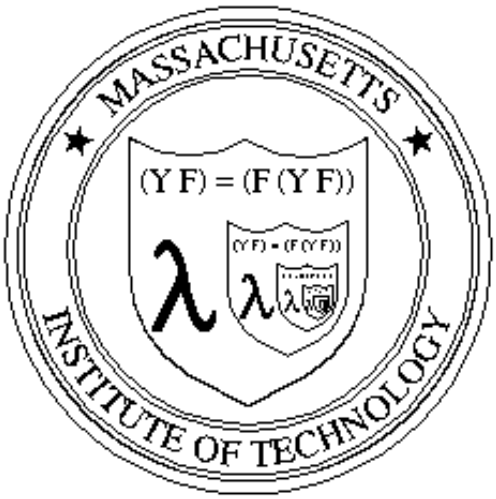
技巧背后其实隐藏着深刻的数学原理（甚至是哲学原理）。在没有认识到这一数学原理之前，至少我当时是对于图灵如何想出这一绝妙证明感到无法理解。但后面，在介绍完了与图灵的停机问题“同构”的Y combinator之后，我们会深入哥德尔的不完备性定理，在理解了哥德尔不完备性定理之后，我们从这一同样绝妙的定理出发，就会突然发现，离停机问题和神奇的Y combinator只是咫尺之遥而已。当然，最后我们会回溯到一切的尽头，康托尔那里，看看停机问题、Y combinator、以及不完备性定理是如何自然而然地由康托尔的对角线方法推导出来的，我们将会看到这些看似神奇的构造性证明的背后，其实是一个简洁优美的数学方法在起作用。

Y Combinator

了解Y combinator的请直接跳过这一节，到下一节“哥德尔的不完备性定理”。

让我们暂且搁下但记住绕人的图灵停机问题，走进函数式编程语言的世界，走进由跟图灵机理论等价的lambda算子发展出来的另一个平行的语言世界。让我们来看一看被人们一代一代吟唱着的神奇的Y Combinator...

关于Y Combinator的文章可谓数不胜数，这个由师从希尔伯特的著名逻辑学家Haskell B.Curry（Haskell语言就是以他命名的，而函数式编程语言里面的Curry手法也是以他命名）“发明”出来的组合算子（Haskell是研究组合逻辑(combinatory logic)的）仿佛有种神奇的魔力，它能够算出给定lambda表达式（函数）的不动点。从而使得递归成为可能。事实上，我们待会就会看到，Y Combinator在神奇的表面之下，其实隐藏着深刻的意义，其背后体现的意义，曾经开出过历史上最灿烂的数学之花，所以MIT的计算机科学系将它做成系徽也就不足为奇了[5]。



当然，要了解这个神奇的算子，我们需要一点点lambda算子理论的基础知识，不过别担心，lambda算子理论是我目前见过的最简洁的公理系统，这个系统仅仅由三条非常简单的公理构成，而这三条公理里面我们又只需要关注前两条。

以下小节——lambda calculus——纯粹是为了没有接触过lambda算子理论的读者准备的，并不属于本文重点讨论的东西，然而要讨论Y combinator就必须先了解一下lambda（当然，以编程语言来了解也行，但是你会看到，丘齐最初提出的lambda算子理论才是最最简洁和漂亮的，学起来也最省事。）所以我单独准备了一个小节来介绍它。如果你已经知道，可以跳过这一小节。不知道的读者也可以跳过这一小节去wikipedia上面看，这里的介绍使用了wikipedia上的方式

lambda calculus

先来看一下lambda表达式的基本语法(BNF)：

<expr> ::= <identifier>
<expr> ::= lambda <identifier-list>. <expr>
<expr> ::= (<expr> <expr>)

前两条语法用于生成lambda表达式（lambda函数），如：

lambda x y. x + y

haskell里面为了简洁起见用“\”来代替希腊字母lambda，它们形状比较相似。故而上面的定义也可以写成：
\ x y. x + y

这是一个匿名的加法函数，它接受两个参数，返回两值相加的结果。当然，这里我们为了方便起见赋予了lambda函数直观的计算意义，而实际上lambda calculus里面一切都只不过是文本替换，有点像C语言的宏。并且这里的“+”我们假设已经是一个具有原子语义的运算符[6]，此外，为了方便我们使用了中缀表达（按照lambda calculus系统的语法实际上应该写成“(+ x y)”才对——参考第三条语法）。

那么，函数定义出来了，怎么使用呢？最后一条规则就是用来调用一个lambda函数的：

((lambda x y. x + y) 2 3)

以上这一行就是把刚才定义的加法函数运用到2和3上（这个调用语法形式跟命令式语言(imperative language)惯用的调用形式有点区别，后者是“f(x, y)”，而这里是“(f x y)”，不过好在顺序没变:)）。为了表达简洁一点，我们可以给(lambda x y. x + y)起一个名字，像这样：

let Add = (lambda x y. x + y)

这样我们便可以使用Add来表示该lambda函数了：

(Add 2 3)

不过还是为了方便起见，后面调用的时候一般用“Add(2, 3)”，即我们熟悉的形式。

有了语法规则之后，我们便可以看一看这个语言系统的两条简单至极的公理了：

Alpha转换公理：例如，“lambda x y. x + y”转换为“lambda a b. a + b”。换句话说，函数的参数起什么名字没有关系，可以随意替换，只要函数体里面对参数的使用的地方也同时注意相应替换掉就是了。
Beta转换公理：例如，“(lambda x y. x + y) 2 3”转换为“2 + 3”。这个就更简单了，也就是说，当把一个lambda函数用到参数身上时，只需用实际的参数来替换掉其函数体中的相应变量即可。

就这些。是不是感觉有点太简单了？但事实就是如此，lambda算子系统从根本上其实就这些东西，然而你却能够从这几个简单的规则中推演出神奇无比的Y combinator来。我们这就开始！

递归的迷思

敏锐的你可能会发现，就以上这两条公理，我们的lambda语言中无法表示递归函数，为什么呢？假设我们要计算经典的阶乘，递归描述肯定像这样：

```
f(n):
 if n == 0 return 1
 return n*f(n-1)
```

当然，上面这个程序是假定n为正整数。这个程序显示了一个特点，f在定义的过程中用到了它自身。那么如何在lambda算子系统中表达这一函数呢？理所当然的想法如下：

```
lambda n. If_Else n==0 1 n*<self>(n-1)
```

当然，上面的程序假定了If_Else是一个已经定义好的三元操作符（你可以想象C的“?:”操作符，后面跟的三个参数分别是判断条件、成功后求值的表达式、失败后求值的表达式。那么很显然，这个定义里面有一个地方没

法解决，那就是<self>那个地方我们应该填入什么呢？很显然，熟悉C这类命令式语言的人都知道应该填入这个函数本身的名字，然而lambda算子系统里面的lambda表达式（或称函数）是没有名字的。

怎么办？难道就没有办法实现递归了？或者说，丘齐做出的这个lambda算子系统里面根本没法实现递归从而在计算能力上面有重大的缺陷？显然不是。马上你就会看到Y combinator是如何把一个看上去非递归的lambda表达式像变魔术那样变成一个递归版本的。在成功之前我们再失败一次，注意下面的尝试：

let F = lambda n. IF_Else n==0 1 n*F(n-1)

看上去不错，是吗？可惜还是不行。因为let F只是起到一个语法糖的作用，在它所代表的lambda表达式还没有完全定义出来之前你是不可以使用F这个名字的。更何况实际上丘齐当初的lambda算子系统里面也并没有这个语法元素，这只是刚才为了简化代码而引入的语法糖。当然，了解这个let语句还是有意义的，后面还会用到。

一次成功的尝试

在上面几次失败的尝试之后，我们是不是就一筹莫展了呢？别忘了软件工程里面的一条黄金定律：“任何问题都可以通过增加一个间接层来解决”。不妨把它沿用到我们面临的递归问题上：没错，我们的确没办法在一个lambda函数的定义里面直接（按名字）来调用其自身。但是，可不可以间接调用呢?

我们回顾一下刚才不成功的定义：

lambda n. If_Else n==0 1 n*<self>(n-1)

现在<self>处不是缺少“这个函数自身”嘛，既然不能直接填入“这个函数自身”，我们可以增加一个参数，也就是说，把<self>参数化：

lambda self n. If_Else n==0 1 n*self(n-1)

上面这个lambda算子总是合法定义了吧。现在，我们调用这个函数的时候，只要加传一个参数self，这个参数不是别人，正是这个函数自身。还是为了简单起见，我们用let语句来给上面这个函数起个别名：

let P = lambda self n. If_Else n==0 1 n*self(n-1)

我们这样调用，比如说我们要计算3的阶乘：

P(P, 3)

也就是说，把P自己作为P的第一个参数（注意，调用的时候P已经定义完毕了，所以我们当然可以使用它的名字了）。这样一来，P里面的self处不就等于是P本身了吗？自身调用自身，递归！

可惜这只是个美好的设想，还差一点点。我们分析一下P(P, 3)这个调用。利用前面讲的Beta转换规则，这个函数调用展开其实就是（你可以完全把P当成一个宏来进行展开！）：

IF_Else n==0 1 n*P(n-1)

看出问题了吗？这里的P(n-1)虽然调用到了P，然而只给出了一个参数；而从P的定义来看，它是需要两个参数的（分别为self和n）！也就是说，为了让P(n-1)变成良好的调用，我们得加一个参数才行，所以我们得稍微修改一下P的定义：

let P = lambda self n. If_Else n==0 1 n*self(self, n-1)

请注意，我们在P的函数体内调用self的时候增加了一个参数。现在当我们调用P(P, 3)的时候，展开就变成了：

IF_Else 3==0 1 3*P(P, 3-1)

而P(P, 3-1)是对P合法的递归调用。这次我们真的成功了！

不动点原理

然而，看看我们的P的定义，是不是很丑陋？“n*self(self, n-1)”？什么玩意？为什么要多出一个多余的self？DRY！怎么办呢？我们想起我们一开始定义的那个失败的P，虽然行不通，但最初的努力往往是大脑最先想到的最直观的做法，我们来回顾一下：

let P = lambda self n. If_Else n==0 1 n*self(n-1)

这个P的函数体就非常清晰，没有冗余成分，虽然参数列表里面多出一个self，但我们其实根本不用管它，看函数体就行了，self这个名字已经可以说明一切了对不对？但很可惜这个函数不能用。我们再来回想一下为什么不能用呢？因为当你调用P(P, n)的时候，里面的self(n-1)会展开为P(n-1)而P是需要两个参数的。唉，要是这里的self是一个“真正”的，只需要一个参数的递归阶乘函数，那该多好啊。为什么不呢？干脆我们假设出一个“真正”的递归阶乘函数：

```
power(n):
 if(n==0) return 1;
 return n*power(n-1);
```

但是，前面不是说过了，这个理想的版本无法在lambda算子系统中定义出来吗（由于lambda函数都是没名字的，无法自己内部调用自己）？不急，我们并不需要它被定义出来，我们只需要在头脑中“假设”它以“某种”方式被定义出来了，现在我们把这个真正完美的power传给P，这样：

P(power, 3)

注意它跟P(P, 3)的不同，P(P, 3)我们传递的是一个有缺陷的P为参数。而

P(power, 3)我们则是传递的一个真正的递归函数power。我们试着展开P(power, 3):

IF_Else 3==0 1 3*power(3-1)

发生了什么？？power(3-1)将会计算出2的阶乘（别忘了，power是我们设想的完美递归函数），所以这个式子将会忠实地计算出3的阶乘！

回想一下我们是怎么完成这项任务的：我们设想了一个以某种方式构造出来的完美的能够内部自己调用自己的递归阶乘函数power，我们发现把这个power传给P的话，P(power, n)的展开式就是真正的递归计算n阶乘的代码了。

你可能要说：废话！都有了power了我们还要费那事把它传给P来个P(power, n)干嘛？直接power(n)不就得了？！别急，之所以设想出这个power只是为了引入不动点的概念，而不动点的概念将会带领我们发现Y combinator。

什么是不动点？一点都不神秘。让我们考虑刚才的power与P之间的关系。一个是真正可递归的函数，一个呢，则是以一个额外的self参数来试图实现递归的伪递归函数，我们已经看到了把power交给P为参数发生了什么，对吧？不，似乎还没有，我们只是看到了，“把power加上一个n一起交给P为参数”能够实现真正的递归。现在我们想考虑power跟P之间的关系，直接把power交给P如何？

P(power)

这是什么？这叫函数的部分求值(partial evaluation)。换句话说，第一个参数是给出来了，但第二个参数还悬在那里，等待给出。那么，光给一个参数得到的是什么呢？是“还剩一个参数待给的一个新的函数”。其实也很简单，只要按照Beta转换规则做就是了，把P的函数体里面的self出现处皆替换为power就可以了。我们得到：

IF_Else n==0 1 n*power(n-1)

当然，这个式子里面还有一个变量没有绑定，那就是n，所以这个式子还不能求值，你需要给它一个n才能具体求值，对吧。这么说，这可不就是一个以n为参数的函数么？实际上就是的。在lambda算子系统里面，如果给一个lambda函数的参数不足，则得到的就是一个新的lambda函数，这个新的lambda函数所接受的参数也就是你尚未给出的那些参数。换句话来说，调用一个lambda函数可以分若干步来进行，每次只给出一部分参数，而只有等所有参数都给齐了，函数的求值结果才能出来，否则你得到的就是一个“中间函数”。

那么，这跟不动点定理有什么关系？关系大了，刚才不是说了，P(power)返回的是一个新的“中间函数”嘛？这个“中间函数”的函数体我们刚才已经看到了，就是简单地展开P(power)而已，回顾一遍：

IF_Else n==0 1 n*power(n-1)

我们已经知道，这是个函数，参数n待定。因此我们不妨给它加上一个“lambda n”的帽子，这样好看一点：

lambda n. IF_Else n==0 1 n*power(n-1)

这是什么呢？这可不就是power本身的定义？（当然，如果我们能够定义power的话）。不信我们看看power如果能够定义出来像什么样子：

let power = lambda n. IF_Else n==0 1 n*power(n-1)

一模一样！也就是说，P(power)展开后跟power是一样的。即：

P(power) = power

以上就是所谓的不动点。即对于函数P来说power是这样一个“点”：当把P用到power身上的时候，得到的结果仍然还是power，也就是说，power这个“点”在P的作用下是“不动”的。

可惜的是，这一切居然都是建立在一个不存在的power的基础上的，又有什么用呢？可别过早提“不存在”这个词，你觉得一样东西不存在或许只是你没有找到使它存在的正确方法。我们已经看到power是跟P有着密切联系的。密切到什么程度呢？对于伪递归的P，存在一个power，满足P(power)=power。注意，这里所说的“伪递归”的P，是指这样的形式：

let P = lambda self n. If_Else n==0 1 n*self(n-1) // 注意，不是self(self,n-1)

一般化的描述就是，对任一伪递归F（回想一下伪递归的F如何得到——是我们为了解决lambda函数不能引用自身的问题，于是给理想的f加一个self参数从而得到的），必存在一个理想f（F就是从这个理想f演变而来的），满足F(f) = f。

那么，现在的问题就归结为如何针对F找到它的f了。根据F和f之间的密切联系（F就比f多出一个self参数而已），我们可以从F得出f吗？假设我们可以（又是假设），也就是说假设我们找到了一根魔棒，把它朝任意一个伪递归的F一挥，眼前一花，它就变成了真正的f了。这根魔棒如果存在的话，它具有什么性质？我们假设这个神奇的函数叫做Y，把Y用到任何伪递归的函数F上就能够得到真正的f，也就是说：

Y(F) = f

结合上面的F(f) = f，我们得到：

Y(F) = f = F(f) = F(Y(F))

也就是说，Y具有性质：

Y(F) = F(Y(F))

性质倒是找出来了，怎么构造出这个Y却又成了难题。一个办法就是使用抽象法，这是从工程学的思想的角度，也就是通过不断迭代、重构，最终找到问题的解。然而对于这里的Y combinator，接近问题解的过程却显得复杂而费力，甚至过程中的有些点上的思维跳跃有点如羚羊挂角无迹可寻。然而，在这整个Y combinator介绍完了之后我们将会介绍著名的哥德尔不完备性定理，然后我们就会发现，通过哥德尔不完备性定理证明中的一个核心构造式，只需一步自然的推导就能得出我们的Y combinator。而且，最美妙的是，还可以再往下归约，把一切都归约到康托尔当初提出的对角线方法，到那时我们就会发现原来同样如羚羊挂角般的哥德尔的证明其实是对角线方法的一个自然推论。数学竟是如此奇妙，我们由简单得无法再简单的lambda calculus系统的两条公理居然能够导出如此复杂如此令人目眩神迷的Y Combinator，而这些复杂性其实也只是荡漾在定理海洋中的涟漪，拨开复杂性的迷雾我们重又发现它们居然寓于极度的简洁之中。这就是数学之美。

让我们先来看一看Y combinator的费力而复杂的工程学构造法，我会尽量让这个过程显得自然而流畅[7]:

我们再次回顾一下那个伪递归的求阶乘函数:

let P = lambda self n. If_Else n==0 1 n*self(n-1)

我们的目标是找出P的不动点power，根据不动点的性质，只要把power传给P，即P(power)，便能够得到真正的递归函数了。

现在，关键的地方到了，由于:

power = P(power) // 不动点原理

这就意味着，power作为一个函数（lambda calculus里面一切都是函

数），它是自己调用了自己的。那么，我们如何实现这样一个能够自己调用自己的power呢？回顾我们当初成功的一次尝试，要实现递归，我们是通过增加一个间接层来进行的：

let power_gen = lambda self. P(self(self))

还记得self(self)这个形式吗？我们在成功实现出求阶乘递归函数的时候不就是这么做的？那么对于现在这个power_gen，怎么递归调用？

power_gen(power_gen)

不明白的话可以回顾一下前面我们调用P(P, n)的地方。这里power_gen(power_gen)展开后得到的是什么呢？我们根据刚才power_gen的定义展开看一看，原来是：

P(power_gen(power_gen))

看到了吗？也就是说：

power_gen(power_gen) => P(power_gen(power_gen))

现在，我们把power_gen(power_gen)当成整体看，不妨令为power，就看得更清楚了：

power => P(power)

这不正是我们要的答案么？

OK，我们总结一下：对于给定的P，只要构造出一个相应的power_gen如下：

let power_gen = lambda self. P(self(self))

我们就会发现，power_gen(power_gen)这个调用展开后正是P(power_gen(power_gen))。也就是说，我们的power_gen(power_gen)就是我们苦苦寻找的不动点了！

铸造Y Combinator
现在我们终于可以铸造我们的Y Combinator了，Y Combinator只要生成一个形如power_gen的lambda函数然后把它应用到自身，就大功告成：

```
let Y = lambda F.
let f_gen = lambda self. F(self(self))
return f_gen(f_gen)
```

稍微解释一下，Y是一个lambda函数，它接受一个伪递归F，在内部生成一个f_gen（还记得我们刚才看到的power_gen吧），然后把f_gen应用到它自身（记得power_gen(power_gen)吧），得到的这个f_gen(f_gen)也就是F的不动点了（因为f_gen(f_gen) = F(f_gen(f_gen))），而根据不动点的性质，F的不动点也就是那个对应于F的真正的递归函数！

如果你还觉得不相信，我们稍微展开一下看看，还是拿阶乘函数说事，首先我们定义阶乘函数的伪递归版本：

```
let Pwr = lambda self n. If_Else n==0 1 n*self(n-1)
```

让我们把这个Pwr交给Y，看会发生什么（根据刚才Y的定义展开吧）：

```
Y(Pwr) =>
 let f_gen = lambda self. Pwr(self(self))
 return f_gen(f_gen)
```

Y(Pwr)的求值结果就是里面返回的那个f_gen(f_gen)，我们再根据f_gen的定义展开f_gen(f_gen)，得到：

Pwr(f_gen(f_gen))

也就是说：

Y(Pwr) => f_gen(f_gen) => Pwr(f_gen(f_gen))

我们来看看得到的这个Pwr(f_gen(f_gen))到底是不是真有递归的魔力。我们展开它（注意，因为Pwr需要两个参数，而我们这里只给出了一个，所以Pwr(f_gen(f_gen))得到的是一个单参（即n）的函数）：

Pwr(f_gen(f_gen)) => If_Else n==0 1 n*f_gen(f_gen) (n-1)

而里面的那个f_gen(f_gen)，根据f_gen的定义，又会展开为Pwr(f_gen(f_gen))，所以：

Pwr(f_gen(f_gen)) => If_Else n==0 1 n* Pwr(f_gen(f_gen)) (n-1)

看到加粗的部分了吗？因为Pwr(f_gen(f_gen))是一个接受n为参数的函数，所以不妨把它令成f（f的参数是n），这样上面的式子就是：

f => If_Else n==0 1 n*f(n-1)

完美的阶乘函数！

哥德尔的不完备性定理

了解哥德尔不完备性定理的可以跳到下一节，“大道至简——康托尔的天才”

然而，漫长的Y Combinator征途仍然并非本文的最终目的，对于Y combinator的构造和解释，只是给不了解lambda calculus或Y combinator

的读者看的。关键是马上你会看到Y combinator可以由哥德尔不完备性定理证明的一个核心构造式一眼瞧出来！

让我们的思绪回到1931年，那个数学界风起云涌的年代，一个名不经传的20出头的学生，在他的博士论文中证明了一个惊天动地的结论。

在那个年代，希尔伯特的数学天才就像太阳的光芒一般夺目，在关于数学严格化的大纷争中希尔伯特带领的形式主义派系技压群雄，得到许多当时有名望的数学家的支持。希尔伯特希望借助于形式化的手段，抽掉数学证明中的意义，把数学证明抽象成一堆无意义的符号转换，就连我们人类赖以自豪的逻辑推导，也不过只是一堆堆符号转换而已（想起lambda calculus系统了吧：））。这样一来，一个我们日常所谓的，带有直观意义和解释的数学系统就变成了一个纯粹由无意义符号表达的、公理加上推导规则所构成的形式系统，而数学证明呢，只不过是在这个系统内玩的一个文字游戏。令人惊讶的是，这样一种做法，真的是可行的！数学的意义，似乎竟然真的可以被抽掉！另一方面，一个形式系统具有非常好的性质，平时人们证明一个定理所动用的推导，变成了纯粹机械的符号变换。希尔伯特希望能够证明，在任一个无矛盾的形式系统中所能表达的所有陈述都要么能够证明要么能够证伪。这看起来是个非常直观的结论，因为一个结论要么是真要么是假，而它在它所处的领域/系统中当然应该能够证明或证伪了（只要我们能够揭示出该系统中足够多的真理）。

然而，哥德尔的证明无情的击碎了这一企图，哥德尔的证明揭示出，任何足够强到蕴含了皮亚诺算术系统（PA）的一致（即无矛盾）的系统都是不完备的，所谓不完备也就是说在系统内存在一个为真但无法在系统内推导出的命题。这在当时的数学界揭起了轩然大波，其证明不仅具有数学意义，而且蕴含了深刻的哲学意义。从那时起这一不完备性定理就被引申到自然科学乃至人文科学的各个角落...至今还没有任何一个数学定理居然能够产生这么广泛而深远的影响。

哥德尔的证明非常的长，达到了200多页纸，但其中很大的成分是用在了一些辅助性的工作上面，比如占据超过1/3纸张的是关于一个形式系统如何

映射到自然数，也就是说，如何把一个形式系统中的所有公式都表示为自然数，并可以从一自然数反过来得出相应的公式。这其实就是编码，在我们现在看来是很显然的，因为一个程序就可以被编码成二进制数，反过来也可以解码。但是在当时这是一个全新的思想，也是最关键的辅助性工作之一，另一方面，这正是“程序即数据”的最初想法。

现在我们知道，要证明哥德尔的不完备性定理，只需在假定的形式系统T内表达出一个为真但无法在T内推导出（证明）的命题。于是哥德尔构造了这样一个命题，用自然语言表达就是：命题P说的是“P不可在系统T内证明”（这里的系统T当然就是我们的命题P所处的形式系统了），也就是说“我不可以被证明”，跟著名的说谎者悖论非常相似，只是把“说谎”改成了“不可以被证明”。我们注意到，一旦这个命题能够在T内表达出来，我们就可以得出“P为真但无法在T内推导出来”的结论，从而证明T的不完备性。为什么呢？我们假设T可以证明出P，而因为P说的就是P不可在系统T内证明，于是我们又得到T无法证明出P，矛盾产生，说明我们的假设“T可以证明P”是错误的，根据排中律，我们得到T不可以证明P，而由于P说的正是“T不可证明P”，所以P就成了一个正确的命题，同时无法由T内证明！

如果你足够敏锐，你会发现上面这番推理本身不就是证明吗？其证明的结果不就是P是正确的？然而实际上这番证明是位于T系统之外的，它用到了一个关于T系统的假设“T是一致（无矛盾）的”，这个假设并非T系统里面的内容，所以我们刚才其实是在T系统之外推导出了P是正确的，这跟P不能在T之内推导出来并不矛盾。所以别担心，一切都正常。

那么，剩下来最关键的问题就是如何用形式语言在T内表达出这个P，上面的理论虽然漂亮，但若是P根本没法在T内表达出来，我们又如何能证明“T内存在这个为真但无法被证明的P”呢？那一切还不是白搭？

于是，就有了哥德尔证明里面最核心的构造，哥德尔构造了这样一个公式：

N(n) is unprovable in T

这个公式由两部分构成，n是这个公式的自由变量，它是一个自然数，一旦给定，那么这个公式就变成一个明确的命题。而N则是从n解码出的货真价实的（即我们常见的符号形式的）公式（记得哥德尔的证明第一部分就是把公式编码吗？）。”is unprovable in T”则是一个谓词，这里我们没有用形式语言而是用自然语言表达出来的，但哥德尔证明了它是可以用形式语言表达出来的，大致思路就是：一个形式系统中的符号数目是有限的，它们构成这个形式系统的符号表。于是，我们可以依次枚举出所有长度为1的串，长度为2的串，长度为3的串... 此外根据形式系统给出的语法规则，我们可以检查每个串是否是良构的公式（well formed formula，简称wff，其实也就是说，是否符合语法规则，前面我们在介绍lambda calculus的时候看到了，一个形式系统是需要语法规则的，比如逻辑语言形式化之后我们就会看到P->Q是一个wff，而->PQ则不是），因而我们就可以枚举出所有的wff来。最关键的是，我们观察到形式系统中的证明也不过就是由一个个的wff构成的序列（想想推导的过程，不就是一个公式接一个公式嘛）。而wff构成的序列本身同样也是由符号表内的符号构成的串。所以我们只需枚举所有的串，对每一个串检查它是否是一个由wff构成的序列（证明），如果是，则记录下这个wff序列（证明）的最后一个wff，也就是它的结论。这样我们便枚举出了所有的可由T推导出的定理。然后为了表达出”X is unprovable in T”，本质上我们只需说“不存在这样一个自然数S，它所解码出来的wff序列以X为终结”！这也就是说，我们表达出了“is unprovable in T”这个谓词。

我们用UnPr(X)来表达“X is unprovable in T”，于是哥德尔的公式变成了：

UnPr(N(n))

现在，到了最关键的部分，首先我们把这个公式简记为G(n)——别忘了G内有一个自由变量n，所以G现在还不是一个命题，而只是一个公式，所以谈不上真假：

G(n): UnPr(N(n))

又由于G也是个wff，所以它也有自己的编码g，当然g是一个自然数，现在我们把g作为G的参数，也就是说，把G里面的自由变量n替换为g，我们于是得到一个真正的命题：

G(g): UnPr(G(g))

用自然语言来说，这个命题G(g)说的就是“我是不可在T内证明的”。看，我们在形式系统T内表达出了“我是不可在T内证明的”这个命题。而我们一开始已经讲过了如何用这个命题来推断出G(g)为真但无法在T内证明，于是这就证明了哥德尔的不完备性定理[8]。

哥德尔的不完备性定理被称为20世纪数学最重大的发现（不知道有没有“之一”:)）现在我们知道为真但无法在系统内证明的命题不仅仅是这个诡异的“哥德尔命题”，还有很多真正有意义的明确命题，其中最著名的就是连续统假设，此外哥德巴赫猜想也有可能是个没法在数论系统中证明的真命题。

从哥德尔公式到Y Combinator

哥德尔的不完备性定理证明了数学是一个未完结的学科，永远有需要我们以人的头脑从系统之外去用我们独有的直觉发现的东西。罗杰·彭罗斯在《The Emperor's New Mind》中用它来证明人工智能的不可实现。当然，这个结论是很受质疑的。但哥德尔的不完备性定理的确还有很多很多的有趣推论，数学的和哲学上的。哥德尔的不完备性定理最深刻的地方就是它揭示了自指（或称自引用，递归调用自身等等）结构的普遍存在性，我们再来看一看哥德尔命题的绝妙构造：

G(n): UnPr(N(n))

我们注意到，这里的UnPr其实是一个形式化的谓词，它不一定要说“X在T内可证明”，我们可以把它泛化为一个一般化的谓词，P：

G(n): P(N(n))

也就是说，对于任意一个单参的谓词P，都存在上面这个哥德尔公式。然后我们算出这个哥德尔公式的自然数编码g，然后把它扔给G，就得到：

G(g): P(G(g))

是不是很熟悉这个结构？我们的Y Combinator的构造不就是这样一个形式？我们把G和P都看成一元函数，G(g)可不正是P这个函数的不动点么！于是，我们从哥德尔的证明里面直接看到了Y Combinator！

至于如何从哥德尔的证明联系到停机问题，就留给你去解决吧:) 因为更重要的还在后面，我们看到，哥德尔的证明虽然巧妙至极，然而其背后的思维过程仍然飘逸而不可捉摸，至少我当时看到G(n)的时候，“乃大惊”“不知所从出”，他怎么想到的？难道是某一个瞬间“灵光一现”？一般我是不信这一说的，已经有越来越多的科学研究表明一瞬间的“灵感”往往是潜意识乃至表层意识长期思考的结果。哥德尔天才的证明也不例外，我们马上就会看到，在这个神秘的构造背后，其实隐藏着某种更深的东西，这就是康托尔在19世纪80年代研究无穷集合和超限数时引入的对角线方法。这个方法仿佛有种神奇的力量，能够揭示出某种自指的结构来，而同时，这又是一个极度简单的手法，通过它我们能够得到数学里面一些非常奇妙的性质。无论是哥德尔的不完备性定理还是再后来丘齐建立的lambda calculus，抑或我们非常熟悉的图灵机理论里的停机问题，其实都只是这个手法简单推演的结果！

大道至简——康托尔的天才

“大道至简”这个名词或许更多出现在文学和哲学里面，一般用在一些模模糊糊玄玄乎乎的哲学观点上。然而，用在这里，数学上，这个名词才终于适得其所。大道至简，看上去最复杂的理论其实建立在一个最简单最纯粹的道理之上。

康托尔在无穷集合和超限数方面的工作主要集中在两篇突破性的论文上，这也是我所见过的最纯粹最美妙的数学论文，现代的数学理论充斥了太多复杂的符号和概念，很多时候让人看不到最本质的东西，当然，不否认这些东西很多也是有用的，然而，要领悟真正的数学美，像集合论和数论这种纯粹的东西，真的非常适合。不过这里就不过多谈论数学的细节了，只说康托尔引入对角线方法的动机和什么是对角线方法。

神奇的一一对应

康托尔在研究无穷集合的时候，富有洞察性地看到了对于无穷集合的大小问题，我们不能再使用直观的“所含元素的个数”来描述，于是他创造性地将一一对应引入进来，两个无穷集合“大小”一样当且仅当它们的元素之间能够构成一一对应。这是一个非常直观的概念，一一对应嘛，当然个数相等了，是不是呢？然而这同时就是它不直观的地方了。对于无穷集合，我们日常的所谓“个数”的概念不管用了，因为无穷集合里面的元素个数本就是无穷多个。不信我们来看一个小小的例子。我们说自然数集合能够跟偶数集合构成一一对应，从而自然数集合跟偶数集合里面元素“个数”是一样多的。怎么可能？偶数集合是自然数集合的真子集，所有偶数都是自然数，但自然数里面还包含奇数呢，说起来应该是二倍的关系不是？不是！我们只要这样来构造一一对应：

1 2 3 4 ...
2 4 6 8 ...

用函数来描述就是 f(n) = 2n。检验一下是不是一一对应的？不可思议对吗？还有更不可思议的，自然数集是跟有理数集一一对应的！对应函数的构造就留给你解决吧，提示，按如下方式来挨个数所有的有理数：

1/1 1/2 2/1 1/3 2/2 3/1 1/4 2/3 3/2 4/1 ...

用这种一一对应的手法还可以得到很多惊人的结论，如一条直线上所有的点跟一个平面上所有的点构成一一对应（也就是说复数集合跟实数集合构成一一对应）。以致于连康托尔自己都不敢相信自己的眼睛了，这也就是

为什么他在给戴得金的信中会说“我看到了它，却不敢相信它”的原因。

然而，除了一一对应之外，还有没有不能构成一一对应的两个无穷集合呢？有。实数集合就比自然数集合要“大”，它们之间实际上无法构成一一对应。这就是康托尔的对角线方法要解决的问题。

实数集和自然数集无法构成一一对应？！
我们只需将实数的小数位展开，并且我们假设实数集能够与自然数集一一对应，也就是说假设实数集可列，所以我们把它们与自然数一一对应列出，如下：

1 a10.a11a12a13...
2 a20.a21a22a23...
3 a30.a31a32a33...
4 ...
5 ...
（注：aij里面的ij是下标）

现在，我们构造一个新的实数，它的第i位小数不等于aii。也就是说，它跟上面列出的每一个实数都至少有一个对应的小数位不等，也就是说它不等于我们上面列出的所有实数，这跟我们上面假设已经列出了所有实数的说法相矛盾。所以实数集只能是不可列的，即不可与自然数集一一对应！这是对角线方法的最简单应用。

对角线方法——停机问题的深刻含义
对角线方法有很多非常奇妙的结论。其中之一就是文章一开始提到的停机问题。我想绝大多数人刚接触停机问题的时候都有一个问题，图灵怎么能够想到这么诡异的证明，怎么能构造出那个诡异的“说停机又不停机，说不停机又停机”的悖论机器。马上我们就会看到，这其实只是对角线方法的一个直接结论。

还是从反证开始，我们假设存在这样一个图灵机，他能够判断任何程序在

任何输入上是否停机。由于所有图灵机构成的集合是一个可列集（也就是说，我们可以逐一列出所有的图灵机，严格证明见我以前的一篇文章《图灵机杂思》），所以我们可以很自然地列出下表，它表示每个图灵机分别在每一个可能的输入（1,2,3,...）下的输出，N表示无法停机，其余数值则表示停机后的输出：

	1	2	3	4	...
M1	N	1	N	N	...
M2	2	0	N	0	...
M3	0	1	2	0	...
M4	N	0	5	N	...
...					

M1，M2，M3 ... 是逐一列出的图灵机，并且，注意，由于程序即数据，每个图灵机都有唯一编码，所以我们规定在枚举图灵机的时候Mi其实就代表编码为i的图灵机，当然这里很多图灵机将会是根本没用的玩意，但这不要紧。此外，最上面的一行1 2 3 4 ... 是输入数据，如，矩阵的第一行代表M1分别在1，2，3，...上面的输出，不停机的话就是N。

我们刚才假设存在这样一个图灵机H，它能够判断任何程序在任何输入上能否停机，换句话说，H(i,j)（i是Mi的编码）能够给出“Mi(j)”是N（不停）呢还是给出一个具体的结果（停）。

我们现在来运用康托尔的对角线方法，我们构造一个新的图灵机P，P在1上的输出行为跟M1(1)“不一样”，在2上的输出行为跟M2(2)“不一样”，...总之P在输入i上的输出跟Mi(i)不一样。只需利用一下我们万能的H，这个图灵机P就不难构造出来，如下：

```
P(i):
if( H(i, i) == 1 ) then // Mi(i) halts
return 1 + Mi(i)
else // if H(i, i) == 0 (Mi(i) doesn't halt)
```

return 0

也就是说，如果Mi(i)停机，那么P(i)的输出就是Mi(i)+1，如果Mi(i)不停机的话，P(i)就停机且输出0。这就保证了P(i)的输出行为跟Mi(i)反正不一样。现在，我们注意到P本身是一个图灵机，而我们上面已经列出了所有的图灵机，所以必然存在一个k，使得Mk = P。而两个图灵机相等当且仅当它们对于所有的输入都相等，也就是说对于任取的n，有Mk(n) = P(n)，现在令n=k，得到Mk(k)=P(k)，根据上面给出的P的定义，这实际上就是：

Mk(k) = P(k) =
1+Mk(k) if Mk(k) halts
0 if Mk(k) doesn't halt

看到这个式子里蕴含的矛盾了吗？如果Mk(k)停机，那么Mk(k)=1+Mk(k)；如果Mk(k)不停机，则Mk(k)=0（给出结果0即意味着Mk(k)停机）；不管哪种情况都是矛盾。于是我们得出，不存在那样的H。

这个对角线方法实际上说明了，无论多聪明的H，总存在一个图灵机的停机行为是它无法判断的。这跟哥德尔定理“无论多‘完备’的形式化公理系统，都存在一个‘哥德尔命题’是无法在系统内推导出来的”从本质上其实是一模一样的。只不过我们一般把图灵的停机问题称为“可判定问题”，而把数学的称为“可证明问题”。

等等！如果我们把那个无法判定是否停机的图灵机作为算法的特例纳入到我们的H当中呢？我们把得到的新的判定算法记为H1。然而，可惜的是，在H1下，我们又可以相应地以同样的手法从H1构造出一个无法被它（H1）判定的图灵机来。你再加，我再构造，无论你加多少个特例进去，我都可以由同样的方式构造出来一个你无法够到的图灵机，以彼之矛，攻彼之盾。其实这也是哥德尔定理最深刻的结论之一，哥德尔定理其实就说明了无论你给出多少个公理，即无论你建立多么完备的公理体系，这个系统里面都有由你的那些公理出发所推导不到的地方，这些黑暗的角落，就是人类直觉之光才能照射到的地方！

本节我们从对角线方法证明了图灵的停机问题，我们看到，对角线方法能够揭示出某种自指结构，从而构造出一个“悖论图灵机”。实际上，对角线方法是一种有深远影响的方法，哥德尔的证明其实也是这个方法的一则应用。证明与上面的停机问题证明如出一辙，只不过把Mi换成了一个形式系统内的公式fi，具体的证明就留给聪明的你吧:)我们现在来简单的看一下这个奇妙方法的几个不那么明显的推论。

罗素悖论

学过逻辑的人大约肯定是知道著名的罗素悖论的，罗素悖论用数学的形式来描述就是：

R = {X:X不属于X};

这个悖论最初是从康托尔的无穷集合论里面引申出来的。当初康托尔在思考无穷集合的时候发现可以称“一切集合的集合”，这样一个集合由于它本身也是一个集合，所以它就属于它自身。也就是说，我们现在可以称世界上存在一类属于自己的集合，除此之外当然就是不属于自己的集合了。而我们把所有不属于自己的集合收集起来做成一个集合R，这就是上面这个著名的罗素悖论了。

我们来看R是否属于R，如果R属于R，根据R的定义，R就不应该属于R。而如果R不属于R，则再次根据R的定义，R就应该属于R。

这个悖论促使了集合论的公理化。后来策梅罗公理化的集合论里面就不允许X属于X（不过可惜的是，尽管如此还是没法证明这样的集合论不可能产生出新的悖论。而且永远没法证明——这就是哥德尔第二不完备性定理的结论——一个包含了PA的形式化公理系统永远无法在内部证明其自身的一致（无矛盾）性。从而希尔伯特想从元数学推出所有数学系统的一致性的企图也就失败了，因为元数学的一致性又得由元元数学来证明，后者的一致性又得由元元元数学来证明...）。

这里我们只关心罗素是如何想出这个绝妙的悖论的。还是对角线方法！我们罗列出所有的集合，S1,S2,S3 ...

```
   S1  S2  S3 ...
S1 0   1  1 ...
S2 1   1   0 ...
S3 0   0   0 ...
...    ...
```

右侧纵向列出所有集合，顶行横向列出所有集合。0/1矩阵的(i,j)处的元素表示Si是否包含Sj，记为Si(j)。现在我们只需构造一个新的0/1序列L，它的第i位与矩阵的(i,i)处的值恰恰相反：L(i) = 1-Si(i)。我们看到，这个新的序列其实对应了一个集合，不妨也记为L，L(i)表示L是否包含Si。根据L的定义，如果矩阵的(i,i)处值为0（也就是说，如果Si不包含Si），那么L这个集合就包含Si,否则就不包含。我们注意到这个新的集合L肯定等于某个Sk（因为我们已经列出了所有的集合），L = Sk。既然L与Sk是同一集合，那么它们肯定包含同样的元素，从而对于任意n，有L(n) = Sk(n)。于是通过令n=k，得到L(k) = Sk(k)，而根据L的定义，L(k) = 1- Sk(k)。这就有Sk(k) = 1-Sk(k)，矛盾。

通过抽象简化以上过程，我们看到，我们构造的L其实是“包含了所有不包含它自身的集合的集合”，用数学的描述正是罗素悖论！

敏锐的你可能会注意到所有集合的数目是不可数的从而根本不能S1,S2...的一一列举出来。没错，但通过假设它们可以列举出来，我们发现了一个与可列性无关的悖论。所以这里的对角线方法其实可以说是一种启发式方法。

同样的手法也可以用到证明P(A)（A的所有子集构成的集合，也叫幂集）无法跟A构成一一对应上面。证明就留给聪明的你了:)

希尔伯特第十问题结出的硕果

希尔伯特是在1900年巴黎数学家大会上提出著名的希尔伯特第十问题的，简言之就是是否存在一个算法，能够计算任意丢番图方程是否有整根。要解决这个问题，就得先严格定义“算法”这一概念。为此图灵和丘齐分别提出了图灵机和lambda calculus这两个概念，它们从不同的角度抽象出了“有效（机械）计算”的概念，著名的图灵——丘齐命题就是说所有可以有效计算出来的问题都可以由图灵机计算出来。实际上我们已经看到，丘齐的lambda calculus其实就是数学推理系统的一个形式化。而图灵机则是把这个数学概念物理化了。而也正因为图灵机的概念隐含了实际的物理实现，所以冯·诺依曼才据此提出了奠定现代计算机体系结构的冯·诺依曼体系结构，其遵循的，正是图灵机的概念。而“程序即数据”的理念，这个发端于数学家哥德尔的不完备性定理的证明之中的理念，则早就在黑暗中预示了可编程机器的必然问世。

对角线方法——回顾

我们看到了对角线方法是如何简洁而深刻地揭示出自指或递归结构的。我们看到了著名的不完备性定理、停机问题、Y Combinator、罗素悖论等等等等如何通过这一简洁优美的方法推导出来。这一诞生于康托尔的天才的手法如同一条金色的丝线，把位于不同年代的伟大发现串联了起来，并且将一直延续下去...

P.S

1. lambda calculus里面的“停机问题”
实际上lambda calculus里面也是有“停机问题”的等价版本的。其描述就是：不存在一个算法能够判定任意两个lambda函数是否等价。所谓等价当然是对于所有的n,有f(n)=g(n)了。这个问题的证明更加能够体现对角线方法的运用。仍然留给你吧。
2. 负喧琐话(http://blog.csdn.net/g9yuayon)是个非常不错的blog:)。g9的文字轻松幽默，而且有很多名人八卦可以养眼，真的灰常...灰常...不错哦。此外g9老兄还是个理论功底非常扎实的牛。所以，anyway，看了他的blog就知道啦！最初这篇文章的动机也正是看了上面的一篇关于Y Combinator的铸造过程的介绍，于是想揭示一些更深的东西，于是便有了本文。
3. 文章起名《康托尔、哥德尔、图灵——永恒的金色对角线》其实是为了

纪念看过的一本好书GEB，即《Godel、Escher、Bach-An Eternal Golden Braid》中文译名《哥德尔、埃舍尔、巴赫——集异璧之大成》——商务印书馆出版。对于一本定价50元居然能够在douban上卖到100元的二手旧书，我想无需多说。另，幸福的是，电子版可以找到:)

4. 其实很久前想写的是一篇《从哥德尔到图灵》，但那篇写到1/3不到就搁下了，一是由于事务，二是总觉得少点什么。呵呵，如今把康托尔扯进来，也算是完成当时扔掉的那一篇吧。

5. 这恐怕算是写得最曲折的一篇文章了。不仅自己被这些问题搞得有点晕头转向（还好总算走出来），更因为要把这些东西自然而然的串起来，也颇费周章。很多时候是利用吃饭睡觉前或走路的时间思考本质的问题以及如何表达等等，然后到纸上一气呵成。不过同时也锻炼了不拿纸笔思考数学的能力，呵呵。

6. 关于图灵的停机问题、Y Combinator、哥德尔的不完备性定理以及其它种种与康托尔的对角线之间的本质联系，几乎查不到完整系统的深入介绍，一些书甚至如《The Emperor's New Mind》也只是介绍了与图灵停机问题之间的联系（已经非常的难得了），google和baidu的结果也是基本没有头绪。很多地方都是一带而过让人干着急。所以看到很多地方介绍这些定理和构造的时候都是弄得人晕头转向的，绝大部分人在面对如Y Combinator、不完备性定理、停机问题的时候都把注意力放在力图理解它是怎么运作的上面了，却使人看不到其本质上从何而来，于是人们便对这些东东大为惊叹。这使我感到很不痛快，如隔靴搔痒般。这也是写这篇文章的主要动机之一。

Reference

[1]《数学——确定性的丧失》

[2] 也有观点认为函数式编程语言之所以没有广泛流行起来是因为一些实际的商业因素。

[3] Douglas R.Hofstadter的著作《Godel, Escher, Bach: An Eternal Golden Braid》（《哥德尔、艾舍尔、巴赫——集异璧之大成》）就是围绕这一思想写出的一本奇书。非常建议一读。

[4]《数学——确定性的丧失》

[5] 虽然我觉得那个系徽做得太复杂，要表达这一简洁优美的思想其实还能

有更好的方式。
[6] 关于如何在lambda calculus系统里实现“+”操作符以及自然数等等，可参见这里，这里，和这里。
[7] g9的blog（负暄琐话）http://blog.csdn.net/g9yuayon/ 上有一系列介绍lambda calculus的文章（当然，还有其它好文章:)），非常不错，强烈推荐。最近的两篇就是介绍Y combinator的。其中有一篇以javaScript语言描述了迭代式逐步抽象出Y Combinator的过程。
[8] 实际上这只是第一不完备性定理，它还有一个推论，被称为第二不完备性定理，说的是任一个系统T内无法证明这个系统本身的一致性。这个定理的证明核心思想如下：我们前面证明第一不完备性定理的时候用的推断其实就表明 Con/T -> G(g) （自然语言描述就是，由系统T的无矛盾，可以推出G(g)成立），而这个“Con/T -> G(g)”本身又是可以在T内表达且证明出来的（具体怎么表达就不再多说了）——只需要用排中律即可。于是我们立即得到，T里面无法推出Con/T，因为一旦推出Con/T就立即推出G(g)从而推出UnPr(G(g))，这就矛盾了。所以，Con/T无法在T内推出（证明）。

本文来自CSDN博客，转载请标明出处：http://blog.csdn.net/pongba/archive/2006/10/15/1336028.aspx

分类: Algorithm