你好，类型

作者：**thzt**

目录

一、开篇

类型（type），是编程语言中一个经常被人们提及的概念，

当我们看待一门编程语言的时候，言必谈之类型系统（type system）。

它到底是显式类型的（explicit typing），还是隐式类型的（implicit typing），
是静态类型的（static typing），还是动态类型的（dynamic typing），
类型检查（type check）是较强的（stronger），还是较弱的（weaker）。

它是否支持高阶类型（high-order type），是否支持递归类型（recusive type），
是否支持子类型（subtype），是否支持多态（polymorphism）。

这些都是一个有主见的技术爱好者，乐于去了解的内容。

然而，我发现理解它们并不容易，我们欠缺最基本的数理逻辑和证明论相关的知识。
类型系统，可以看做是附着在语言语法之上的一套符号证明系统。

> In programming languages, a type system is a set of rules that assigns a property called type to the various constructs of a computer program, such as variables, expressions, functions or modules.

给表达式确定类型的过程，相当于对程序应该具备的属性做形式证明，
因此，数理逻辑是我们的朋友。

另一方面，从语义（semantics）角度对类型进行理解，我们会遇到更大的阻碍，
因为，这又涉及到了公理集合论和代数学相关的必备知识。

不过，这些仍然是一个有主见的技术爱好者，乐于去了解的内容。

本系列文章，我计划从无类型 λ 演算开始，逐步介绍简单类型（simply typed）λ 演算，
介绍递归类型和不动点（fixed point）之间关系，
介绍组合子逻辑（combinatory logic）。
然后，回归到本原，学习命题逻辑和一阶谓词逻辑相关的内容，
建立起逻辑学与类型理论之间的桥梁。

时间允许的话，我们还可以探讨模型论相关的内容，在补充了代数学相关的内容之后，

我们就可以讨论 CPO，Henkin 模型，Kripke 模型，以及笛卡儿闭范畴（CCC）了。

有的人说，讨论这些其实一点用都没有，
我只想说，作为一个有主见的技术爱好者，请别忘了咱们的初心是什么。

二、Lambda calculus

1. 匿名函数

现在很多种编程语言都支持匿名函数了，
例如，C# 3.0，C++ 11 和 Java 8 中的 lambda 表达式，
又例如，Python 2.2.2 中的 lambda，ECMAScript 3 的匿名函数，
ECMAScript 2015 的箭头函数（arrow function）等等。

更不论，Haskell，Lisp，Standard ML，这些函数式编程语言了。

越来越多的语言拥抱匿名函数，是因为在很多场景中，我们无需给函数事先指定一个名字，
并且结合词法作用域和高阶函数，会使某些问题用更直观的方式得以解决。

从理论上来讲，匿名函数具有和一般函数同样的计算能力，
使用某些技术手段，可以让匿名函数支持递归运算，从而完成任何图灵可计算的任务。

然而，要想理解这一切，我们首先还得静下心来，从基础的 λ 演算开始吧。

2. 自然数

λ 演算听起来是一个高大上的概念，实际上它只是一套“符号推导系统”，
人们首先定义某些合法的符号，然后再定义一些符号推导规则，
最后，就可以计算了，从一堆合法的符号得到另一堆，这种推导过程称之为“演算”。

为了让 λ 演算更容易被接受，我们暂时先岔开话题，看看自然数是怎么定义的。

2.1 Peano 系统

1889 年，皮亚诺（Peano）为了给出自然数的集合论定义，
他建立了一个包含 5 条公设的公理系统，后人称之为 Peano 系统。

Peano 系统是满足以下公设的有序三元组(M,F,e)，
其中 M 为一个集合，F 是 M 到 M 的函数，e 为首元素，

（1）$e \in M$
（2）M 在 F 下是封闭的
（3）e 不在 F 的值域中
（4）F 是单射
（5）如果 M 的子集 A 满足，$e \in A$，且 A 在 F 下封闭，则 $A=M$。

2.2 后继

设 A 为一个集合，我们称 $A \cup \{A\}$为 A 的后继，记作 $A+$，
求集合后继的操作，称为后继运算。

例如，$\varnothing+=\varnothing \cup \{\varnothing\}=\{\varnothing\}$
$\varnothing++=\varnothing+ \cup \{\varnothing+\}=\{\varnothing\} \cup \{\{\varnothing\}\}=\{\varnothing,\{\varnothing\}\}$
$\varnothing+++=\{\varnothing,\{\varnothing\},\{\varnothing,\{\varnothing\}\}\}$。

2.3 归纳集

设 A 为一个集合，若 A 满足，
（1）$\varnothing \in A$
（2）$\forall a \in A$，$a+ \in A$
则称 A 是归纳集。

例如，$\{\varnothing,\varnothing+,\varnothing++,\cdots\}$是一个归纳集。
从归纳集的定义可知，$\varnothing,\varnothing+,\varnothing++,\cdots$是所有归纳集的元素，
于是，可以将它们定义为自然数，自然数集记为 N。

设 $\sigma:N \to N$，满足 $\sigma(n)=n+$，则称 σ 为后继函数，
则可以证明$(N,\sigma,\varnothing)$是一个 Peano 系统。

3. λ 演算

λ 演算，是 1930 年由邱奇（Alonzo Church）发明的一套形式系统，
它是从具体的函数定义，函数调用和函数复合中，抽象出来的数学概念。

3.1 语法

形式上，λ 演算由 3 种语法项（term）组成，
（1）一个变量 x 本身，是一个合法的 λ 项，
（2）$\lambda x.t1$，是一个合法的 λ 项，称为从项 $t1$ 中抽象出 x，
（3）$t1t2$，是一个合法的 λ 项，称为将 $t1$ 应用于 $t2$。

例如，$(\lambda x.(xy))$，$(x(\lambda x.(\lambda x.x)))$,，$((\lambda y.y)(\lambda x.(xy)))$，都是合法的 λ 项。
为了简化描述，我们通常会省略一些括号，以上三个 λ 项可以写成，
$\lambda x.xy$，$x(\lambda x.\lambda x.x)$,，$(\lambda y.y)(\lambda x.xy)$，
对于形如 $\lambda x.t1$ 的 λ 项来说，“.”后面会向右包含尽量多的内容。

现在我们有了一堆合法的字符串了。
可是，在给定推导规则之前，这些字符串之间都是没有关联的。
而且，我们也还没有为这些符号指定语义，它们到底代表什么也是不清楚的。

很显然给这些符号指定不同的推导规则，会得到不同的公理系统，
在众多 λ 演算系统中，最简单的是 $\lambda\beta$ 系统，它指定了 α 和 β 两种变换。

3.2 α 变换

设 λ 项 P 中包含了 $\lambda x.M$，
则我们可以把 M 中所有自由出现的 x，全都换成 y，即 $\lambda y.[y/x]M$，
这种更名变换，称为 α 变换。

其中，“自由出现”指的是 x 不被其他 λ 抽象所绑定，
例如，$\lambda x.xy$ 中，y 是自由的，
而 x 就不是自由的，因为它被 $\lambda x.$ 绑定了。

如果 P 可以经过有限步 α 变换转换为 Q，就写为 $P\equiv\alpha Q$。

例如，
$\lambda xy.x(xy)=\lambda x.(\lambda y.x(xy))$
$\equiv\alpha\lambda x.(\lambda v.x(xv))$
$\equiv\alpha\lambda u.(\lambda v.u(uv))$
$=\lambda uv.u(uv)$

3.3 β 变换

形如$(\lambda x.M)N$的 λ 项，可以经由 β 变换转换为$[N/x]M$，
指的是，把 M 中所有自由出现的 x 都换成 N。

如果 P 可以经过有限步 β 变换转换为 Q，就写为 $P \triangleright \beta Q$。

例如，
$(\lambda x.x(xy))N \triangleright \beta N(Ny)$
$(\lambda x.xx)(\lambda x.xx) \triangleright \beta [(\lambda x.xx)/x](xx) = (\lambda x.xx)(\lambda x.xx) \triangleright \beta \cdots$

我们发现，某些 λ 项，可以无限进行 β 变换。
而那些最终会终止的 β 变换的结果，称为 β 范式（β normal form）。

3.4 邱奇编码

现在我们有 $\lambda\beta$公理系统了，就可以依照 α 或 β 变换，对任意合法的 λ 项进行变换。

假设我们有一个 λ 项，$\lambda f.\lambda x.x$，
还有另外一个 λ 项，$\lambda n.\lambda f.\lambda x.f(nfx)$，记为 $succ$，
我们来计算，$succ(\lambda f.\lambda x.x)$，
可得，$(\lambda n.\lambda f.\lambda x.f(nfx))(\lambda f.\lambda x.x) \triangleright \beta \lambda f.\lambda x.fx$，
我们再运用一次 $succ$，$succ(\lambda f.\lambda x.fx) \triangleright \beta \lambda f.\lambda x.f(fx)$。

我们发现每次应用 $succ$，都会给 $\lambda f.\lambda x.x$ 中加一个 f，
最终我们可以得到以下这些 λ 项，
$\lambda f.\lambda x.x$
$\lambda f.\lambda x.fx$
$\lambda f.\lambda x.f(fx)$
$\lambda f.\lambda x.f(f(fx))$
$\cdots$
$\lambda f.\lambda x.fnx$

如果我们记 $\lambda f.\lambda x.x \equiv 0$，$\lambda f.\lambda x.fx \equiv 1$，$\cdots$，
$\lambda f.\lambda x.fnx \equiv n$，
我们就得到了自然数的另一种表示方式，称之为邱奇编码。

可以看到邱奇编码与归纳集之间有异曲同工之妙。

3.5 语义

到目前为止，我们并未谈及 λ 项到底表示什么含义，
虽然 $\lambda x.M$ 看起来像是函数定义，$(\lambda x.M)N$ 看起来像是函数调用。

我们谨慎的使用公理化方法，从什么是合法的 λ 项出发，
定义 $\lambda\beta$系统中的公理——合法的 λ 项，
然后又指定了该系统中的推导规则——α 和 β 变换，
最终得到了一个形式化的公理系统（公理+推导规则）。

后文中，我们将谈及 λ 项的语义，然后再逐渐给它加上类型。

参考

离散数学教程
Lambda-Calculus and Combinators，an Introduction
Lecture Notes on the Lambda Calculus

三、Combinatory logic

回顾

上一篇中，我们介绍了 λ 演算，
它是由一堆合法的符号和一些推导规则构成的公理系统，
在众多 λ 演算中，我们介绍了最常用的 $\lambda\beta$系统，
它指定了 α 和 β 两种对 λ 项的变换规则。

作为形式系统，上一篇中，我们展现了它的编码能力，
将邱奇编码，与公理集合论中自然数的归纳集定义，进行了对比。

本文我们将介绍另一套形式系统，
组合子逻辑（combinatory logic）。

1. 组合子逻辑

CL（组合子逻辑），与 λ 演算很相似，只是不需要对变量进行绑定，
和函数作用在值上不同的是，组合子作用在函数上，从而生成另一个函数。
例如，我们可以定义一个组合子 B，使得$(B(f,g))(x)=f(g(x))$，其中，f 和 g 都是函数。

为了避免过早的谈及语义，我们和 λ 演算一样，使用公理化的方法来定义它，
首先我们要说明什么是公理，即什么是合法的 CL 项，
（1）所有的变量，常量，以及组合子 I，K，S，都是合法的 CL 项，
（2）如果 X 和 Y 是合法的 CL 项，那么(XY)也是。

例如，以下字符串都是合法的 CL 项，
$((S(KS))K)$，$((S(Kv0))((SK)K))$。

同样为了简化，某些情况下括号是可以省略的，如果我们默认各个 CL 项都是左结合的，
因此，$(((UV)W)X)$可以简写为 $UVWX$。

2. Weak reduction

现在，我们要完成公理化的第二步了，那就是给合法的 CL 项制定变换规则，
在 CL（组合子逻辑中）中，我们称之为 weak reduction，即我们令，

（1）IX，可以变换为 X，
（2）KXY，可以变换为 X，
（3）$SXYZ$，可以变换为 $XZ(YZ)$。

如果 U 和经过有限步 weak reduction 转换为 V，就写为 $U \triangleright wV$。
与 λ 项的 β 范式一样，我们将不能再继续进行 weak reduction 的 CL 项，称为 weak 范式（weak normal form）。

我们来看一个例子，
设 $B=S(KS)K$，来计算 $BXYZ$，
$BXYZ=S(KS)KXYZ$
$>wKSX(KX)YZ$，因为 $S(KS)KX \triangleright wKSX(KX)$，
$>wS(KX)YZ$，因为 $KSX \triangleright wS$，
$>wKXZ(YZ)$，
$>wX(YZ)$。

有了合法的 CL 项（公理），以及 weak reduction（推导规则），
我们就建立了另一个形式系统 CLw。

3. CL 与 λ 演算之间的关系

Notation	*Meaning for* λ	*Meaning for CL*
term	λ-term	CL-term
$X \equiv Y$	$X \equiv_\alpha Y$	X is identical to Y
$X \triangleright_{\beta,w} Y$	$X \triangleright_\beta Y$	$X \triangleright_w Y$
$X =_{\beta,w} Y$	$X =_\beta Y$	$X =_w Y$
λx	λx	$[x]$

以上我们看到 CL 项，似乎只能进行项的应用（application）操作，对应于 λ 项的用法为 (MN)，
然而，其实 CL 的威力却不止于此，它的计算能力是与 λ 演算相当的。

为了证明等价性，建立 CL 项与 λ 项之间的关系，
现在我们用 I，K，S 三个组合子，来定义与 $\lambda x.M$ 相似的概念。

对于任意的 CL 项 M，以及任意的变量 x，我们定义$[x].M$ 用如下方式表示，

（1）$[x].M=KM$，如果 M 中不含有 x，

（2）$[x].x=I$，

（3）$[x].Ux=U$，如果 U 中不含有 x，

（4）$[x].UV=S([x].U)([x].V)$，如果（1）和（3）都不适用的话。

例如，

$[x].xy=S([x].x)([x].y)=SI(Ky)$

可见，$[x].M$ 可以完全用 I，K，S 三个组合子来构建出来，
它表示了与 $\lambda x.M$ 相对应的概念。

因此，我们可以建立 λ 项与 CL 项的对应关系了，
在 CL 中，$X=Y$，相当于 λ 演算中，$X\equiv\alpha Y$，可以统一记为 $X\equiv Y$
$X\triangleright wY$，相当于 $X\triangleright\beta Y$，可以统一记为 $X\triangleright\beta,wY$。

相应的，I，K，S 也可以使用 λ 项来表示，
$I=\lambda x.x$，$K=\lambda xy.x$，$S=\lambda xyz.xz(yz)$。

4. 不动点定理

在 λ 演算和 CL 中，存在组合子 Y，使得 $Yx\triangleright\beta,wx(Yx)$。

证明：令 $U=\lambda ux.x(uux)$，$Y=UU$，则，
$Yx=(\lambda u.(\lambda x.x(uux)))Ux=x(UUx)=x(Yx)$。

总结

本文我们用公理化的方法，创建了另一个形式系统 CLw，
接着，我们发现 CLw实际上是与 $\lambda\beta$等价的。

可悲的是，知道 λ 演算的人很多，但是知道 CL（组合子逻辑）的人却很少，这简直是不可思议的。
下文中，我们将继续沿着公理化和形式系统的道路向前走，敲开数理逻辑的大门。

参考

Lambda-Calculus and Combinators，an Introduction

四、Propositional logic

前两篇中，我们介绍了 λ 演算和 CL（组合子逻辑），

我们采用了公理化的方法，先定义系统中的公理，然后定义推导规则，
最终得到了两个形式系统，$\lambda\beta$系统以及 CLw系统。

值得注意的是，公理系统不仅仅包含由公理和推导构成的形式系统，
还包含给这个形式系统所选择的语义。

给形式系统选择一个可靠的语义，是复杂的，我们将在后文再详细介绍。
从这一篇开始，我们先开始介绍数理逻辑，
看看逻辑学是怎么看待形式化问题的。

1. 命题逻辑形式系统

下文中，我们采用与 $\lambda\beta$系统，CLw系统相同的方式，
来介绍命题逻辑（propositional logic）。

命题逻辑，是在研究命题的证明和推理的过程中抽象出来的，
先不考虑语义，仅仅从符号的角度（形式化）来考虑它，则是更简单直接的。

1.1 公理和推导规则

首先我们给出命题逻辑形式系统中的公理，
（1）$\alpha\to(\beta\to\alpha)$
（2）$(\alpha\to(\beta\to\gamma))\to((\alpha\to\beta)\to(\alpha\to\gamma))$
（3）$(\neg\alpha\to\neg\beta)\to(\beta\to\alpha)$

然后，我们给出命题逻辑形式系统中的推导规则，

（1）$\beta\alpha,\alpha\to\beta$

以上推导规则可以理解为，
如果 α 和 $\alpha\to\beta$ 成立，则 β 成立。

这里我们采用了推导规则的常用写法，$CP1,P2,\cdots,Pn$，
横线上面的部分“$P1,P2,\cdots,Pn$”称为规则的**前提**（premise），

横线下面的部分“C”，称为**结论**（conclusion）。

在命题逻辑中，根据公理和推导规则，得到的公式称为**定理**。
根据公理，定理和推导规则，得到的公式也是定理。

1.2 例子

下面我们来看一个例子，
求证：$(\alpha\to\beta)\to(\alpha\to\alpha)$是一个定理。

证明：
首先，我们使用公理（1），我们有下式成立，
$\alpha\to(\beta\to\alpha)$

然后，我们使用公理（2），并令其中的 $\gamma=\alpha$，
$(\alpha\to(\beta\to\alpha))\to((\alpha\to\beta)\to(\alpha\to\alpha))$

最后，我们结合以上两个结论，再使用推导规则（1），就得到了，
$(\alpha\to\beta)\to(\alpha\to\alpha)$

证毕。

2. Hilbert-style 和 Gentzen-style

以上的证明过程中，每一行断言，是一个不包含条件的定理，
这种风格的演绎系统（deduction system）称为具有 **Hilbert-style**。

如果 Γ,Δ 是有限的公式集合，则，
$\Gamma\vdash\Delta$，称为一个**序贯**（sequent）。
其中，Γ 称为序贯的前提（antecedent），Δ 称为序贯的**结论**（succedent）。
它表示，如果公式集 Γ 都成立，那么 Δ 中至少有一个成立。

如果证明过程中，每一行断言是一个序贯，
这种风格的演绎系统，称为具有 **Gentzen-style**。

Hilbert-style 演绎系统，通常具有较多的公理，但是具有较少的推导规则，

Gentzen-style 演绎系统，则反之，具有较少的公理，却具有较多的推导规则。

如果 Δ 中总是只包含一个公式，
则称该演绎系统为**自然演绎系统**（natural deduction system）。

如果有限序列，$\Gamma_1 \vdash \alpha_1, \Gamma_2 \vdash \alpha_2, \cdots, \Gamma_n \vdash \alpha_n$，满足，
（1）$\Gamma_1, \Gamma_2, \cdots, \Gamma_n$为有限公式集
（2）$\alpha_1, \alpha_2, \cdots, \alpha_n$为公式
（3）每个 $\Gamma_i \vdash \alpha_i$，$(1 \leqslant i \leqslant n)$，都是它之前若干个 $\Gamma_j \vdash \alpha_j$，$(1 \leqslant j < i \leqslant n)$，应用某条推导规则得到的

我们就称这个有限序列，为 $\Gamma_n \vdash \alpha_n$的一个**（形式）证明序列**。
此时，也称 α_n可由 Γ_n**（形式）证明**。

3. 命题逻辑的自然演绎系统

以上定义的形式系统，称为命题逻辑形式系统 P，
下面我们再定义一个与之等价的，命题逻辑的自然演绎系统 N。

3.1 语法

（1）可数个命题符号：$p_1, p_2, \cdots$
（2）5 个联接词符号：$\neg, \vee, \wedge, \rightarrow, \leftrightarrow$
（3）2 个辅助符号：$), ($

3.2 合法的公式

我们使用 BNF 来定义，

$\alpha ::= p \mid (\neg \alpha) \mid (\alpha_1 \vee \alpha_2) \mid (\alpha_1 \wedge \alpha_2) \mid (\alpha_1 \rightarrow \alpha_2) \mid (\alpha_1 \leftrightarrow \alpha_2)$

3.3 推导规则

$$（1）包含律：\Gamma \vdash \alpha \quad \alpha \in \Gamma$$

（2）¬消去律：$\Gamma\vdash\alpha\Gamma,\neg\alpha\vdash\beta;\Gamma,\neg\alpha\vdash\neg\beta$

（3）→消去律：$\Gamma\vdash\beta\Gamma\vdash(\alpha\to\beta);\Gamma\to\alpha$

（4）→引入律：$\Gamma\vdash\alpha\to\beta\Gamma,\alpha\vdash\beta$

（5）∨消去律：$\Gamma,\alpha\vee\beta\vdash\gamma\Gamma,\alpha\vdash\gamma;\Gamma,\beta\vdash\gamma$

（6）∨引入律：$\Gamma\vdash\alpha\vee\beta;\Gamma\vdash\beta\vee\alpha\Gamma\vdash\alpha$

（7）∧消去律：$\Gamma\vdash\alpha;\Gamma\vdash\beta\Gamma\vdash\alpha\wedge\beta$

（8）∧引入律：$\Gamma\vdash\alpha\wedge\beta\Gamma\vdash\alpha;\Gamma\vdash\beta$

（9）↔消去律：$\Gamma\vdash\beta\Gamma\vdash\alpha\leftrightarrow\beta;\Gamma\vdash\alpha$，$\Gamma\vdash\alpha\Gamma\vdash\alpha\leftrightarrow\beta;\Gamma\vdash\beta$

（10）↔引入律：$\Gamma\vdash\alpha\leftrightarrow\beta\Gamma,\alpha\vdash\beta;\Gamma,\beta\vdash\alpha$

3.4 例子

以上，我们定义了命题逻辑的自然演绎系统 N，
它比命题逻辑形式系统 P 更复杂一些，但是这两个系统是等价的。

我们来看一个例子。
求证，$\alpha\to\beta,\alpha\vdash\beta$ 是一个定理。

证明：
首先，我们根据包含律有，
$\alpha\to\beta,\alpha\vdash\alpha\to\beta$
$\alpha\to\beta,\alpha\vdash\alpha$

然后，根据这两个结论，以及→消去律，就有，
$\alpha\to\beta,\alpha\vdash\beta$。

证毕。

4. 系统 N 和 P 的等价性

本文我们介绍了两个形式系统，
命题逻辑形式系统 P，以及命题逻辑的自然演绎系统 N，
可以证明，对于 P（或 N）中的公式 α，$\vdash N\alpha$ 当且仅当$\vdash P\alpha$。

因此，这两个系统中的定理集是一样的，
某个定理在 P 中可证，当且仅当在 N 中也可证。

其中，我们令，
$\neg\alpha\rightarrow\beta$，表示 $\alpha\vee\beta$，
$\neg(\neg\alpha\vee\neg\beta)$，表示 $\alpha\wedge\beta$，
$(\alpha\rightarrow\beta)\wedge(\beta\rightarrow\alpha)$，表示 $\alpha\leftrightarrow\beta$。

对于这两个系统而言，如上文所示，
我们只进行了符号的推导操作，即对公式进行形式证明，
至于这些符号到底代表什么意思，我们却故意没有提及。

我们并没有说 $\vee$ 表示“或”，也没有说 $\wedge$ 表示“与”，
在学数理逻辑的时候，一开始就将形式系统与它的语义模型相区分，
是非常有益的，后文我们将看到这样做的好处。

5. 总结

本文介绍了命题逻辑，以及与之相关的两个形式系统 P 和 N，
和 $\lambda\beta$，CLw一样，我们采用了公理化的方式构建它们，
这样得到的形式系统，只是符号演算，还没有被赋予特定的语义，
下文我们开始介绍一阶谓词逻辑。

参考

离散数学教程
数理逻辑
Proof calculus
Hilbert system
Natural deduction
Sequent calculus
Practical Foundations for Programming Languages
Lambda-Calculus and Combinators，an Introduction

五、Predicate logic

从形式系统的角度来看，

一阶谓词逻辑，只是比命题逻辑多添加了一些公理，
或者多添加了一些推导规则，
然而，这样的举动，却会让形式系统截然不同。

欧几里得第五公设，是一个公理，无法由前四个公设推导证明，
在原来的欧氏几何中去掉它，然后添加上不同的第五公设，就变成了不同的几何，
黎曼几何与闵可夫斯基几何。

因此，不同的公理和推导规则，构成了不同的形式系统，
哪怕是有很小的变化。

本文我们来扩充前一篇中提到的，命题逻辑形式系统 P，
以及，命题逻辑的自然演绎系统 N。

1. 一阶谓词逻辑形式系统

1.1 公理和推导规则

命题逻辑形式系统 P，只有三条公理，一条推导规则，
以下我们保持推导规则不变，为它添加四条公理，
这样得到的系统，我们记为 KL。

公理：
（1）~（3）与命题逻辑形式系统 P 相同
（4）$\forall x\alpha \to \alpha(x/t)$，若 t 对 x 在 α 中自由出现
（5）$\alpha \to \forall x\alpha$，若 x 不在 α 中自由出现
（6）$\forall x(\alpha \to \beta) \to (\forall x\alpha \to \forall x\beta)$
（7）若 α 是 KL的一个公理，则 $\forall x\alpha$ 也为 KL的一个公理 P

推导规则：
（1）与命题逻辑形式系统 P 相同

1.2 辖域和自由出现

其中（5）和（6）中提到了“自由出现”的概念，
现在解释如下，

我们称公式$\forall x\alpha$中的α，为量词$\forall x$的辖域。
变元符号x在公式α中的某处出现，如果是在量词$\forall x$的辖域内，
则称为约束出现，否则称为自由出现。

例如，$\forall x1F(x1)\to F(x1)$，
由于→的优先级比较低，上式相当于$(\forall x1F(x1))\to F(x1)$，
所以，第一个$x1$是约束出现，第二个$x1$是自由出现。

1.3 例式

上述公理（4）中，出现了$\alpha(x/t)$，它称为公式α的例式。
它表示，将α中每一个自由出现的x，都替换为项t之后，得到的公式。

1.4 简写

我们记，
$\alpha\vee\beta$，为$\neg\alpha\to\beta$的简写，
$\alpha\wedge\beta$，为$\neg(\neg\alpha\vee\neg\beta)$的简写，
$\alpha\leftrightarrow\beta$，为$(\alpha\to\beta)\wedge(\beta\to\alpha)$的简写，
$\exists x\alpha$，为$\neg\forall x(\neg\alpha)$的简写。

1.5 举个例子

求证：设项t对变元符号x在α中自由，则$\alpha(x/t)\to\exists x\alpha$是$KL$是一条定理。

证明：
因为$(\neg\alpha)(x/t)=\neg(\alpha(x/t))$，
又根据公理（4），$\forall x(\neg\alpha)\to(\neg\alpha)(x/t)$，
所以，$\forall x(\neg\alpha)\to\neg(\alpha(x/t))$。

又根据公理（3），$(\neg\alpha\to\neg\beta)\to(\beta\to\alpha)$，
$\forall x(\neg\alpha)\to\neg(\alpha(x/t))\to(\alpha(x/t)\to\neg\forall x(\neg\alpha))$

再根据推导规则（1），$\beta\alpha\to\beta,\alpha$，
$\alpha(x/t)\to\neg\forall x(\neg\alpha)$，
即，$\alpha(x/t)\to\exists x\alpha$。

证毕。

2. 一阶谓词逻辑的自然演绎系统

上一篇中我们介绍了命题逻辑的自然演绎系统 N，
而且，我们知道了 N 是与命题逻辑形式系统 P 是等价的。

现在我们扩充 N，得到一阶谓词逻辑的自然演绎系统 NL，
它与一阶谓词逻辑形式系统 KL也是等价的。

KL和 P 一样，是 Hilbert-style 演绎系统，
它们具有更多的公理，更少的推导规则，
而 NL和 N，是 Gentzen-style 自然演绎（natural dedution）系统，
它们具有更少的公理，更多的推导规则。

由于 NL是对 N 的扩充，
下文中，我们只列出新增的公理和推导规则。

2.1 公理和推导规则

公理：
公理集仍为空集。

推导规则：
（1）~（10）与命题逻辑的自然演绎系统 N 相同
（11）增加前提律：$\Gamma,\beta\vdash\alpha\Gamma\vdash\alpha$
（12）$\forall$ 消去律：$\Gamma\vdash\alpha(x/t)\Gamma\vdash\forall x\alpha$，$t$ 对 x 在 α 中自由出现

（13）∀引入律：$\Gamma\vdash\forall x\alpha\Gamma\vdash\alpha$

（14）∃消去律：$\Gamma\cup\{\exists x\alpha\}\vdash\beta\Gamma,\alpha\vdash\beta$，$x$ 不在 $\Gamma\cup\{\beta\}$中自由出现

（15）∃引入律：$\Gamma\vdash\exists x\alpha\Gamma\vdash\alpha(x/t)$，$t$ 对 x 在 α 中自由出现

2.2 举个例子

求证：$\forall x(\alpha\leftrightarrow\beta)\vdash\forall x\alpha\leftrightarrow\forall x\beta$

证明：
根据规则（1）包含律，
（1）$\forall x(\alpha\leftrightarrow\beta),\forall x\alpha\vdash\forall x(\alpha\leftrightarrow\beta)$，
（2）$\forall x(\alpha\leftrightarrow\beta),\forall x\alpha\vdash\forall x\alpha$，

根据规则（12）∀消去律，
（3）$\forall x(\alpha\leftrightarrow\beta),\forall x\alpha\vdash\alpha\leftrightarrow\beta$，
（4）$\forall x(\alpha\leftrightarrow\beta),\forall x\alpha\vdash\alpha$，

式（3）（4），根据规则（9）↔消去律，
（5）$\forall x(\alpha\leftrightarrow\beta),\forall x\alpha\vdash\beta$，

式（4）（5），根据规则（13）∀引入律，
（6）$\forall x(\alpha\leftrightarrow\beta),\forall x\alpha\vdash\forall x\alpha$，
（7）$\forall x(\alpha\leftrightarrow\beta),\forall x\alpha\vdash\forall x\beta$，

式（6）（7），根据规则（10）↔引入律，
（8）$\forall x(\alpha\leftrightarrow\beta),\forall x\alpha\vdash\forall x\alpha\leftrightarrow\forall x\beta$。

证毕。

3. 总结

本文介绍了两种风格的一阶谓词逻辑演算系统，
其中 KL是 Hilbert-style 演绎系统，
NL是 Gentzen-style 自然演绎系统，
可以证明它们是等价的。

结合上一篇，我们已经从形式系统的角度介绍了命题逻辑和一阶谓词逻辑，
我们有了足够的基础之后，就可以给 $\lambda\beta$系统加上类型，
开始介绍简单类型 λ 演算了。

参考

离散数学教程

六、Simply typed lambda calculus

简单类型化 λ 演算（simply typed lambda calculus）$\lambda\to$，

是无类型 λ 演算的类型化版本，
它是众多类型化 λ 演算中最简单的一个。

它只包含一个类型构造器（type constructor）$\to$，
即，接受两个类型 $T1,T2$作为参数，返回一个函数类型 $T1\to T2$。

下文中，我们首先从最基础的概念说起，
详细的区分项（term）和值（value）的概念，
然后介绍简单类型化 λ 演算系统的求值规则和类型规则。

1. 项和值

1.1 项

项（term）是一个语法概念，一个合法的项，指的是一段符合语法的字符串。例如，
在 $\lambda\beta$系统中，项的定义如下，
$t::=x|\lambda x.t|t\ t$

它表明，一个合法的 $\lambda\beta$项，要么是一个变量 x，
要么是一个 λ 抽象（abstraction）$\lambda x.t$，
要么是一个 λ 应用（application）$t\ t$。

1.2 值

值（value）是一个和语义相关的概念，有三种常用的方法为项指定语义。
（1）操作语义，通过定义一个简单的抽象机器，来说明一个程序语言的行为。
（2）指称语义，一个项的语义是一个数学对象。
（3）公理语义，不是首先定义程序的行为，而是用项所满足的规则限定它的语义。

下面我们采用操作语义的方法，来定义求值的概念。
首先，我们人为指定项的一个子集，将其中的元素称为值。

假如项的定义如下，$t::=true|false|if\ t\ then\ t\ else\ t$，
我们可以定义值，$v::=true|false$。
值可能是项被求值的最终结果，但也不全是，因为对某些项的求值过程可能不会终止。

1.3 求值规则

求值规则，是定义在项上的推导规则，例如，
（1）$if\ true\ then\ t1\ else\ t2 \to t1$，
（2）$if\ false\ then\ t1\ else\ t2 \to t2$，
（3）$if\ t1\ then\ t2\ else\ t3 \to if\ t1'\ then\ t2\ else\ t3 t1 \to t1'$
其中，$x \to y$ 表示，项 x 可以一步求值为项 y。

1.4 范式

一个不含自由变量的项，称为封闭项，封闭项也称为组合子。
例如，恒等函数 $id=\lambda x.x$ 就是一个封闭项。

如果没有求值规则可用于项 t，就称该项是一个范式。
范式可能是一个值，也可能不是，但每一个值都应该是范式。

如果一个封闭项是一个范式，但不是一个值，就称该项受阻。
不是值的范式，在运行时间错误分析中起着极其重要的作用。

2. 类型

2.1 类型上下文

一个类型上下文（也称类型环境）Γ，是一个变量和类型之间绑定关系的集合。
空上下文，可以记为$\varnothing$，但是我们经常省略它。

用逗号可以在 Γ 右边加入一个新的绑定，例如，$\Gamma,x:T$。
$\vdash t:T$，表示项 t 在空的类型上下文中，有类型 T。

2.1 类型规则

$\lambda\rightarrow$是一个新的系统，比起$\lambda\beta$而言，
增加了一些基于类型的推导规则。

其中，$\lambda\rightarrow$中λ项的语法如下：
（1）$t::=x|\lambda x:T.t|t\ t$
（2）$T::=T\rightarrow T$
（3）$\Gamma::=\varnothing|\Gamma,x:T$

推导规则：

$$（1）\Gamma\vdash x:T x:T\in\Gamma$$

$$（2）\Gamma\vdash\lambda x:T1.t:T1\rightarrow T2\Gamma,x:T1\vdash t:T2$$

$$（3）\Gamma\vdash t1\ t2:T2\Gamma\vdash t1:T1\rightarrow T2\quad\Gamma\vdash t2:T1$$

根据以上的推导规则，我们可以证明，
$\vdash(\lambda x:Bool.x)\ true:Bool$

2.3 求值规则

$\lambda\rightarrow$系统中，值的定义如下：
（1）$v::=\lambda x:T.t$

求值规则，定义如下：

$$（1）t1\ t2\rightarrow t1'\ t2 t1\rightarrow t1'$$

$$（2）t1\ t2\rightarrow t1\ t2' t2\rightarrow t2'$$

（3）$(\lambda x:T.t)\ v\rightarrow[x\mapsto v]t$

其中（2），相当于β变换，
$[x\mapsto v]t$，表示将t中所有自由出现的x换为v。

2.4 Curry-style and Church-style

对于$\lambda\rightarrow$系统来说，通常有两种不同风格的解释方式，
如果我们首先定义项，然后定义项的求值规则——语义，

最后再定义一个类型系统，用以排除掉我们不需要的项，
这种语义先于类型的定义方式，称为 Curry-style。

另一方面，如果我们定义项，然后再给出良类型的定义，
最后再给出这些良类型项的语义，就称为 Church-style，类型先于语义，
在 Church-style 的系统中，我们不关心不良类型项的语义。

历史上，隐式类型的 λ 演算系统，通常是 Curry-style 的，
而显式类型的 λ 演算系统，通常是 Church-style 的。

3. 关于单位类型

简单类型化 λ 演算，直接用起来可能并不好用，
人们会再为它扩充一些类型，例如，添加一些基本类型 $Bool$，Nat 或者 $String$，
定义单位类型，列表类型，元组类型，和类型，等等。

下面我们选择单位类型进行介绍。

满足单位类型的项只有一个，为此我们新增一个项的定义，
$t ::= \dots | unit$

再新增一个类型的定义，
$T ::= Unit$

以及一个推导规则，
$\Gamma \vdash unit : Unit$

$Unit$ 的作用类似于 C 和 Java 中的 $void$ 类型，主要用于表示副作用，
在这样的语言中，我们往往并不关心表达式的结果，而只关心它的副作用，
因此，用 $Unit$ 来表示结果的类型，是一个合适的选择。

这里提到单位类型，是为以后 Top 类型和 Bot 类型做铺垫。

参考

Wikipedia: Simply typed lambda calculus

七、Recursive type

上文我们介绍了简单类型化 λ 演算系统 $\lambda \to$，并给它扩充了单位类型，

本文继续为它添加新的特性。

我们将提到 `let` 和 `letrec` 表达式，函数的不动点，
代数数据类型，以及递归类型。

最后我们发现，无类型系统实际上是具有，唯一递归类型的一种情形。

let 绑定

当写一个复杂表达式的时候，为了避免重复和增加可读性，
通常我们会给某些子表达式命名，
其中一个常用办法是，使用 `let` 表达式。

为此，我们要对简单类型化 λ 演算系统 $\lambda \to$ 进行扩展，
添加 `let` 表达式的语法项，求值规则以及类型规则。

新的语法：
$t ::= \dots | let\ x:T=t\ in\ t$

新的求值规则：
$let\ x:T=v\ in\ t \to [x \mapsto v]t$

$$\frac{t1 \to t1'}{let\ x:T1=t1\ in\ t2 \to let\ x:T1=t1'\ in\ t2}$$

新的类型规则：

$$\frac{\Gamma \vdash t1:T1 \quad \Gamma, x:T1 \vdash t2:T2}{\Gamma \vdash let\ x:T1=t1\ in\ t2:T2}$$

这样我们就可以写出 `let` 表达式了，
$let\ x:T1=t1\ in\ t2$，
它表示，求值表达式 $t1$，然后将其绑定到 $t2$ 中自由出现的 x 上面，
即在当前这种情况下（顾及 let polymorphism），`let` 可以表示为，$(\lambda x:T1.t2)t1$。

不动点

以上 `let` 表达式，$let\ x:T1=t1\ in\ t2$，
对 $t1$ 求值的时候有一个限制，那就是 $t1$ 中不能出现 x，
否则就像方程一样，x 出现在了等式的两边，$x=t1(x)$，
此时，$t2$ 中自由出现的 x，将是这个方程的解。

不过，通常而言，$t1$ 中是可以出现 x 的，
这时候我们就需要使用 `letrec` 进行绑定了。

为了看清 `letrec` 的真面目，我们用一个求阶乘的例子来说明问题，
$letrec\ f:nat\to nat=\lambda y:nat.(if\ Eq?\ y\ 0\ then\ 1\ else\ y*f(y-1))\ in\ f\ 5$
其中，nat 表示整数类型。

为了求解 $f=\lambda y:nat.(if\ Eq?\ y\ 0\ then\ 1\ else\ y*f(y-1))$，
我们定义一个新函数 F，使得，
$F=\lambda f:nat\to nat.\lambda y:nat.(if\ Eq?\ y\ 0\ then\ 1\ else\ y*f(y-1))$，
注意到 $f=F(f)$，所以 f 是 F 的不动点。

这里我们暂且不讨论不动点的存在性和唯一性问题，
只是引入一个不动点算子，$fix\sigma:(\sigma\to\sigma)\to\sigma$，
它可以用来计算任意函数 $F:\sigma\to\sigma$ 的不动点。

为了达到这个目的，$fix\sigma$ 必须满足以下约束条件，
$fix\sigma=\lambda f:\sigma\to\sigma.f(fix\sigma f)$。

引入了 $fix\sigma$ 之后，`letrec` 就可以用 `let` 表示出来了，
$letrec\ f:\sigma=t1\ in\ t2\Leftrightarrow let\ f:\sigma=(fix\sigma\lambda f:\sigma.t1)\ in\ t2$

代数数据类型

在某些编程语言中，可以自定义递归类型，例如在 Haskell 中，

```
data List a = Nil | Cons a (List a)
lst :: List Int
lst = Cons 1 $ Cons 2 $ Nil
```

以上定义采用了递归的方式，定义了一个代数数据类型`List`，
所谓代数数据类型，是由基本类型经过复合运算，得来的类型。

Haskell中，使用|表示和类型（sum type），
而带参数值构造器（value constructor）`Cons`，用于表示各参数（`a`，`List a`）类型的积类型（product type），
无参构造器`Nil`，用来表示单位乘积类型（empty product）。
（关于函数类型与指数的关系，以后有机会再介绍。）

和`letrec`中的场景相似的是，`List`也出现在了等式的两边，
于是，我们定义$\mu t.\sigma$，表示满足等式$t=\sigma$的最小类型，
其中，t和σ是类型，且t通常会在σ中出现。

因此，以上递归定义的`List`类型可以表示为，
$\mu\phi.\lambda\alpha.(1+\alpha\times(\phi\ \alpha))$

其中，α表示类型参数`a`，
ϕ是一个函数，用于表示类型构造器（data constructor）`List`，
μ算子用来计算类型的不动点。

无类型λ演算

在无类型λ演算系统$\lambda\beta$中，
定义递归类型，$\mu t.t\to t$，它满足类型等式$t=t\to t$。

这样的话，无类型λ演算系统$\lambda\beta$，
就可以无缝的迁移到一个类型化的系统中去了，
该系统只存在一个类型，即递归类型$\mu t.t\to t$，
所有的项，都具有这个类型。

因此，对于支持递归类型的系统而言，
无类型相当于具有唯一类型（Untyped means uni-typed）。

总结

本文介绍了递归类型，引入了两个算子fix_σ和μ，分别用来求解函数和类型的不动点。

但是，待求的不动点是否存在，是否唯一，我们仍不能确定。
要想证明这件事情并不简单，需要补充很多额外的数学知识，例如，良基归纳法和最小不动点定理，
此外，不动点的存在性，和递归的终止性也有关系。

好在我们所遇到的大多数场景，都是满足这些要求的，
因为，我们总是先确信这个解是存在的，然后再去编程，例如，

```
fact :: Int -> Int
fact 1 = 1
fact n = n * fact (n - 1)
```

将`fact`写到等式两边，能让我们更方便对`fact`进行定义。

参考

Algebraic data type
Empty product
Foundations for programmming languages
Practical Foundations for Programming Languages

八、Subtype

对象和类

子类型几乎是面向对象编程语言所特有的，
在面向对象的编程语言中，计算是由对象和对象之间消息传递来完成的，
对象（object）通常包含两个组成部分，数据（data）和代码（method）。

其中数据（data）一般是可变的，由每个对象所专有，
通常称之为对象的状态（state）。
代码（method）通常是不可变的。

大部分面向对象的编程语言是基于类（class）的，但类（class）却并不是必须的，
一个支持词法作用域的编程语言中，
闭包（closure）就是包含内部状态的对象（object）。

```
; let over lambda
(define counter
      (let ((n 0))
            (lambda () (set! n (+ n 1)) n)))
(counter)    ; 1
(counter)    ; 2
```

类（class）实际上可以看做一个工厂函数，用来生成对象。

```

; lambda over let over lambda
(define create-counter
      (lambda ()
            (let ((n 0))
                  (lambda () (set! n (+ n 1)) n))))
(define counter
      (create-counter))
(counter)    ; 1
(counter)    ; 2

```

类型和类

类型（type）是与类（class）不同的概念。
对象所属的类（class）是它的工厂函数，
而对象的类型（type），是它在形式系统中，所具有的逻辑性质（logical property）。

子类（subclass），通过编写与父类之间的差别，创建一个新类，目的是代码复用。

> A subclass is a differential description of a class.

包含子类（subclass）之后，众多类（class）之间，
构成一个偏序关系（partial order）。

而引入子类型（subtype）是为了放宽类型系统的约束条件。
例如，$(\lambda r:\{x:Nat\}.\ r.x)\ \{x=0,y=1\}$
其中，r 的类型为 $\{x:Nat\}$，表示记录类型（record）。

$\{x=0,y=1\}$ 的类型为 $\{x:Nat,y:Nat\}$，与 r 的类型不同，
根据推导规则，$\Gamma\vdash t1\ t2:T2\Gamma\vdash t1:T1\rightarrow T2\quad \Gamma\vdash t2:T1$，
$(\lambda r:\{x:Nat\}.\ r.x)\ \{x=0,y=1\}$ 将无法通过类型检查。

可是，函数中确实只用到了 $r.x$，多传一个 y 理应总是安全的。
因此，不带子类型的简单类型化 λ 演算，它的推导规则就显得过于严谨了。

我们可以引入记录类型（record）之间的子类型关系，记为，
$\{x:Nat,y:Nat\}<:\{x:Nat\}$
用于表示类型 $\{x:Nat,y:Nat\}$ 是 $\{x:Nat\}$ 的子类型。
对于记录类型来说，这里可能有些奇怪，因为更“小”的类型却包含更多的字段。

一般的，$S<:T$ 表示 S 为 T 的子类型，
如果在某个上下文中，期待一个 T 类型的项，那么 S 在这个上下文中也是合法的，
即，$\Gamma\vdash t:T\Gamma\vdash t:S\quad S<:T$
该推导规则，通常称之为安全替换原则。

为了能够安全替换，子类型应该具有自反性：$S<:S$，
还应该具有传递性，$S<:TS<:U\quad U<:T$。

Top 类型与 Bottom 类型

（1）Top 类型

理解了子类型之后，我们就可以引入一个新的类型常量 Top，称为 Top 类型。
所有其它类型都是它的子类型，$S<:Top$。

因为具有自反性和传递性，子类型之间构成了一个前序关系（preorder），
由于记录类型中的字段，顺序是可以置换的，
$\{x:Nat,y:Nat\}$和$\{y:Nat,x:Nat\}$分别为另一个的子类型，
因此，子类型关系不是一个偏序关系（partial order）。

（2）Bottom 类型

除了 Top 类型之外，我们很自然的会问，
是否存在一个类型，它是所有其他类型的子类型，
为此，我们需要对类型系统再扩展，引入 Bot 类型常量，称为 Bottom 类型，
满足 $Bot<:T$。

Bot 类型中是不能有闭值的，否则，假设 v 是 Bot 类型的一个值，
则根据安全替换原则有 $v:Top\to Top$，表明 v 是一个函数，
此外还有 $v:\{\}$，表明 v 是一个记录，
但是 v 作为一个值，不可能既是函数又是记录，矛盾。

我们在第六篇中提到过，所谓封闭，指的是不含自由变量。
所谓值，就是事先约定好的项的子集。
值都是范式，没有求值规则可被继续使用，是对项求值的最终结果。

Bot 类型中虽然不能有闭值，
但是却可以包含受阻项，即事先约定好的不是值的范式。

例如，我们可以指定 $error$ 是一个受阻项（不给它指定求值规则），
再指定它为 Bottom 类型，$error:Bot$。
这样 $error$ 就可以在不同的上下文中，被提升为不同的类型了。
$\lambda x:T.$

if...*then*
  result
else
  error
以上 λ 项是良类型的（well typed），无论 *result* 是何种类型。

关于 Top 类型和 Bottom 类型，我们最后再看一个例子，
TypeScript 中的 any 类型，是一个 Top 类型，
而 never 类型，是一个 Bottom 类型。

总结

本文为简单类型化 λ 演算添加了子类型，
并且对比了类（class）与类型（type）这两个概念。

lambda calculus 是函数式编程语言的计算模型，
前几篇，我们保持基本的演算系统（求值规则）不变，
给它添加了不同的类型推导规则，得到了不同的类型系统。

object calculus 是面向对象语言的计算模型，
在它之上，我们同样可以添加相似的类型系统。

因此，类型系统是与演算（calculus）独立的概念。
这印证了我们之前在第一篇中提到的一句话，
类型系统，可以看做是附着在语言语法之上的一套符号证明系统。

参考

Let Over Lambda
Types and programming languages
A Theory of Objects

九、Let polymorphism

类型变量

到目前为止，我们遇到的每一个 λ 项都有唯一确定的类型，
因为，项的类型都被显式的注释在了它的后面。
例如，我们可以定义一个恒等函数 $id=\lambda x{:}Nat.\ x{:}Nat\to Nat$，
则 id 的类型就是固定的，$Nat\to Nat$，而 $id true$ 就不是良类型的。

为每一个类型的恒等函数都定义各自的版本，是非常繁琐的，
因此，一个自然的想法是，我们能否让 id 的类型参数化，
让它在不同的上下文中，实例化为不同的具体类型。
例如，$id=\lambda x{:}X.\ x{:}X\to X$，其中 X 是类型参量。

类型代换

类型代换 σ，指的是一个从类型变量到类型的有限映射。
例如，$\sigma=[X\mapsto T,Y\mapsto U]$，会将类型变量 X,Y 分别代换为 T,U。
其中，X,Y 称为代换 σ 的定义域，记为 $dom(\sigma)$，
而 T,U 称为代换 σ 的值域，记为 $range(\sigma)$。

值得一提的是，所有的代换都是同时进行的，$\sigma=[X\mapsto Bool,Y\mapsto X\to X]$，
是将 X 映射成 $Bool$，将 Y 映射成 $X\to X$，而不是 $Bool\to Bool$。

代换可以用下面的方式来定义，
（1）$\sigma(X)=X$，如果 $X\notin dom(\sigma)$
（2）$\sigma(X)=T$，如果$(X\mapsto T)\in\sigma$
（3）$\sigma(Nat)=Nat$，$\sigma(Bool)=Bool$
（4）$\sigma(T1\to T2)=\sigma T1\to\sigma T2$

对于类型上下文 $\Gamma=\{x1{:}T1,\cdots,xn{:}Tn\}$来说，
$\sigma\Gamma=\{x1{:}\sigma T1,\cdots,xn{:}\sigma Tn\}$

类型代换的一个重要特性是它保留了类型声明的有效性，
如果包含类型变量的项是良类型的，那么它的所有代换实例也都是良类型的。

类型推断

在类型上下文 Γ 中，对于包含类型变量的项 t，我们通常会提出两个问题，

（1）它的所有代换实例，是否都是良类型的？
即，是否 $\forall\sigma\exists T,\sigma\Gamma\vdash\sigma t:T$。

（2）是否存在良类型的代换实例？
即，是否 $\exists\sigma\exists T,\sigma\Gamma\vdash\sigma t:T$。

对于第一个问题，将引出参数化多态（parametric polymorphism），
例如，$\lambda f:X\to X.\lambda a:X.f(f(a))$，它的类型为$(X\to X)\to X\to X$，
无论用什么具体类型 T 来代换 X，代换实例都是良类型的。

对于第二个问题，原始的项可能不是良类型的，
但是可以选择合适的类型代换使之实例化为良类型的项。

例如，$\lambda f:Y.\lambda a:X.f(f(a))$，是不可类型化的，
但是如果用 $Nat\to Nat$ 代换 Y，用 Nat 代换 X，
$\sigma=[X\mapsto Nat,Y\mapsto Nat\to Nat]$，
就可以得到，$\lambda f:Nat\to Nat.\lambda a:Nat.f(f(a))$，
可类型化为$(Nat\to Nat)\to Nat\to Nat$。

或者，取 $\sigma'=[Y\mapsto X\to X]$，结果也能得到一个良类型的项，尽管仍包含变量。

在寻找类型变量有效实例的过程中，出现了类型推断（type inference）的概念。
意味着由编译器来帮助推断 λ 项的具体类型，
在 ML 语言中，程序员可以忽略所有的类型注释——隐式类型（implicit typing）。

在进行推断的时候，对每一个原始的 λ 抽象 $\lambda x.t$，
都用新的类型变量进行注释，写成 $\lambda x:X.t$，
然后采取特定的类型推导算法，找到使项通过类型检查的一个最一般化的解。

设 Γ 为类型上下文，t 为项，
(Γ,t)的解，是指这样的一个序对(σ,T)，使得 $\sigma\Gamma\vdash\sigma t:T$ 成立。

例如，设 $\Gamma=f:X,a:Y$，$t=f\ a$，则

$(\sigma=[X\mapsto Y\to Nat],Nat)$，$(\sigma=[X\mapsto Y\to Z],Z)$，
都是(Γ,t)的解。

基于约束的类型化

（1）约束集

在实际情况中，(Γ,t)的解，并不一定满足其他类型表达式的约束条件，
所以，我们寻找的是满足这些约束条件的特解。

所谓约束条件，实际上指的是约束集 C，
它由一些包含类型参量的项的等式构成，$\{Si=Ti|i\in l..n\}$。

如果一个代换 σ 的代换实例，σS 和 σT 相同，则称该代换合一（unify）了等式 $S=T$。
如果 σ 能合一 C 中的所有等式，则称 σ 能合一（unify）或满足（satisfy）C。

我们用 $\Gamma\vdash t:T|\chi C$，来表示约束集 C 满足时，项 t 在 Γ 下的类型为 T，
其中 χ 为约束集中，所有类型变量的集合，有时为了讨论方便可以省略它。

例如，对于项 $t=\lambda x:X\to Y.x\ 0$，
约束集可以写为$\{Nat\to Z=X\to Y\}$，则 t 类型为$(X\to Y)\to Z$。（算法略）
而代换 $\sigma=[X\mapsto Nat,Z\mapsto Bool,Y\mapsto Bool]$，
使得等式 $Nat\to Z=X\to Y$ 成立，
所以，我们推断出了$(Nat\to Bool)\to Bool$ 是项 t 的一个可能类型。

（2）约束集的解

约束集的解一般不是唯一的，所以一个关键问题是如何确定一个“最好”的解。

我们称代换 σ 比 σ' 更具一般性（more general），如果 $\sigma'=\gamma\circ\sigma$，记为 $\sigma\sqsubseteq\sigma'$，
其中，γ 为一个代换，$\gamma\circ\sigma$ 表示代换的复合，$(\gamma\circ\sigma)S=\gamma(\sigma S)$。

约束集 C 的主合一子（principal unifier）指的是代换 σ，
它能满足 C，且对于所有满足 C 的代换 σ'，都有 $\sigma\sqsubseteq\sigma'$。

如果(Γ,t,S,C)的解(σ,T)，对于任何其他解(σ',T')，都有 $\sigma\sqsubseteq\sigma'$，
则称(σ,T)是一个主解（principal solution），称 T 为 t 的主类型（principal type）。
可以证明，如果(Γ,t,S,C)有解，则它必有一个主解。

let 多态

多态（polymorphism）指的是单独一段程序能在不同的上下文中实例化为不同的类型。
其中 let 多态，是由 let 表达式引入的多态性。

（1）单态性

假设我们定义了一个 *double* 函数，它能将一个函数对参数应用两次，
$let\ double=\lambda f{:}Nat\to Nat.\lambda a{:}Nat.f(f(a))\ in$
　$double\ (\lambda x{:}Nat.succ\ x)\ 1$
此时，*double* 的类型为$(Nat\to Nat)\to Nat\to Nat$。

如果我们想将 *double* 应用于其他类型，就必须重写一个新的 *double′*，
$let\ double'=\lambda f{:}Bool\to Bool.\lambda a{:}Bool.f(f(a))\ in$
　$double'\ (\lambda x{:}Bool.x)\ true$
此时 *double′*的类型为$(Bool\to Bool)\to Bool\to Bool$。

我们不能让一个 *double* 函数，既能用于 *Nat* 类型，又能用于 *Bool* 类型。
即使在 *double* 中用类型变量也没有用，
$let\ double=\lambda f{:}X\to X.\lambda a{:}X.f(f(a))\ in\ \cdots$

例如，如果写，
$let\ double=\lambda f{:}X\to X.\lambda a{:}X.f(f(a))\ in$
　$let\ a=double\ (\lambda x{:}Nat.succ\ x)\ 1\ in$
　　$let\ b=double\ (\lambda x{:}Bool.x)\ true\ in\ \cdots$
则在 a 定义中使用 *double*，会产生一个约束 $X\to X=Nat\to Nat$，
而在 b 定义中使用 *double*，则会产生约束 $X\to X=Bool\to Bool$，
这样会使类型变量 X 的求解发生矛盾，导致整个程序不可类型化。

（2）多态性

let 多态所做的事情，就是打破这个限制，
让类型参量 X 在上述不同的上下文中，可以分别实例化为 Nat 和 $Bool$。

这需要改变与 let 表达式相关的类型推导规则，在第七篇中，我们提到过，

$$\Gamma \vdash let\ x:T1=t1\ in\ t2:T2 \Gamma \vdash t1:T1 \quad \Gamma,x:T1 \vdash t2:T2$$

它会首先计算 $T1$作为 x 的类型，然后再用 x 来确定 $T2$的类型。
此时，let 表达式 $let\ x=t1:T1\ in\ t2$，可以看做$(\lambda x:T1.t2)t1$的简写。

为了引入多态性，我们需要对上述类型推导规则进行修改，

$$\Gamma \vdash let\ x=t1\ in\ t2:T2 \Gamma \vdash [x \mapsto t1]t2:T2$$

它表示，先将 $t2$中的 x 用 $t1$代换掉，然后再确定 $t2$的类型。

这样的话，
$let\ double=\lambda f:X \to X.\lambda a:X.f(f(a))\ in$
$\quad let\ a=double\ (\lambda x:Nat.succ\ x)\ 1\ in$
$\qquad let\ b=double\ (\lambda x:Bool.x)\ true\ in\ \cdots$

就相当于，
$let\ a=\lambda f:X \to X.\lambda a:X.f(f(a))\ (\lambda x:Nat.succ\ x)\ 1\ in$
$\quad let\ b=\lambda f:Y \to Y.\lambda a:Y.f(f(a))\ (\lambda x:Bool.x)\ true\ in\ \cdots$
通过 let 多态，产生了 $double$ 的两个副本，并为之分配了不同的类型参量。

此时，let 表达式 $let\ x=t1\ in\ t2$，可以看做$[x \mapsto t1]t2$的简写。

参考

Hindley–Milner type system
Types and programming languages
Haskell 2010 Language Report

十、Parametric polymorphism

回顾

上文我们介绍了 let 多态，
将 let 表达式 $let\ x=t1\ in\ t2$，看做了$[x\mapsto t1]t2$的简写，
即，把 $t2$中出现的所有 x，都用 $t1$替换掉，因此这些副本可以具有不同的类型。

本文将介绍另外一种多态形式，称为参数化多态（parametric polymorphism），例如，

```
1data Maybe a = Nothing | Just a
```

以上 Haskell 代码，定义了一个`Maybe a`类型，
其中`Maybe`称为类型构造器（type constructor），`a`是它的参数。

`Maybe`不是一个合法的类型，它只有和某个具体的`a`放在一起，才是一个合法的类型，
例如，`Maybe Int`，`Maybe Char`。

System F

为了实现参数化多态，我们需要对简单类型化 λ 演算（$\lambda\to$系统）进行扩展，
在 $\lambda\to$中，我们用 $\lambda x.t$ 来表示 λ 抽象（lambda abstraction），
而使用 $t1\ t2$来表示 λ 应用（lambda application）。

现在我们引入一种新的抽象形式，$\lambda X.t$，它的参数 X 是一个类型，称为类型抽象，
再引入一种新的应用形式，$t[T]$，称为类型实例化，其中 T 是一个类型表达式。

求值规则如下，
$(\lambda X.t)[T]\to[X\mapsto T]t$

例如，我们可以这样定义一个多态函数，$id=\lambda X.\lambda x:X.x$
当把它应用于类型 Nat 时，
$id[Nat]\to[X\mapsto Nat](\lambda x:X.x)=\lambda x:Nat.x$，它为 Nat 类型上的恒等函数。
而把它应用于类型 $Bool$ 时，
$id[Bool]\to[X\mapsto Bool](\lambda x:X.x)=\lambda x:Bool.x$，它为 $Bool$ 类型上的恒等函数，

可见，id 的具体类型，依赖于它的类型参数。
它应用于任意一个类型 T，$id[T]$都会得到一个类型为 $T\to T$ 的函数，
因此，人们通常将 id 的类型记为$\forall X.X\to X$。

类型规则如下，

$$（1）\Gamma\vdash\lambda X.t:\forall X.T\Gamma,X\vdash t:T$$

$$（2）\Gamma\vdash t[T2]:[X\mapsto T2]T1\Gamma\vdash t1:\forall X.T1$$

其中，类型$\forall X.T$，叫做全称类型（universal type），
∀称为全称量词（universal quantifier），
引入了全称类型之后得到的系统，称为 System F。

Rank-N Types

有了全称类型之后，函数的参数类型和返回值类型，都有可能具有全称类型。
不难看出，函数返回值类型的全称量词，总是可以提取出来，放到最外面，
但是参数类型的全称量词，不能提取出来。

例如，$\forall X.X\to(\forall Y.Y\to X)$，相当于$\forall X.\forall Y.X\to Y\to X$，
而$(\forall X.X\to X)\to Nat$，与$\forall X.(X\to X)\to Nat$ 则不同。

不包含全称量词的类型表达式，具有 rank-0 类型，也称为单态类型（monotype），
全称量词都可以提取出来类型表达式，具有 rank-1 类型（rank-1 type），
一个函数类型，它的入参具有 rank-n 类型，那么该函数就具有 rank-(n+1)类型。

例如，
$((\forall X.X\to X)\to Nat)\to Bool\to Bool$，具有 rank-3 类型，
$Nat\to(\forall X.X\to X)$，具有 rank-1 类型。

System F 的功能是很强大的，但是不幸的是，
人们发现，该系统中的类型推导算法是不可判定的。
例如，一般而言，一个 rank-3 及其以上 rank-N 类型的表达式，其类型是不可确定的，
为了确定它的类型，人们不得不手工加上必要的类型信息。

Haskell 采用了 Hindley–Milner 类型系统，
它是 System F 的一个子集，其中包含了可判定的类型推导算法。
在 Haskell 中，类型参量（type variable）默认具有全称类型（universally quantified），
例如，`a -> a`，实际上表示类型$\forall a.a\to a$，
`a -> a`可看做`(->) a a`类型，其中`->`为函数类型构造器。

非直谓性

在数学和逻辑学中，一个定义称为非直谓的（impredicative），
指的是它包含了自引用（self-reference）。
例如，在定义一个集合的时候，用到了正在定义的这个集合。

罗素悖论就是用非直谓的方式构造出来的，
如果我们定义 $R=\{x|x\notin x\}$，那么 $R\in R\Leftrightarrow R\notin R$。
非直谓定义并不一定导致矛盾，有些情况下还是有用的，
例如，我们可以非直谓的定义，集合中的最小元素为，$Amin=x$，$\forall y.x\leqslant y$。

具有参数化多态的类型表达式，
也有直谓（predicative）和非直谓（impredicative）之分。

如果它可以用一个多态类型实例化，例如用它自己来实例化，
就称为非直谓多态类型（impredicative polymorphism）。

反之，如果一个多态类型表达式，只能用单态类型实例化，
就称它具有直谓多态类型（predicative_polymorphism）。

单态限制

```
f x = let g y z = ([x,y], z)
          in ...
```

假设 `x` 的类型为 `a`，那么 `g` 的类型只能为 `a -> b -> ([a], b)`，
其中，由于列表类型的限制，`x` 和 `y` 必须具有相同的类型。

此时，只有 `b` 可以具有全称量词，即 $\forall b.a\to b\to([a],b)$，
因为 `a` 在类型上下文中，已经出现了，不能再被实例化为其他的类型了。
我们称，`g` 的第一个参数 `a` 具有单态性（monomorphism）。

例如，`(g True, g 'c')`不是良类型的，
而`(g True, g False)`是良类型的。

值得一提的是，显式的给 `g` 注明类型，也不能阻止 `a` 的单态行为，

```
f x = let
```

```
            g :: a -> b -> ([a],b)
            g y z = ([x,y], z)
        in ...
```
此时，`a`仍然是单态的。

在 Hindley-Milner 类型系统中，
如果一个类型变量，不在类型上下文中出现，它就可以被全称化（generalize）。
但是 Haskell 考虑到性能和模块间的类型推导，
还增加了特殊的单态限制（monomorphism restriction）避免全称化。

Rule 1

在一组相互依赖的声明中，满足以下两个条件，其中的类型变量才会被全称化，
（1）每一个变量，都被函数或模式匹配所绑定，
（2）被模式匹配绑定的变量，都有显式的类型签名。

Rule 2

导入到其他模块（module）的单态类型变量，被认为是有歧义的（ambiguous），
类型通过其来源模块内的`default`声明来决定。

```
module M1(len1) where
   default( Int, Double )
   len1 = genericLength "Hello"
module M2 where
   import M1(len1)
   len2 = (2 * len1) :: Rational

```
当模块`M1`的类型推导结束后，根据 Rule 1，`len1`具有单态类型，`len1 :: Num a => a`，
Rule 2 表明，类型变量`a`具有歧义性，必须使用`default`声明来解决歧义。
因此，根据`default( Int, Double )`，`len1`得到了类型`Int`，
不过，`M2`中对`len1 :: Int`的使用导致了类型错误。

参考

Types and programming languages

Haskell 2010 Language Report
Glasgow Haskell Compiler User's Guide
Parametric polymorphism
Practical type inference for arbitrary-rank types
System F
Hindley–Milner type system