理解 Algorithm W

Jeremy Mikkola
原作者

Chuigda Whitegive
翻译

Alias Qli
校对

Anqur
类型论支持

Flaribbit
校对
Typst 技术支持

Lyra
校对

译者前言

本文是对 Jeremy Mikkola 的文章 Understanding Algorithm W 的中文翻译，并使用 Typst 重写了原文中的部分公式。一些记号和措辞经过改写，某种程度上可能更加可读，或是与其他论文中的记号更加一致。本文附有 Python 实现。

1. 简介

我发现 Algorithm W 真的很难理解，所以我写了这篇博客来帮助自己理解它，希望对你也有用。

本文面向使用过带有类型推导的语言（例如 Haskell, OCaml 和 Rust）并对类型推导感兴趣的人群。熟悉这些语言对于阅读本文来说不是必须的，但会有所帮助。

本文不涉及理论，也不会证明 Algorithm W。我更感兴趣的是 Algorithm W 实际上做了什么以及为什么这么做。

术语表和其他资源的链接在本文最下方（包括最初提出 Algorithm W 的论文）。

2. 语言

我们用一门简单的程序设计语言来阐述 Algorithm W 的运作方式。这门语言非常简单，以至于它甚至不包含 `if` 语句或是数据结构这样的东西，但对于展示类型推导中所需的各种技术而言已经足够。扩展语言使其具有更多特性，同时仍然应用这些技术来推导类型也是完全可行的。

这门语言是纯函数式的，风格上类似于 ML 系语言。语言中的所有结构（甚至是函数定义）都是表达式。事实上，整个程序就是一个巨大的表达式。

语言支持以下表达式：

- 字面值，例如 `123` 或者 `false`
- 变量
- 函数应用（也就是函数调用）
- 函数抽象（也就是函数声明/定义）
- `let` 绑定

非常小的语言，对吧？

2.1. 语法

表达式可以用括号括起来以正确地进行分组。

字面值的语法跟你想的一样，例如 `55` 或者 `true`。变量的语法也一样，例如 `x` 或者 `foobar`。

函数应用有一点不一样。比如说你有一个函数 `plus`，用参数 `x` 和 `y` 调用它的语法是 `plus x y`。如果你要先对 `x` 调用函数 `g`，然后再对结果调用 `f`，你就得使用括号：`f (g x)`。或者写成 `(f (g x))` 也行。某种意义上来说，`plus x y` 其实是 `(plus x) y`，不过我们稍候再讨论。

[1] 原文无此 λ。

函数抽象的语法就是字母 λ 加上单个参数[1]（为什么只有一个参数？还是一样，稍后再说）后面跟着一个点，然后是一个作为函数体的表达式。例如：`λx. plus 1 x` 是一个将给定的数字加上 1 的函数。函数本身没有名字。将函数定义用括号括起来是很常见的。函数的函数体可以是另一个函数，例如 `λx. λy. plus x y`（实际上是 `(λx. (λy. (plus x y)))`）。

`let` 绑定包含一个变量名、一个表达式作为变量的值，以及一个要求值的表达式（其中涉及到刚刚绑定的变量）。例如：`let square = (λx. times x x) in square 10`。

以下是语法的大致 BNF：

```
program := expr

expr := literal
      | variable
      | '(' expr ')'
      | abstraction
      | application
      | let

literal := integer | boolean

integer := digit+
digit := '0'..'9'

boolean := 'true' | 'false'

variable := letter+
letter := 'a'..'z' | 'A'..'Z'

abstraction = 'λ' variable '.' expr
application = expr expr

let := 'let' variable '=' expr 'in' expr
```

2.2. 语义

变量可以引用一个值或者函数。有两种方式可以引入变量：使用 `let` 绑定，或是通过函数参数。一个变量声明可以隐藏一个已有的变量声明：子表达式中的变量名指向新的变量，而在子表达式之外原先的变量仍然可以访问。注意在形如 `let x = plus x 1 in x` 这样的绑定中，`plus 1 x` 中使用的是 `x` 的旧定义，而 `in x` 中使用的则是 `x` 的新定义。（在接下来的例子里，我会假设我们已经有了一组像 `plus` 这样的内建函数）。

该语言中的函数可以访问其被定义时所在定义域中的其他变量。因此，尽管技术上来说这门语言只有单参函数，我们仍然像这样可以实现多参函数：

```
let hypotenuse = (λx. (λy. sqrt (times x x) (times y y)))
in hypotenuse 5 10
```

函数 `hypotenuse` 接受一个数字（作为参数 `x`），返回另一个函数。返回的这个函数同样接受一个数字（作为参数 `y`），但它返回一个数字。当这个函数像 `hypotenuse 5 10` 这样被调用时，它实际上的意思是“将函数 `hypotenuse` 函数应用到 `5` 上，然后把返回的这个函数应用到 `10` 上”，也就是 `(hypotenuse 5) 10`，括号中的部分先求值。

让所有函数都只接受单个参数可以简化类型推导（因为这样就不用检查函数调用时传参的数量了）。这同时也意味着你可以部分地应用函数，例如：

```
let double = times 2
in double 10
```

将 2 应用于 `times` 会得到另一个函数，我们可以给这个函数一个名字，并在之后使用它。

最后要指出一点：这门语言从语法上禁止了递归调用（在函数定义的内部函数名尚未被绑定，因此无法在函数定义内引用函数自身）。尽管如此，使用不动点组合子将非递归函数转换为递归函数还是可以的，虽说靠 Algorithm W 无法推出不动点组合子的类型[2]。

3. 类型系统

这门语言中包含几种基本类型，例如 int 和 bool。

语言中还包含函数类型，它将输入类型和输出类型用一个中缀运算符 $\to$ 组合起来。例如，$\text{int} \to \text{int}$ 表示一个函数接受一个 int 且返回一个 int。$(\text{int} \to \text{int}) \to \text{int}$ 表示一个函数接受另一个函数 f_{temp} 且返回 int，其中函数 f_{temp} 接受 int 且返回 int。$\text{int} \to (\text{int} \to \text{int})$ 实际上就是表示一个函数接受两个 int 且返回一个 int，但严谨地来说，这个函数接受一个 int，返回另一个函数 f_{temp}，而函数 f_{temp} 接受 int 且返回 int。

除了像这样的具体类型之外，还有类型变量。类型通常写作单个希腊字母，例如 α，β 等，有时也带有下标，例如 α_0，β_2。不要把类型变量和程序设计语言中的变量类比，相比之下，类型变量更像是代数中的未知量：一个变量总归会有一个值（另一个类型），我们只是暂时不知道这个值是什么。例如，$\alpha \to \alpha$ 是一个函数类型，它接受某些东西（我们还没有指定），并返回同类型的某些东西。

总结下来，一个类型就是以下三者之一：

- 像 int，bool 这样的具体类型
- 像 α，β 这样的类型变量
- 函数类型 $\tau_1 \to \tau_2$，其中包含两个其他类型

就这么简单。这门语言中不支持类或者用户自定义结构，所以没有表示这些东西的类型的方式。

类型变量带来了一个使得类型推导更为有趣的小细节：泛型。让我们看看恒等函数（原样返回其输入的函数）：`(λx. x)`。它的类型是 $\alpha \to \alpha$。我们在书写它的类型时必须使用类型变量 α，因为我们仅凭阅读函数定义尚不知道 `x` 的类型。也就是说，在实际调用这个函数的时候，`x` 会是一个具有某种类型的值。比如说我们用一个整数来调用这个函数：

```
let id = (λx. x)
in id 123
```

这里输入类型就变成了整数。而如果我们要在另一个计算中使用整数结果：

```
let id = (λx. x)
in square (id 123)
```

我们怎么知道 `(id 123)` 能求值出具有正确类型的值呢？这个时候我们就要用到包含类型变量的类型 $\alpha \to \alpha$ 了。我们知道输入给 `id` 的类型是 int，我们将其与函数匹配，看看如果我们将 α 替换为 int 情况会变得如何。当我们将类型变量替换成一个具体的类型时，就要替换所有这个类型变量出现的地方。在这个例子里，这也就意味着把两个 α 都替换成 int，结果就是 $\text{int} \to \text{int}$，也就是我们知道了这个函数返回的是 int。

不过我刚刚说的其实有一点不对的地方：我们并不是直接替换掉类型 α。让我们看另一个例子，看看为什么简单地“替换”是不对的，然后我再告诉你实际上应该怎么做。

```
let id = (λx. x)
in (id square) (id 123)
```

[2]译注：原文为“though this type system can't actually handle the fixed-point combinator”，直译为“这个类型系统实际上不能处理不动点组合子”。译者推测这里的意思是用本文中的类型推导算法 Algorithm W 推不出不动点组合子的类型。不动点组合子 Y 的类型 $\forall\alpha.(\alpha \to \alpha) \to \alpha$ 还是可以在这个类型系统中表示的，并且如果把不动点组合子作为内建函数，它仍可以正常地参与其他部分的类型推理，并被用于模拟递归和互递归。参见 Principal type-schemes for functional programs。

这里，我们先是把 `square` 传给 `id` 函数（`id` 函数会原样返回 `square`），然后又把 `123` 传给 `id`（也是原样返回）。就程序的输出而言，这个程序等价于 `let id = (λx. x) in square 123`（或者更简单地，`square 123`）。

然而，这个程序中的类型很有意思。假设 `square` 具有类型 $\text{int} \to \text{int}$，那就意味着第一次调用 `id` 时 `id` 应该具有类型 $(\text{int} \to \text{int}) \to (\text{int} \to \text{int})$，而第二次调用 `id` 时它却具有类型 $\text{int} \to \text{int}$。如果我们要替换 $\alpha \to \alpha$ 中的 α，我们该用哪个类型来替换呢？不管我们选择哪一边，它都会为另一边的 `id` 赋予错误的类型。

所以实际上，我们需要在每次用到函数 `id` 的时候“实例化”它的类型。这就意味着将其中一些函数变量用新的类型变量（还没有使用过的类型变量）替代。我们不会用两个相互冲突的类型来替换 a，而是首先实例化 `id` 的类型，为第一次调用生成 $\beta \to \beta$，为第二次调用生成 $\gamma \to \gamma$。之后，我们将 β 替换为 $\text{int} \to \text{int}$，这样这个类型就可以被用于 `(id square)`；将 γ 替换为 int，这样类型就可以被用于 `(id 123)`。具体细节会在系列与泛化一节中涉及。

4. 类型推导

在描述如何在这门语言中推导表达式的类型之前，我得先讲解两组基本类型操作。

4.1. 归一化（Unification）与替换（Substitution）

归一化只需要很少的代码就能实现，但它却是类型推导算法的核心。我们就是用归一化来分析什么东西是什么类型，并找出类型错误的。可以说，类型推导中的其他部分就是准备调用归一化函数所需的数据，并使用归一化函数的结果。

所以归一化到底是什么呢？它是一个用来找出“要使得两个类型相等，需要作出哪些改变”的算法。如果无法让两个类型相等，就意味着程序中有类型错误。要让两个类型相等，归一化算法唯一能做的只有替换类型中的类型变量（可能是用另一个类型变量来替换）。也就是说，如果两个类型不一样，那么里面最好有某些我们能改变的类型变量！

让我们看一些例子。归一化已经相等的两个类型，例如 int 和 int，不会进行任何实际的操作。归一化完全不同的两个类型，例如 int 和 bool 或是 bool 和 $\text{int} \to \text{bool}$，会产生一个类型错误（不管怎么捣鼓类型变量都不可能让它们相等）。

而当两个类型中的一个有类型变量，而另一个类型里有些其他的东西时，会发生变量的取代（replacement）。例如，将 int 和 α 归一化会将 α 换成 int。我们称之为替换（substitution），即我们用 int 取代了 α。如果输入的方式不一样（比如次序是 α 和 int），结果也是一样的。如果两边都是类型变量，例如 α 和 β，那么我们就用右边的类型变量取代左边的（比如在这个例子里用 β 取代 α）。

函数类型怎么办呢？等我先围绕替换多说一点再讨论那个。

替换（substitution）就是从类型变量到“要用来替代类型变量的东西”的映射。比如说我们要把 α 换成 $\beta \to \beta$，把 γ 换成 int。从现在开始，我会用这样的记号表示替换：$\{\alpha : \beta \to \beta, \gamma : \text{int}\}$。我们可以将替换应用到类型上，也就是寻找类型中需要被取代的类型变量，并取代它们。例如，对类型 $\alpha \to (\beta \to \gamma)$ 应用上述替换会产生类型 $(\beta \to \beta) \to (\beta \to \text{int})$。

归一化算法（如果成功的话）会返回一个能使得两个类型相等的替换。换句话来说，如果归一化算法为类型 τ_1 和 τ_2 返回了替换 S，则 $\text{apply}(\tau_1, S) \equiv \text{apply}(\tau_2, S)$。

好了，回到函数类型上来。以 $\text{int} \to \alpha$ 和 $\beta \to \text{bool}$ 这两个类型为例，归一化算法首先归一化左边（int 和 β），得到替换 $\{\beta : \text{int}\}$。接着，归一化算法再归一化右边（α 和 bool），得到替换 $\{\alpha : \text{bool}\}$。接下来，归一化算法将两个替换结合起来，得到 $\{\alpha : \text{bool}, \beta : \text{int}\}$。如果我们对两个输入的类型应用这个替换，就会得到同一个类型：$\text{int} \to \text{bool}$。

还跟得上吧？好，现在是时候看看更高级的情况了。如果我们归一化 $\alpha \to \alpha$ 和 $\text{int} \to \gamma$ 会怎么样？（如果我们将恒等函数应用于一个整数，我们就会看到这样的归一化，其中 γ 用来捕获函数的返回类型。）和之前一样，我们先归一化左边的 α 和 int 得到 $\{\alpha : \text{int}\}$。这里的情况有点微妙，

我们得先把另一个 α（函数类型右边的哪个）换成 int，然后再继续操作：正如之前所说，当你取代一个类型变量的时候，你得把它所有出现的地方都换掉[3]。所以在归一化左边之后，两个类型就变成了 $\text{int} \to \text{int}$ 和 $\text{int} \to \gamma$。现在我们可以归一化右边的 int 和 γ 了，这会得到 $\{\gamma : \text{int}\}$。最后我们结合两个替换，得到 $\{\alpha : \text{int}, \gamma : \text{int}\}$。恭喜，你刚刚检查出了对一个整数应用恒等函数会获得一个整数！

有两种归一化的特殊情况，我得提一下：

1. 通常，当你把一个类型变量和什么东西归一化的时候，会产生一个替换，其中包含将这个类型变量取代为另一个类型（即使它是另一个类型变量）。例外就是当两边是同一个类型变量的时候，结果会是一个空替换 {}（因为不需要改变什么就能让两个类型相等）。
2. 在取代一个类型变量时，你不能用一个包含了这个类型变量的类型取代它。比如，不能用 $\alpha \to \beta$ 取代 α（用 $\gamma \to \beta$ 取代就是可以的）。遇到这种情况（比如归一化 $\alpha \to \delta$ 和 $(\alpha \to \beta) \to \text{int}$ 的左边的时候）应该报告类型错误。如果允许这种替换就会搞出无限大的类型。在许多文献中，这种类型也被称为“无限类型（infinite type）”。

4.2. 无限类型

（如果你是第一次阅读本文，你可以跳过这一节）

要搞出无限类型，一个简单的办法就是 `(λf. f f)`。这个表达式用我们的语言来写的话不是很容易读，所以我用 Python 重写一份：

```python
def something(f):
  return f(f)
```

这样的例子很有意思，因为我们没法用一个（有限的）类型来表达 `f` 的类型，而我们传给 `f` 的某些值可以让这个函数正常运行，不会引起崩溃（比如恒等函数就可以）。

让我们绕个弯，看看为什么类型会变得无限大。假设 `something` 函数的类型是：

$$\tau_f \to \tau_{f\text{的返回值}}$$

我们知道 `f` 本身必须也是函数，那么我们可以将 τ_f 分解为 $\tau_{f\text{的参数}}$ 和 $\tau_{f\text{的返回值}}$：

$$\left(\tau_{f\text{的参数}} \to \tau_{f\text{的返回值}}\right) \to \tau_{f\text{的返回值}}$$

看看我们到底能不能搞清楚 $\tau_{f\text{的参数}}$ 是什么。因为我们把 `f` 作为参数传给了 `f`，那么参数的类型必须和 `f` 的类型相等。好，那就试试：

$$\left(\left(\tau_{f\text{的参数}} \to \tau_{f\text{的返回值}}\right) \to \tau_{f\text{的返回值}}\right) \to \tau_{f\text{的返回值}}$$

等下，我们还是没能从式子里干掉 $\tau_{f\text{的参数}}$。好在我们知道它必须和 `f` 的类型相等：

$$\left(\left(\left(\tau_{f\text{的参数}} \to \tau_{f\text{的返回值}}\right) \to \tau_{f\text{的返回值}}\right) \to \tau_{f\text{的返回值}}\right) \to \tau_{f\text{的返回值}}$$

哦天啊，我们好像永远都无法摆脱 $\tau_{f\text{的参数}}$ 了。没错，正因如此，`something` 函数的类型是一个无限类型。

事实：当你对不动点组合子的定义作类型检查时，就会遇到无限类型的问题。

4.3. 应用与组合替换

目前为止我只非形式化地描述了如何应用一个替换。接下来让我试着严谨一点。让我根据被应用替换的类型的类别来进行分类讨论：

- 对具体类型（例如 int 和 bool）应用替换会原样返回该类型。
- 对类型变量应用替换时：

[3] “在所有出现的地方替换”实现起来其实是“在类型推导算法当前正在处理的类型上替换”。之后，调用者应该在所有需要的地方应用这个替换。

- 如果这个类型变量已经有一个替换了，返回那个替换。比如说，如果类型变量是 γ，而存在替换 $\{\gamma : \text{bool}\}$，则返回 bool；
- 否则，原样返回类型变量。比如说，如果，如果类型变量是 β，而存在替换 $\{\gamma : \text{bool}\}$，则返回 β。

· 对函数类型应用替换时，分别对 $\to$ 的左侧和右侧的类型应用替换，然后用替换后的类型构造一个新的函数类型。

现在我可以更准确地描述如何“结合”或者“组合”两个替换了。比如说有两个替换 S_1 和 S_2，那么应用结合之后的替换 S_3 应该和先后分别应用两个替换 S_1，S_2 的效果相同。也就是说，若 $S_3 = \text{compose-substitutions}(S_1, S_2)$，则 $\text{apply}(\tau, S_3)$ 和 $\text{apply}(\text{apply}(\tau, S_1), S_2)$ 的行为一致。

例：设 τ 为 $\alpha \to \beta$，S_1 为 $\{\beta : \gamma \to \text{int}\}$，$S_2$ 为 $\{\alpha : \text{bool}, \gamma : \text{float}\}$。对 τ 应用 S_1 会得到 $\alpha \to (\gamma \to \text{int})$，再对其应用 S_2 就会得到 $\text{bool} \to (\text{float} \to \text{int})$。有了这个例子，让我们试着定义如何结合两个替换：

> （**这个不对**）结合两个替换就是将两组要替换的东西的映射结合起来。如果两个替换都为同一个变量提供了一个替换物，左边的替换胜出。

这个定义已经很接近了，但还是漏了点东西。要搞清楚到底漏了什么，让我们先按这个规则构造一个 S_3 出来。因为 S_1 和 S_2 中没有冲突，我们可以直接组合两个替换，得到 $S_3 = \{\beta : \gamma \to \text{int}, \alpha : \text{bool}, \gamma : \text{float}\}$。当我们将这个替换应用到 τ（也就是 $\alpha \to \beta$）上之后，我们得到的是 $\text{bool} \to (\gamma \to \text{int})$。哦不，这可不对，我们没把 γ 替换成“float”。这是因为我们本来应该先应用 $\{\beta : \gamma \to \text{int}\}$，然后再在它的结果上应用 $\{\gamma : \text{float}\}$。

能否将这一行为写进我们的替换 S_3 中呢？$S_3 = \{\beta : \text{float} \to \text{int}, \alpha : \text{bool}, \gamma : \text{float}\}$ 如何？看起来这能行。我们修改了 β 的替换，把 γ 的取代考虑在内了。从效果上来说，我们已经在 β 上应用了 γ 的替换。所以正确的定义应该是这样：

> 要组合两个替换 S_1 和 S_2，首先对 S_2 中所有用于取代的类型应用 S_1，然后将 S_1 中的所有条目加入 S_2 中，除非 S_2 中已经存在该条目。

多数情况下，将 S_1 应用于 S_2 中的类型不会改变任何东西。不过你可以构造出这样的例子：

```
let id = (λx. x)
in (id id) (id id)
```

记得看完算法部分之后回到这个例子上，看看你能不能学以致用（这可能得花上个把小时，过程是有点味同嚼蜡，不过它能让你更好地理解这个算法的机制）。

5. 系列（Scheme）与泛化（Generalization）

一个系列（scheme，或称类型系列 type scheme）[4] 就是在类型上附加一些额外的信息：一个由类型变量组成的列表。列表中的类型变量是类型中出现的类型变量的子集（一些地方将这些类型变量称为“自由类型变量”）。通常这个列表是空的，但有时候对于函数类型而言，这个列表中会存在一些变量。

光这么说太抽象了，还是看看我们的老伙计恒等函数吧。我之前说恒等函数的类型是 $\alpha \to \alpha$，这其实并不准确。如果我们把恒等函数在一个变量里存储起来：

```
let id = (λx. x) in ...
```

变量 `id` 的类型实际上是一个类型系列：$\text{Scheme}([\alpha], \alpha \to \alpha)$。事实上，任何变量的的类型总是用类型系列来表示的（不管变量是否在函数中）。

好，现在我们用 Scheme 把类型 $\alpha \to \alpha$ 包了起来，附加上了一个列表 $[\alpha]$，并且列表中的所有类型变量都在类型中出现了。现在你可能会问，“这到底有什么用？”

[4] 译注：在翻译这部分时，一个主要的困难在于如何翻译 Scheme 这个词。译者根据柯林斯词典的解释“a systematic arrangement of correlated parts; system”将其意译为“系列”。

回到例子上，让我们调用一下恒等函数：

```
let id = (λx. x)
in (id square) (id 123)
```

（想起来了吗？这个例子在类型系统一节里出现过。）这里，id 函数需要具有两种类型：当它被应用于 square 的时候，它的类型应该是 $(\text{int} \to \text{int}) \to (\text{int} \to \text{int})$，而当它被应用于 123 的时候，它的类型却应该是 $\text{int} \to \text{int}$。怎么办呢？这就要用到一种叫做实例化（instantiation）的方法。

每次我们调用 id 函数的时候，我们根据 id 的类型系列 $\text{Scheme}([\alpha], \alpha \to \alpha)$，将列表中的每个类型变量用一个全新的类型变量取代。这个过程会得到一个类型（而不是类型系列）。这里，α 在列表中出现过，所以我们用一个全新的类型变量取代它。注意两次调用 id 函数的时候我们都要分别这么做一次，每次使用不同的、全新的类型变量。那么第一个 id（被应用于 square）的类型就可以被实例化为 $\alpha_0 \to \alpha_0$，而第二个 id 的类型可以被实例化为 $\alpha_1 \to \alpha_1$。

现在我们就可以将 α_0 替换为 $(\text{int} \to \text{int})$ 而不会影响到另一个 id 了。同理，我们可以将 α_1 替换为 int。（我们怎么知道要把 α_0 替换为 $\text{int} \to \text{int}$ 呢？这是归一化函数告诉我们的。稍候我们会看到如何调用归一化函数。）

现在你就知道类型系列是什么，以及如何实例化一个类型系列来创造新的类型了。类型系列

$$\text{Scheme}([\tau_1, \tau_2, ..., \tau_n], \sigma)，其中\ \tau_i 是 \sigma\ 中的变量$$

其实就是：

$$\forall \tau_1 \forall \tau_2 ... \forall \tau_n . \sigma，其中\ \tau_i 是 \sigma\ 中的变量$$

这就是为什么类型系列有时候被称为泛型类型（generic type）、量化类型（quantified type）、全称类型（universal type）或者多态类型（polymorphic type）[5]。

不过我们还没说过如何创建类型系列。类型系列是由一个类型（就是出现在类型系列中的那个类型）和环境中的其他信息创建出来的。类型可以被原封不动地抄到类型系列里，所以我们只要搞清楚哪些类型要被全称量词 $\forall$ 量化就可以了。办法很简单：对于类型中的所有类型变量，如果它**没有**在类型环境中出现过，就用全称量词 $\forall$ 量化它。

哈，我操，类型环境？啥？

简单来说，类型环境就是我们已经知道类型（并且仍在作用域内）的东西。用一个例子来解释会容易不少：

```
(λx. let f = (λy. x) in f 123)
```

注意 f 的类型。它是个函数，所以一定具有函数类型。参数类型是什么？可以是任何东西，所以我们用一个类型变量来表示它。方便阅读起见，我就称之为 τ_y。那么返回类型呢？就是 x 的类型，我称之为 τ_x。那么函数 f 的类型就是 $\tau_y \to \tau_x$。

当我们将 f 应用 123 的时候，这会把参数类型从 τ_y 变成 int，但返回类型丝毫不受影响，它还是 τ_x。也就是说，当我们实例化 f 的类型的时候，我们应该只替换掉 τ_y 而保持 τ_x 不变。要在类型系列中表示这一点，我们说 f 的类型是 $\forall \tau_y . \tau_y \to \tau_x$。

回到我们原先的问题上：我们怎么知道应该只用量词 $\forall$ 量化 τ_y，而不是把 τ_x 和 τ_y 都用量词 $\forall$ 量化呢？因为 τ_x 在类型环境中出现过。那什么是类型环境？就是当前所有已经定义的变量的类型。x 是在 f 外面定义的，并且具有类型 τ_x，所以 τ_x 不应该被 $\forall$ 量化，就算它在类型 $\tau_y \to \tau_x$ 中出现过。

值得注意的是，只有 let 绑定创建的变量才会被泛化。函数的参数不会被泛化（它们的类型系列没有任何 $\forall$）。

总结下来：

[5]译注：后一种记号以及这一段文本是译者根据个人理解自己加的，并且译者之后会继续使用第二种记号。

- 类型系列 $\forall\tau_1\forall\tau_2...\forall\tau_n.\sigma$ 由一个类型 σ 和若干被全称量词 $\forall$ 量化的类型 τ_i 构成，其中 τ_i 是 σ 中的类型变量
- 当声明变量时，泛化变量的类型
- 泛化就是从一个类型创造出一个类型系列的过程
- 类型系列中被全称量词 $\forall$ 量化的类型变量，就是类型系列的类型中那些**不**存在于类型环境中的类型变量
- 当使用变量时，实例化它的类型系列
- 实例化一个类型系列会产生一个类型
- 实例化就是将类型系列中被全称量词 $\forall$ 量化的类型变量全部替换为全新的类型变量

6. 推导表达式的类型

现在，~~周瑜！~~终于，我们可以描述完整的算法了！

类型推导函数接受两个参数：要推导类型的表达式，以及当前的类型环境。类型环境就是一个从变量名（语言中的变量名，不是类型变量）到类型系列的映射。类型环境中包含了当前位于作用域中的变量。当我们开始推导整个程序的类型时，类型环境是空的。随着类型推导的推进，算法会逐步将信息添加到类型环境中。

类型推导函数有两个返回值：表达式的类型，以及一个替换。你可以将替换视为算法返回所了解的程序其他部分（除了表达式）的一种方式。

算法递归地遍历表达式树。当一个表达式包含另一个表达式（比如函数应用包含一个作为函数的表达式和一个作为参数的表达式）时，算法对子表达式递归地调用自身。细节我马上会提到，但我得现在说明一些事情：有时候算法会向递归调用传递一个修改过的类型环境的**副本**。注意被修改的是一个副本，原先的类型环境不会改变。

毫无疑问，算法需要对语言中不同类型的表达式做不同的事。它们大致上遵循这样一个模式：递归地处理子表达式，在下降的过程中向类型环境中添加信息，并在上升的过程中，结合返回的替换信息，逐渐分析出所有东西的类型。

让我们依次讲解每种表达式的处理方法：

6.1. 字面量，例如 `123`

超级简单。我们返回这个值的类型（比如说，如果值是 `false` 就返回 bool，如果值是 `123` 就返回 int）以及一个空的替换。

6.2. 变量，例如 `foo`

也不太坏。我们在类型环境中查找变量名，得到一个类型系列，并将其实例化成一个类型，把这个类型和一个空的替换一并返回。如果类型环境中找不到这个变量，就意味着这个变量尚未被定义，这是一个错误。

注意，实例化是发生在变量被使用的时候，而不是在函数被应用的时候。

6.3. 函数抽象，例如 `λx. times 2 x`

记住这门语言里的所有函数都有且仅有一个参数。首先，创建一个新的类型变量来表示这个参数的类型。方便起见，我们将参数称为 `x`，将类型变量称为 ν_0。接着，创建一个类型环境的副本，在其中加入一项 `x` 并将其与类型系列 ν_0（这个类型系列中没有任何全称量词，即没有被量化的类型变量）关联；记住，被修改的是副本，原先的类型环境不变）。

使用这个更新后的类型环境，递归地对函数的函数体应用算法，这会返回函数体的类型 τ和一个替换 S。

接着，对我们刚刚创建的类型变量 ν_0 应用 S 得到替换后的类型变量 ν：这一步是利用我们分析函数体得到的信息反推参数需要是什么类型，反映在 ν_0 中的类型变量被什么东西取代。当然，应用替换 S 也可能不会产生任何效果。

现在我们可以创建函数的类型了。函数具有函数类型（惊讶吧）。参数类型就是我们对 ν_0 应用替换 S 后得到的 ν，而返回类型就是函数体的类型 τ。

例子时间到。比如说分析函数体的递归调用返回了类型 $\tau = \text{bool}$ 和替换 $S = \{\nu_0 : \text{int}\}$，我们对类型变量 ν_0 应用 S，得到 $\nu = \text{int}$。把它们组装起来就能得到函数的类型 $\nu \to \tau = \text{int} \to \text{bool}$。

6.4. 函数应用，例如 `foo 55`

在分析函数应用的类型时，我们不能直接把函数的返回类型当作整个表达式的类型。考虑向恒等函数（具有类型 $\alpha \to \alpha$）传递一个数字（具有类型 int）。恒等函数会原样返回其输入，所以返回值类型显然应该是 int，而不是 α。实际上我们需要做的是搞清楚如何让函数类型中 $\to$ 左边的类型和实际参数的类型相等（也就是说，找到一个能使得它们相等的替换）。在这之后，我们就有足够的信息来分析返回类型了。

1. 创建一个新的类型变量 π，这个类型变量会被用于“捕获”函数的返回类型。这个类型变量不会马上被用到，但我们第一步就创建它，这样我们创建类型变量的顺序就有一定意义（因而方便调试）。
2. 递归地调用算法来分析被调函数的类型，获得一个替换 S_1 和被调函数的类型 τ_1。如果函数是一个变量（例如 `foo`），那么这次递归调用会帮我们实例化这个变量的类型系列。

现在，我们就有了函数应用左边的类型信息（τ_1）

3. 对类型环境中的所有类型应用替换 S_1 以创建一个新的类型环境 Γ_1。为什么要这么做？因为我们刚刚推导出了左边的类型，而在这个过程中我们发现的某些信息可能对右边的类型推导有用（考虑形如 `(times x) (id x)` 的表达式，`x` 在左右两边都出现了），所以在继续之前我们应该更新类型环境。这一步对接下来推导出右边的正确类型非常重要。
4. 使用新创建的类型环境 Γ_1，递归地对右边应用算法，获得替换 S_2 和作为参数的表达式的类型 τ_2。
5. 对 τ_1 应用 S_2得到 τ_1'，和刚才一样，这是为了利用我们推导右边的类型时获得的信息，完善左边的类型。

现在我们有了左右两边的类型，但这还不是整个函数应用的类型。

6. 创建一个函数类型 $\tau_2 \to \pi$，将其与 τ_1'（函数的类型）归一化，并得到替换 S_3。

这会将 π 与函数实际的返回类型归一化，这样我们就能很方便地获得返回类型（只要看 π 被替换成了什么类型就可以了）。

归一化函数是唯一会向替换中添加信息的情况。也就是说，如果一个表达式里没有任何函数应用，那么你就可以确定类型检查算法对这个表达式会返回一个空的替换。

7. 马上就完成了，接下来准备两个返回值就好。首先我们得制作要返回的替换：把替换 S_1, S_2 和 S_3 组合起来，得到一个大的替换 S。
8. 最后，我们对 π 应用 S，得到 π'，也就是函数应用实际返回的类型。

步骤很多，所以我们总结一下：如果你检查出函数的类型是 $\tau_1 \to \tau_2$，而参数的类型是 τ_3，那么引入一个新的类型变量 π，归一化 $\tau_1 \to \tau_2$ 和 $\tau_3 \to \pi$，最后再搞清楚 π 会被替换成什么类型，就搞定了。

6.5. `let` 绑定，例如 `let x = 123 in x`

最后一个了，再坚持一下！

1. 推导出第一个表达式（例子中的 `123`）的类型，获得替换 S_1 和类型 τ_1。

2. 对类型环境中的所有类型应用替换 S_1 以创建一个新的类型环境 Γ_1，基于这个类型环境将类型 τ_1 泛化成类型系列 $\forall ... \tau_1$。之后，我们不再使用 Γ_1。
3. 创建一个新的类型环境 Γ_2，其中包含变量（例子中的 `x`）的类型系列 $\forall ... \tau_1$。

4. 对 Γ_2 中的所有类型应用替换 S_1，创建一个新的类型环境（又一个!）Γ_3。这是因为我们在推导第一个表达式的类型的途中获得了一些信息，我们要在第二个表达式的推导中用上这些信息。
5. 使用 Γ_3 对第二个表达式递归调用算法，得到替换 S_2 和第二个表达式的类型 τ_2。

算法返回的类型就是 τ_2，返回的替换则是 S_1 与 S_2 的组合。

这次我也总结一下。首先推导第一个表达式的类型（τ_1），然后将其泛化成类型系列，将这个类型系列加入类型环境中，然后推导第二个表达式的类型。

有一件很重要的事，我得在这强调一下。在上面第二步中，我们在创建类型系列时泛化了一个类型。如果你回去看函数抽象那一节，你会看到我们也（为函数参数）创建了类型系列，但那个类型系列中没有任何被全称量词量化的类型变量。也就是说，由 `let` 引入的变量可以被实例化出不同的类型，而函数参数引入的变量不行。比如以下代码是合法的：

```
let id = (λx. x)
in (id square) (id 44)
```

而以下代码是非法的：

```
(λid. (id square) (id 44)) (λx. x)
```

在所有用到函数参数的地方，函数参数都必须具有同一个类型，而由 `let` 引入的变量则可以具有不同的类型。多数语言的类型推导都是这样的（在 Haskell 中你可以启用编译器特性“N 阶泛型”并给编译器提供一个行得通的类型）。

此外也得说一下这个算法如何能够避免自引用/递归绑定。（如果我们有递归绑定的话，我们就可以搞出递归函数，因为因为函数体中可以引用函数本身。）原理很简单：在我们对 `let` 中第一个表达式进行类型检查的时候，变量名还没有被加入到类型环境中。也就是说在第一个表达式中对该变量名的任何引用都会引用到这个变量名之前的绑定，而不是由 `let` 表达式引入的绑定。

举个例子，表达式

```
let x = 5
in let x = square x
   in x
```

的结果是 25。

7. 结论

这就是 Algorithm W。它对于不同类型的表达式应用我们所引入的不同类型的操作

- 对字面量作类型检查：不涉及任何类型操作。
- 对变量作类型检查：需要在类型环境中查找类型系列并进行实例化。
- 对函数抽象作类型检查：需要创建一个新的类型变量，修改类型环境并应用替换。
- 对函数应用作类型检查：同样需要创建一个新的类型变量并应用替换。此外它还要使用归一化算法，并且组合替换。
- 对 `let` 绑定作类型检查：同样需要修改类型环境，需要应用和组合替换。
- 对后三种表达式作类型检查还涉及到对子表达式递归应用算法。

或者写成伪代码[6]：

[6]译注：这段伪代码是译者加的，取自论文 Generalizing Hindley-Milner Type Inference Algorithms，有修改。

$$W :: TypeEnvironment \times Expression \to Substitution \times Type$$

$$\begin{aligned}
W(\Gamma, x) &= ([], instantiate(\sigma)), \text{where } (x : \sigma) \in \Gamma \\
W(\Gamma, \lambda x \to e) &= \textbf{let } (S_1, \tau_1) = W(\Gamma \setminus x \cup \{x : \beta\}, e), \text{fresh } \beta \\
&\quad \textbf{in } (S_1, S_1\beta \to \tau_1) \\
W(\Gamma, e_1 e_2) &= \textbf{let } (S_1, \tau_1) = W(\Gamma, e_1), \\
&\qquad (S_2, \tau_2) = W(S_1\Gamma, e_2), \\
&\qquad S_3 = unify^{7}(S_2\tau_1, \tau_2 \to \pi), \text{fresh } \pi^{8} \\
&\quad \textbf{in } (S_3 \circ S_2 \circ S_1, S_3\pi) \\
W(\Gamma, \textbf{let } x = e_1 \textbf{ in } e_2) &= \textbf{let } (S_1, \tau_1) = W(\Gamma, e_1), \\
&\qquad (S_2, \tau_2) = W(S_1\Gamma \setminus x \cup \{x : generalize(S_1\Gamma, \tau_1)\}, e_2), \\
&\quad \textbf{in } (S_2 \circ S_1, \tau_2)
\end{aligned}$$

希望这篇文章能让你理解诸如泛化或归一化之类的单个操作是如何工作的，以及它们的用途。也许在反复阅读本文之后，你甚至能理解它们在每种类型的表达式上的作用。

注释

如果以上文本存在内容性错误，请在 Twitter 上报告给原作者；如果存在翻译错误，请通过 GitHub issue 报告问题。

最广归一（Most General Unifier）

本算法中的归一化过程返回的是“最广归一”，即返回的是最泛化的类型（或者说，能够获得最泛化的类型的替换）。你可以理解为这是让两个类型相等所需要的最少的修改。

比如说，我们要归一化类型 $\alpha \to \alpha$ 和 $\beta \to \gamma$，一个可行的归一（替换）是 $\{\alpha : \text{int}, \beta : \text{int}, \gamma : \text{int}\}$。这当然可以成功地归一化两个类型（都会变成 $\text{int} \to \text{int}$），但这不是最泛化的类型。一个可行的最广归一是 $\{\beta : \alpha, \gamma : \alpha\}$（两个类型都会变成 $\alpha \to \alpha$）。另一个可行的的最广归一是 $\{\alpha : \gamma, \beta : \gamma\}$（两个类型都会变成 $\gamma \to \gamma$）。

最广归一有一个有趣的性质，~~这段读不通不翻了。~~

术语表

- 抽象（Abstraction）：创建一个函数
- 应用（Application）：
  - 应用替换（Application of substitutions）：将一个类型中的某些类型变量用其它类型取代
  - 函数应用（Function application）：调用一个函数
- BNF：巴科斯-瑙尔范式
- 绑定（Binding）：创建一个变量
- 泛化（Generalize）：从类型创建类型系列
- 恒等函数（Identity function）：一个原样返回其参数的函数
- 实例化（Instantiate）：从类型系列创建类型
- 系列（Scheme）：见“类型系列”
- 替换（Substitution）：从类型变量到用来取代它们的类型的映射
- 类型环境（Type environment）：作用域中现存的类型系列
- 类型系列（Type vcheme）：一个带有类型变量列表的类型
- 类型变量（Type variable）：类型系统中的一个未知量

[7]译注：$unify$ 原为 mgu，即 Most General Unify。
[8]译注：π 原为 β。

- 归一（Unifier）：一个能使得两个类型相等的替换
- 变量（Variable）：语言中的一个变量可以存储某些值

参见

- Algorithm W Step by Step：使用 Haskell 实现了 Algorithm W
- Principal type-schemes for functional programs：给出了完整的 Algorithm W，读起来比较紧凑
- Excessive explaination：首个尝试事无巨细地解释 Algorithm W 的一系列博客

Your reality

Every day, I imagine a future where I can be with you
In my hand is a pen that will write a poem of me and you
The ink flows down into a dark puddle
Just move your hand - write the way into his heart!
But in this world of infinite choices
What will it take just to find that special day?
What will it take - just to find that special day?

Have I found everybody a fun assignment to do today?
When you're here, everything that we do is fun for them anyway
When I can't even read my own feelings
What good are words when a smile says it all?
And if this world won't write me an ending
What will it take just for me to have it all?

Does my pen only write bitter words for those who are dear to me?
Is it love if I take you, or is it love if I set you free?
The ink flows down into a dark puddle
How can I write love into reality?
If I can't hear the sound of your heartbeat
What do you call love in your reality?

And in your reality, if I don't know how to love you
I'll leave you be