类型重构

前面若干章不断充实着我们的编程语言，同时，也在无形之中让语法趋向繁杂。其中，最「无聊」的语法就是给参数标注类型了。比如编写一个极其简单的递归函数

```
let g = λf:Int→Bool. λn:Int.
          if n=0 then False
          else if n=1 then True
          else f (n-2)
in fix g
```

我们被迫不厌其烦地注明 f 与 n 的类型，以致程序凌乱、思路不畅。有没有可能让机器自动标注类型呢？此即「类型重构」的出发点。

先从例子中得到一点启发。设程序

```
let g = λf. λn.
          if n=0 then False
          else if n=1 then True
          else f (n-2)
in fix g
```

怎么推理出其类型呢？

首先，既然 f 和 n 的类型未知，而我们又必须使用其类型，那么不妨就用类型变量 F 和 N 代表之：

```
let g = λf:F. λn:N.
          if n=0 then False
          else if n=1 then True
          else f (n-2)
in fix g
```

接下来，让我们摆正自己的位置：作为一台「类型重构机器」，我们得做

个老好人，千方百计地让类型检查通过。所以，不到迫不得已，我们决计不让错误发生。以下推理即遵循该原则。

1° 留意到比较句「n=0」，若想让程序通过类型检查，则迫不得已地要求 N=Int。「n=1」同理。

2° 留意到函数调用「f (n-2)」。若想让程序通过检查，则迫不得已地要求 F=Int ▸ y，其中 y 是一个新引入的类型变量，取值完全自由。

3° 又为了让 if-then-else 通过检查，我们得要求

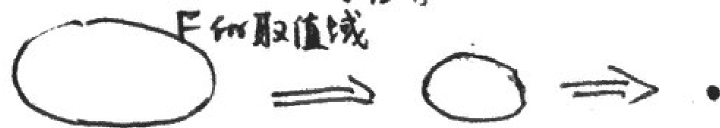
(1) if 从句类型为 Bool。还好，「n=0」的类型本就必为 Bool。（「n=1」）

(2) else 从句与 then 从句的类型相同。因而 f (n-2) 的类型 y=Bool.

4° 综合而言，g 的类型为
(Int ▸ Bool) ▸ (Int ▸ Bool)

5° 最后，fix g 的类型为 Int ▸ Bool，故整个程序亦然。

上述例子给我们的启发：一开始，对参数类型全然无知，故直接安插上一些完全自由的「类型变量」—— 它们相互独立，取值随意。接下去，我们深入分析函数内部，从小处开始着眼，逐步添加必要的约束，细化类型变量的取值范围。所谓「必要」，即迫不得已，即如不添加该约束则无法通过类型检查。如若有些约束相互矛盾，则意味着这程序「无可救药」了，纵使我们再宽容也是白搭。



为了数学处理方便，我们把类型重构分拆成两轮来完成：

(1) 引入类型变量，并一气呵成地找出全体约束条件，而不顾这些约束是否一致。

(2) 求解这组约束。若不可解，则报告类型错误。

约定：以下讨论的语言仅包含 Int 与 Bool 两种基本类型，而不考虑诸如元组、记录、指针之类的复杂类型。

def 类型推导与约束生成

判断形式：$\Gamma \vdash t:T \mid Q \to Q' \mid C$

含义：在假设集 Γ 下，t 具有类型 T（T 可能包含类型变量）；并且为了达成这目标，必须满足约束集 C。Q 与 Q' 是队列，用以提供完全新鲜的类型变量，其中 Q 是初始态，Q' 是终态。

规则：

$$\frac{}{\Gamma \vdash \mathrm{True}:\mathrm{Bool} \mid Q \to Q \mid \emptyset}\;[\text{T-TRUE}]$$

$$\frac{}{\Gamma \vdash \mathrm{False}:\mathrm{Bool} \mid Q \to Q \mid \emptyset}\;[\text{T-FALSE}]$$

$$\frac{}{\Gamma \vdash i:\mathrm{Int} \mid Q \to Q \mid \emptyset}\;[\text{T-INT}]$$

$$\frac{Q = X, Q' \qquad \Gamma \cup \{x:X\} \vdash t:T \mid Q' \to Q'' \mid C}{\Gamma \vdash (\lambda x.t): X \blacktriangleright T \mid Q \to Q'' \mid C}\;[\text{T-FUN}]$$

$$\frac{\Gamma \vdash t_1:T_1 \mid Q \to Q' \mid C_1 \qquad \Gamma \cup \{x:T_1\} \vdash t_2:T_2 \mid Q' \to Q'' \mid C_2}{\Gamma \vdash (\mathrm{let}\ x=t_1\ \mathrm{in}\ t_2): T_2 \mid Q \to Q'' \mid C_1 \cup C_2}\;[\text{T-LET}]$$

$$\frac{\Gamma \vdash t:T \mid Q \to Q' \mid C \qquad Q' = X, Q''}{\Gamma \vdash (\mathrm{fix}\ t): X \mid Q \to Q'' \mid C \cup \{T = X \blacktriangleright X\}}\;[\text{T-REC}]$$

$$\frac{\Gamma \vdash t_1:T_1 \mid Q \to Q' \mid C_1 \qquad \Gamma \vdash t_2:T_2 \mid Q' \to Q'' \mid C_2 \qquad Q'' = X, Q'''}{\Gamma \vdash (t_1\ t_2): X \mid Q \to Q''' \mid C_1 \cup C_2 \cup \{T_1 = T_2 \blacktriangleright X\}}\;[\text{T-App}]$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1:T_1 \mid Q \to Q' \mid C_1 \\ \Gamma \vdash t_2:T_2 \mid Q' \to Q'' \mid C_2 \\ \Gamma \vdash t_3:T_3 \mid Q'' \to Q''' \mid C_3\end{array}}{\Gamma \vdash (\mathrm{if}\ t_1\ \mathrm{then}\ t_2\ \mathrm{else}\ t_3): T_2 \mid Q \to Q''' \mid C_1 \cup C_2 \cup C_3 \cup \left\{\begin{array}{l}T_1 = \mathrm{Bool} \\ T_2 = T_3\end{array}\right\}}\;[\text{T-IF}]$$

$$\frac{\Gamma \vdash t_1:T_1 \mid Q \to Q' \mid C_1 \qquad \Gamma \vdash t_2:T_2 \mid Q' \to Q'' \mid C_2}{\Gamma \vdash (t_1\ +/-/*/\div\ t_2): \mathrm{Int} \mid Q \to Q'' \mid C_1 \cup C_2 \cup \{T_1 = \mathrm{Int}, T_2 = \mathrm{Int}\}}\;[\text{T-ARITH}]$$

我们挑有代表性的规则进行说明：

[T-TRUE] 无论在何种Γ下，True的类型都无条件为Bool，故约束集为∅。另外，既然没有使用新的类型变量，故Q以原样奉还。

[T-FUN] 面对$(\lambda x.t)$，我们暂时不知x的类型，因而从队列Q中抽取一个类型变量X给x贴上，并以此为假设作进一步推导。在子过程中，队列可能改变为Q''，且可能产生若干假设C。所有这些，都原样继承下来。

[T-APP] 分别推导t_1与t_2的类型，得T_1与T_2，以及相应的约束条件C_1与C_2。除了C_1、C_2必须满足，我们还得附加必要条件：「$T_1 = T_2 \blacktriangleright X$」，其中X是新从队列中抽取的自由类型变量。

remark. 请注意，虽然我们不断往约束集内新增约束方程，但是却从来没有验证其真假性。集合{Int=Bool}、{X=Bool, X=Int}、{X=Y▶Y, Y=X}之类的矛盾情形是绝对可能的。当前，我们关心的仅仅是一股脑儿地收集全体约束条件。

Problem
请自行补充 $t_1 < t_2$ / $t_1 \le t_2$ / $t_1 > t_2$ / $t_1 \ge t_2$ / $t_1 = t_2$ 的类型规则

def 第一轮分析（收集约束）的成果.
设t是一个程序，取定Q_0={全体类型变量}为包含有无穷多个类型变量的队列（比如，可取成$\{X_1, X_2, \ldots\}$）。
那么，我们称满足

$$\emptyset \vdash t : T \mid Q_0 \to Q' \mid C$$

的T、Q'、C为第一轮分析的成果。其中，T称为待定类型，C称为约束集。（Q'不是重点，会弃之）

e.g.1 $\lambda b.\lambda n.\lambda f.$

if $f(n)>0$ then b else False

经第一轮分析，得到待定类型为

$$\underbrace{X_1}_{b} \blacktriangleright [\underbrace{X_2}_{n} \blacktriangleright (\underbrace{X_3}_{f} \blacktriangleright X_1)]$$

约束集为 $\{X_3 = X_2 \blacktriangleright X_4,\ X_4 = \text{Int},\ \text{Bool} = \text{Bool},\ X_1 = \text{Bool}\}$

e.g.2 $\lambda f.\lambda x.\ f\ f\ x$

经第一轮分析，得到待定类型为

$$X_1 \blacktriangleright (X_2 \blacktriangleright X_4)$$

约束集为 $\{X_1 = X_1 \blacktriangleright X_3,\ X_3 = X_2 \blacktriangleright X_4\}$

e.g.3 $10 * 9 + ($if True then 2 else False$)$

经第一轮分析，得到待定类型 Int

约束集为 $\{\text{Int}=\text{Int},\ \text{Int}=\text{Int},\ \text{Bool}=\text{Bool},\ \text{Bool}=\text{Int},\ \text{Int}=\text{Int}\}$

约束集已收集好，接着就要求解了。不过，什么是「解」呢？

解，即未知量的具体化，使得方程左右两端相等。更抽象地说：解是一种从未知量到特殊量的映射，以使约束在其作用下被满足。在这儿，我们用「滤镜」一词来形象地称呼映射：

def 单滤镜.

设 X_0 是一个类型变量，而 T_0 是一个类型（既可为类型变量，亦可为实在的类型，还可为二者之拼合）. 定义

$$[X_0 \mapsto T_0] := \begin{cases} X_0 \mapsto T_0 \\ \text{非}X_0 \mapsto \text{本身} \end{cases}$$

称之为一个单重滤镜。

remark. 单重滤镜几乎就是恒等映射——除却在 X_0 一点上表现迥异。

e.g. $\sigma := [Y \mapsto Z \blacktriangleright \text{Bool}]$. 则

$\sigma(X) = X$，$\sigma(Z) = Z$，$\sigma(Y) = Z \blacktriangleright \text{Bool}$

def (组合)滤镜

设 $\sigma_1, \cdots, \sigma_n$ 均为单重滤镜。$(n \geq 0)$

$\sigma := \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_n$。称 σ 为组合滤镜。

e.g. $\sigma := [y \mapsto u \blacktriangleright v] \circ [x \mapsto y]$. 则

$\sigma(x) = \sigma(y) = u \blacktriangleright v, \quad \sigma(u) = u$

为了方便，我们有时会「滥用」滤镜的定义，将滤镜的作用域扩张到任何类型上。

比方说，我们允许上例中的 σ 作用于 $x \blacktriangleright (u \blacktriangleright y) \blacktriangleright \text{Bool}$ 上，得到 $(u \blacktriangleright v) \blacktriangleright (u \blacktriangleright (u \blacktriangleright v)) \blacktriangleright \text{Bool}$。一句话归结，即「有则改之，无则加勉」。严格的定义就不给出了。

def 约束集的解.

（这些等号代表人们的期望，未必真能实现）

设约束集 $C := \{T_1 = S_1, T_2 = S_2, \cdots, T_n = S_n\}$.

如果滤镜 σ 有 $\sigma(T_i) = \sigma(S_i) \quad \forall i = 1, \dots, n$,

（这是数学意义上的相等）

则称 σ 是 C 的解。记作 $\sigma \models C$.

在商讨如何求解以前，有必要暂缓一下，想想为什么约束-求解法是合理的。从直觉出发，我们在引入类型变元及添加约束时，总遵循着保守原则，因此，若连 C 都无解，那么原程序真的就不可能具备类型了；反方面，我们的约束又是充分的，因此，若 C 有解，那么的确有一种标注方法可使程序具备类型。

用定理形式把直觉写下来，即：

Theorem 1

设 $\varnothing \vdash t : T \mid Q_0 \to Q' \mid C$。那么：

$\exists$ 滤镜 σ: $\sigma \models C$

$\iff$ $\exists$ 一种标注方案使得 $\varnothing \vdash \hat{t} : T^*$

其中 $\hat{t}$ 是标注后的 t.

(而且额外地有 $T^* = \sigma(T)$)

这定理虽则正确，但却无法直接拿归纳法证明，原因在于 ∅ 的假设太弱了，会在 [T-FUN] [T-LET] 处卡住。因此，我们必须加强之。

def Γ 的实例化.

设 σ 是滤镜，Γ 是假设集.

$\Gamma = \{x_1 : T_1, x_2 : T_2, \cdots, x_n : T_n\}$.

又设滤镜 γ 满足 $\gamma \circ \sigma(T_i) =$ 实在的类型 $(i = 1, \cdots, n)$，那么，称

$\Gamma_\sigma := \{x_1 : \gamma \circ \sigma(T_1), \cdots, x_n : \gamma \circ \sigma(T_n)\}$

为 Γ 在 σ 作用下的一种实例化。

remark. 简言之，Γ_σ 即把 σ 逐项地作用于 Γ 上。如果结果中仍含有变元，则继续用 γ 将其化为实在类型。

e.g. $\Gamma := \{x : Int, y : U \blacktriangleright U, z : Bool \blacktriangleright U\}$

$\sigma := [U \mapsto Int] \circ [X \mapsto Int]$

那么 Γ_σ 可取成 $\{x : Int, y : Int \blacktriangleright (Bool \blacktriangleright Int),$ $z : Bool \blacktriangleright Int\}$（$(Bool \blacktriangleright Int)$ 被 γ 实例化了.）

Lemma 2 (Soundness)

若 $\Gamma \vdash t : T \mid Q \to Q' \mid C$ 且 $\sigma \models C$

那么存在一种 t 的标注版本 $\hat{t}$ 使得 $\Gamma_\sigma \vdash \hat{t} : T^*$，其中 T^* 是 T 在 σ 下的实例化（记为 T_σ）

proof. 关于 $\Gamma \vdash t : T \mid Q \to Q' \mid C$ 的结构作归纳。

[T-TRUE] [T-FALSE] [T-INT] 显然

[T-FUN] 由 I.H. 知，存在一种 t 的标注版本 $\hat{t}$ 使 $(\Gamma \cup \{x : X\})_\sigma \vdash \hat{t} : T^*$，因此

$\Gamma_\sigma \cup \{x : X\}_\sigma \vdash \hat{t} : T^*$. 故

$\Gamma_\sigma \vdash \lambda x : X_\sigma . \hat{t} \; : \; X_\sigma \blacktriangleright T^*$

这样一来，我们便寻得了合适的标注版本 $(\lambda x : X_\sigma : \hat{t})$，其类型 $X_\sigma \blacktriangleright T^*$ 恰为 $X \blacktriangleright T$ 在 σ 下的实例化。

[T-LET] 类似。

[T-APP] 因为 $\sigma \models C_1 \cup C_2 \cup \{T_1 = T_2 \blacktriangleright X\}$

故 $\sigma \models C_1$ 且 $\sigma \models C_2$ 且 $\sigma(T_1) = \sigma(T_2) \blacktriangleright \sigma(X)$.

由 I.H. 知

$$\Gamma_\sigma \vdash \hat{t}_1 : (T_1)_\sigma \quad , \quad \Gamma_\sigma \vdash \hat{t}_2 : (T_2)_\sigma$$

$$\Downarrow$$

$$\Gamma_\sigma \vdash \hat{t}_1 : (T_2)_\sigma \blacktriangleright X_\sigma$$

从而 $\Gamma_\sigma \vdash (\hat{t}_1\ \hat{t}_2) : X_\sigma$

[T-REC] [T-IF] [T-ARITH] 仿此作出。■

Lemma 3 (Completeness)

若 $\Gamma \vdash t : T \mid Q \to Q' \mid C$，且存在一种 t 的标注版本 $\hat{t}$，及纯实在的假设集 Γ'，满足 $\Gamma' \vdash \hat{t} : T^*$，那么，存在滤镜 σ：

$$\sigma \models C \text{ 且 } \Gamma' = \Gamma_\sigma \text{ 且 } T^* = T_\sigma .$$

proof. 习题. ■ （请格外留神 Q 的用场）

弄清楚解的本质后，求解之算法其实丝毫没有难度。我们把约束集 $C = \{T_1 = S_1, \cdots, T_n = S_n\}$ 换种写法：

$$\begin{cases} T_1 = S_1 \\ \vdots \\ T_n = S_n \end{cases}$$

这立马使人想起线性方程组以及解线性方程组的高斯消元法。确实，虽然我们此处面对的「元」是类型变元而非实数变元，但处理方法是相通的 —— 利用代入法逐步「消灭」变元即可。

e.g.1 $\{x = Int,\ y = x,\ z = x \blacktriangleright y\}$

$\overset{\text{「消去」x}}{\Longrightarrow} \{y = Int,\ z = Int \blacktriangleright y\}$ $\quad [x \mapsto Int]$

$\overset{\text{消去 y}}{\Longrightarrow} \{z = Int \blacktriangleright Int\}$ $\quad [y \mapsto Int] \circ [x \mapsto Int]$

$\overset{\text{消去 z}}{\Longrightarrow} \{\}$ $\quad [z \mapsto Int \blacktriangleright Int] \circ [y \mapsto Int] \circ [x \mapsto Int]$

其实，消去的次序是不要紧的：

$\{x=\mathrm{Int},\ y=x,\ z=x\blacktriangleright y\}$

$\overset{\text{消去}y}{\Longrightarrow}\ \{x=\mathrm{Int},\ z=x\blacktriangleright x\}\qquad [y\mapsto x]$

$\overset{\text{消去}z}{\Longrightarrow}\ \{x=\mathrm{Int}\}\qquad [z\mapsto x\blacktriangleright x]\circ[y\mapsto x]$

$\overset{\text{消去}x}{\Longrightarrow}\ \{\}\qquad [x\mapsto \mathrm{Int}]\circ[z\mapsto x\blacktriangleright x]\circ[y\mapsto x]$

虽然表现形式不同，但此处的解

$[x\mapsto \mathrm{Int}]\circ[z\mapsto x\blacktriangleright x]\circ[y\mapsto x]$

与先前的解

$[z\mapsto \mathrm{Int}\blacktriangleright\mathrm{Int}]\circ[y\mapsto \mathrm{Int}]\circ[x\mapsto \mathrm{Int}]$

是相等的。

e.g.2. $\{\mathrm{Int}\blacktriangleright x_1 = x_2\blacktriangleright \mathrm{Bool},\ x_1=x_2\}$

此时，第一条方程内含有一定结构，变元是包裹在内的。为此，我们把包裹打开

$\Rightarrow \{\mathrm{Int}=x_2,\ x_1=\mathrm{Bool},\ x_1=x_2\}$

$\overset{\text{消去}x_2}{\Longrightarrow}\ \{x_1=\mathrm{Bool},\ x_1=\mathrm{Int}\}\qquad [x_2\mapsto \mathrm{Int}]$

$\overset{\text{消去}x_1}{\Longrightarrow}\ \{\mathrm{Bool}=\mathrm{Int}\}\qquad [x_1\mapsto \mathrm{Bool}]\circ[x_2\mapsto \mathrm{Int}]$

↑ 无法继续，故无解。

把方法规范地写下来，即：

```
Solve(C):
  F := 恒等滤镜
  while C ≠ ∅ do
    equation := C.pop()
    T := equation.left
    S := equation.right
    • if T 是类型变元 then
        F := [T ↦ S] ∘ F
        C := C[T ↦ S]
    • else if S 是类型变元 then
        F := [S ↦ T] ∘ F
        C := C[S ↦ T]
    else if T = x ▸ y ∧ S = u ▸ v then
        C.push("x = u")
        C.push("y = v")
    else if T ≠ S then
        error
```

remark. 打红点的两处隐藏着问题。假设 $T=x$ 且 $S=x\blacktriangleright\ldots$，显然方程无解，但算法却无法识别之。试修正该缺陷。

为了分析方便，我们将修正片段写成递归形式：

```
Solve(C):
  if C = ∅ then I   //恒等滤镜
  else
    suppose C = {"T=S"} ∪ C₀
    [1] if T=S then Solve(C₀)
    [2] else if T是类型变元 ∧ T∉S then
          Solve(C₀[T↦S]) ∘ [T↦S]
    [3] else if S是类型变元 ∧ S∉T then
          Solve(C₀[S↦T]) ∘ [S↦T]
    [4] else if T=x▸y ∧ S=u▸v then
          Solve(C₀ ∪ {"x=u", "y=v"})
    [5] else
          error
```

易证Solve在任何输入C下均能终止——只需考虑 $|C| + (\# \text{ of } \blacktriangleright \text{ in } C)$ 的严格递减性即可。下面论证Solve的正确性。

Lemma 4

$\forall C$，若 $\exists \sigma: \sigma \vDash C$，那么Solve(C)必能正常终止。

proof. 前面已说Solve(C)必能终止，因此我们仅关注其正常/异常与否。我们对Solve在C下的运行步数t作归纳。

<u>初始情形</u>. $t=1$

可能性有如下两种：

(a) $C=\emptyset$. 那么Solve(C)的确终止（正常）

(b) $C \neq \emptyset$，且取出的"$T=S$"使得算法进入分支[5]．这意味着[1]-[4]均不被满足。很显然，$T \neq S$，而且T与S要么呈现包含关系，要么具有不同的「▸结构」。无论如何，$\nexists \sigma: \sigma(S)=\sigma(T)$，从而 $\nexists \sigma \vDash C$，矛盾与

<u>归纳情形</u> 设t时成立，考虑t+1．

这与初始情形类似． ■

remark. 这意味着[1]-[4]覆盖了全体合理情形。

Lemma 5

若Solve(C)正常中止，则它返回的滤镜 σ 必然是一个解，即 $\sigma \vDash C$.

proof. 仍关于 Solve(C)的步数作归纳。

初始情形 $t=1$.

此时唯一可能为 $C=\emptyset$. Solve(C)返回 $\sigma = I$，而显然 $I \vDash \emptyset$.

归纳情形 设 t 时成立，考虑 $t+1$.

[1] 设 Solve(C_0)返回 δ。由归纳假设知 $\delta \vDash C_0$. 又 $S=T$，故 $\delta(S)=\delta(T)$，从而 $\delta \vDash C$。于是 Solve(C)返回的滤镜 δ 的确为 C 的解。

[2] 设 Solve ($C_0[T \mapsto S]$)返回了 δ。由 I.H. 有 $\delta \vDash C_0[T\mapsto S]$.

而 Solve(C)返回的是 $\sigma := \delta \circ [T \mapsto S]$.

我们将证明 $\sigma \vDash C$.

(a) $\sigma(S) = \delta \circ [T\mapsto S]\,(S) = \delta(S)$ 因为 $T \notin S$

$\sigma(T) = \delta \circ [T \mapsto S]\,(T) = \delta(S)$

从而 $\sigma(S) = \sigma(T)$

(b) 对于 C_0 中的任意约束 "$x_0 = y_0$"

$\sigma(x_0) = \delta \circ [T\mapsto S]\,(x_0) = \delta(x_0[T\mapsto S])$

$\sigma(y_0) = \delta \circ [T\mapsto S]\,(y_0) = \delta(y_0[T\mapsto S])$

而因为 "$x_0[T\mapsto S] = y_0[T\mapsto S]$" $\in C_0[T\mapsto S]$

且 $\delta \vDash C_0[T\mapsto S]$

故 $\delta(x_0[T\mapsto S]) = \delta(y_0[T\mapsto S])$

故 $\sigma(x_0) = \sigma(y_0)$

综合(a)(b)知 σ 满足"$S=T$"和 C_0 中全体约束，即 $\sigma \vDash C$.

[3] 同上.

[4] 设 Solve($C_0 \cup \{$"$x=u$", "$y=v$"$\}$) 返回 δ. 由 I.H. 有 $\delta \vDash C_0 \cup \{$"$x=u$", "$y=v$"$\}$ 即有 $\delta(x)=\delta(u)$ 及 $\delta(y)=\delta(v)$。及 $\delta \vDash C_0$. 那么

$\delta(T) = \delta(x \blacktriangleright y) = \delta(x) \blacktriangleright \delta(y)$

$\delta(S) = \delta(u \blacktriangleright v) = \delta(u) \blacktriangleright \delta(v)$

从而 $\delta \vDash C$. ■

其实，Lemma5 还可推广为：Solve(C) 返回的非但是个解，而且是个「通解」.

def 通解.

设 C 是约束集，$\sigma^* \vDash C$.

如果 $\forall \sigma \vDash C$ 均 $\exists \gamma: \sigma = \gamma \circ \sigma^*$ 那么

称 σ^* 是 C 的通解.

直观而言，即任意解均可由 σ^* 实例化得到。

Lemma6 (Lemma5 推广)

若 Solve(C) 正常中止，则其返回值是 C 的通解。

proof. 习题 ■

综合起来，我们有如下定理

Theorem7

Solve(C) 能正常中止 $\iff \exists \sigma \vDash C$.

且 Solve(C) 在解存在时总能给出通解.

正是由于这优异的性质，我们可以给语言引入「多态性」。比如，函数 $\lambda x.x$ 之类型为 $X \to X$（X 是类型变元），因而适用于多种场合，而不仅限于作用在整数上。

为了支持多态，只需将类型推导规则 [T-LET] 改成

$$\frac{\Gamma \vdash t_1 : T_1 \mid Q \to Q' \mid C_1 \qquad \Gamma \vdash t_2[x/t_1] : T_2 \mid Q' \to Q'' \mid C_2}{\Gamma \vdash (\mathrm{let}\ x = t_1\ \mathrm{in}\ t_2) : T_2 \mid Q \to Q'' \mid C_1 \cup C_2}$$

相当于给 x 的类型生成了多个彼此独立副本，以支持不同场景下的使用。如是的概念在库函数的设计中至关重要。

实用的解释器/编译器通常不采取代入 $t_2[x/t_1]$ 的方式，而是像我们讨论单步执行时那样（实现多态），将类型 T_1 包装起来放入 Γ，以后，需要取 x 的类型时再对其作实例化。