基本多态类型检查

Luca Cardelli
原作者

Chuigda Whitegive
翻译

Flaribbit
Typst 技术支持

译者介绍

本文是对 1984 年 Luca Cardelli 论文 Basic Polymorphic Typechecking 的翻译。

1. 介绍

多态类型检查（Polymorphic Type Checking）的基础是由 Hindley 所设计的类型系统 [Hindley 69]，并在之后被 Milner 再次发现并进行了扩展 [Milner 78]。这种算法在 ML 语言中被实现 [Gordon 79] [Milner 84]。这种类型系统和 Algol 68 一样支持编译期检查、强类型和高阶函数，但因为它允许多态 —— 也就是定义在多种类型的参数上具有统一行为的函数，所以它更加灵活。

Milner 的多态类型检查算法是非常成功的：它健全、高效、功能丰富且灵活。它也可以被用于在没有类型信息（untyped）或者只有部分类型信息（partially typed）的程序中推导出类型。

然而，多态类型检查只在一小部分人群中得到了使用。目前的出版物中只有 [Milner 78] 描述了这个算法，并且其行文相对专业，更多地面向理论背景。

为了让这个算法能够为更多的人所接受，我们现在提供了一个 ML 实现，这个实现非常类似于 LCF，Hope 和 ML [Gordon 79] [Burstall 80] [Milner 84] 中所使用的实现。尽管有时候人们会为了逻辑清晰而牺牲性能，但这个实现还是相对高效的，并且可以被实际地用于大型程序的类型检查。

本文中的 ML 实现只考虑了类型检查中最基本的情况，并且许多对常见编程语言结构的扩展都相当明显。目前已知的非平凡扩展（本文不会讨论）包括重载、抽象数据类型、异常处理、可更新的数据、带标签的记录（labelled record）以及 联合（union）类型。许多其他的扩展仍然处于研究之中，并且人们普遍认为在类型检查的理论与实践中，重要的发现尚未浮出水面。

本文给出了看待类型两种方式：由类型方程组成的系统，以及类型推理系统，并尝试非形式化地将它们与实现联系起来。

2. 一个简单的应用序（Applicative）语言

本文所使用的语言是一个简单的带有常量的类型化 lambda 演算，这被认为是 ML 语言的核心。求值机制（call-by-name 或者 call-by-value）并不影响类型检查。

下面给出表达式的具体语法；相应的抽象语法在本文结尾处由程序中的 `Term` 类型给出（不含解析器与打印）。

```
Term ::= Identifier
       | 'if' Term 'then' Term 'else' Term
       | 'fun' Identifier '.' Term
       | Term Term
       | 'let' Declaration 'in' Term
       | '(' Term ')'

Declaration ::= Identifier '=' Term
              | Declaration 'and' Declaration
              | 'rec' Declaration
              | '(' Declaration ')'
```

只要在初始环境中包含一系列预定义的标识符，就可以向这个语言中引入新的数据类型。这样在扩展语言时就不需要修改语法以及类型检查算法。

例如，以下程序定义了一个阶乘函数，并且对 `0` 应用它：

```
let rec factorial n =
    if zero n
    then succ 0
    else (times (pair n (factorial (pred n))))
in factorial 0
```

3. 类型

一个类型可以是一个用于表示任意类型的类型变量（*type variable*），如 α、β，也可以是一个类型算子（*type operator*）。**int**（整数类型）和 **bool**（布尔类型）是无参（*nullary*）类型算子。而 $\rightarrow$（函数类型）和 $\times$（积类型）这样的参数化（*parametric*）类型算子则接受一个或者多个类型作为参数。以上两个算子最一般化的形式是 $\alpha \rightarrow \beta$（表示任意函数的类型）和 $\alpha \times \beta$（任意序对的类型）。包含类型变量的类型被称为多态（*polymorphic*）的，而不含类型变量的类型被称为单态（*monomorphic*）的。常规的编程语言，例如 Pascal 和 Algol 68 等，其中的类型都是单态的。

如果同一个类型变量在表达式中出现了多次，那么这个表达式表示上下文依赖（*contextual dependencies*）。例如，表达式 $\alpha \rightarrow \alpha$ 表达了函数类型的定义域（*domain*）和值域（*codomain*）之间的上下文依赖。类型检查的过程就是匹配类型算子和实例化（*instantiate*）类型变量的过程。当实例化一个类型变量时，这个类型变量所在的其他位置必须被替换为相同的类型。例如，对于类型变量 $\alpha \rightarrow \alpha$，$\mathbf{int} \rightarrow \mathbf{int}$、$\mathbf{bool} \rightarrow \mathbf{bool}$ 和 $(\varepsilon \times \xi) \rightarrow (\varepsilon \times \xi)$ 都是合法的实例化。依据上下文进行实例化的过程是通过归一化（*unification*）[Robinson 65] 来完成的，这也是多态类型检查的基础。当匹配过程遇到两个不同的类型算子（例如 `int` 和 `bool`），或是尝试将类型变量实例化为一个包含它自身的项目（例如将 α 实例化为 $\alpha \rightarrow \beta$，这会产生一个循环结构）时，匹配过程会失败。后者通常会在对自应用（*self-applicative*）（例如 `fun x. (x x)`）的表达式进行类型检查时出现，因而这种表达式是不合法的。

我们来看一个简单的例子。恒等函数 `I = fun x. x` 具有类型 $\alpha \rightarrow \alpha$，因为它将任何类型映射到它自身。在表达式 `(I 0)` 中，`0` 的类型（也就是 **int**）被匹配为函数 `I` 的定义域，于是在这个上下文中，`I` 的类型被特化（*specialize*）为 $\mathbf{int} \rightarrow \mathbf{int}$，于是 `(I 0)` 的类型就是这个特化的 `I` 的值域类型，也就是 **int**。

一般而言，表达式的类型是由一系列原语算子的类型和与语言结构对应的类型结合规则确定的。初始环境可为布尔、整数、序对和列表包含以下原语（其中 $\rightarrow$ 是函数类型算子，**list** 是列表类型算子，$\times$ 是笛卡尔积）：

$$
\begin{array}{ll}
\mathbf{true}, \mathbf{false} & : \mathbf{bool} \\
0, 1, \ldots & : \mathbf{int} \\
\mathbf{succ}, \mathbf{pred} & : \mathbf{int} \rightarrow \mathbf{int} \\
\mathbf{zero} & : \mathbf{int} \rightarrow \mathbf{bool} \\
\mathbf{pair} & : \alpha \rightarrow \beta \rightarrow (\alpha \times \beta) \\
\mathbf{fst} & : (\alpha \times \beta) \rightarrow \alpha \\
\mathbf{snd} & : (\alpha \times \beta) \rightarrow \beta \\
\mathbf{nil} & : \alpha\ \mathbf{list} \\
\mathbf{cons} & : (\alpha \times \alpha\ \mathbf{list}) \rightarrow \alpha\ \mathbf{list} \\
\mathbf{hd} & : \alpha\ \mathbf{list} \rightarrow \alpha \\
\mathbf{tl} & : \alpha\ \mathbf{list} \rightarrow \alpha\ \mathbf{list} \\
\mathbf{null} & : \alpha\ \mathbf{list} \rightarrow \mathbf{bool}
\end{array}
$$

4. `length` 函数的类型

在描述类型检查算法之前，让我们先来探讨一下下面这个计算列表长度的简单递归程序的类型：

```
let rec length =
    fun l .
        if null l
        then 0
        else succ (length (tl l))
```

（为了方便接下来的讨论，我们没有使用等效但更优雅的 `length l = ...` 写法，而是写作 `length = fun l . ...`）

`length` 的类型是 α **list** $\to$ **int**，这是一个泛型类型，因为 `length` 函数可以对任意类型的列表使用。们可以使用两种方式来描述我们推导出这一类型的过程。原理上来说，类型检查就是建立一个由类型约束（*type constraint*）组成的系统，然后求解这个系统中的类型变量。实践上来说，类型检查是自下而上地审视整个程序，在向根部上升的过程中匹配并综合类型；表达式的类型通过其子表达式的类型和上下文中包含的类型约束计算而来，而预定义标识符的类型已经包含在初始环境中。我们审视程序并进行匹配的顺序并不会影响最终的结果，这是类型系统和类型检查算法的重要特性之一。

`length` 函数的类型约束系统是：

[1] **null** : α **list** $\to$ **bool**
[2] **tl** : β **list** $\to$ β **list**
[3] 0 : **int**
[4] **succ** : **int** $\to$ **int**

[5] **null** l : **bool**
[6] 0 : γ
[7] **succ** (**length** (**tl** l)) : γ
[8] **if null** l **then** 0 **else succ** (**length** (**tl** l)) : γ

[9] **null** : $\delta \to \varepsilon$
[10] l : δ
[11] **null** l : ε

[12] **tl** : $\varphi \to \xi$
[13] l : φ
[14] **tl** l : ξ
[15] **length** : $\theta \to \iota$
[16] **tl** l : θ
[17] **length** (**tl** l) : ι

[18] **succ** : $\kappa \to \lambda$
[19] **length** (**tl** l) : κ
[20] **succ** (**length** (**tl** l)) : λ

[21] l : μ
[22] **if null** l **then** 0 **else succ** (**length** (**tl** l)) : ν

[23] **fun** l. **if null** l **then** 0
else succ (**length** (**tl** l)) : $\mu \to \nu$

[24] **length** : π

[25] **fun** l. **if null** l **then** 0
else succ (**length** (**tl** l)) : π

行 1~4 代表预定义全局标识符的类型约束，这部分信息是已知的。条件结构（整个 `if` 表达式，译者注）施加了约束 5~8：测试（`null l`）的结果必须是布尔型，并且两个分支表达式必须具有相同的类型 γ。γ 同时也是整个条件表达式的类型。程序中的四个函数调用施加了约束 9~20：对于函数调用而言，函数符号必须具有一个函数类型（例如 9 中的 $\delta \to \varepsilon$）；参数必须具有和函数定义域相同的类型（例如 10 中的 δ）；函数调用的结果必须具有和函数值域相同的类型（例如 11 中的 ε）。整个 lambda 表达式 23 具有类型 $\mu \to \nu$，其参数具有类型 μ，而其函数体具有类型 ν。最后的定义结构施加了被定义变量（**length** 24）必须具有和其定义 25 相同的类型。

对 `length` 函数作类型检查需要：(i) 确保整个约束系统是一致的（比如绝对不能推导出 **int** = **bool**），并且 (ii) 对 π 求解这些约束。`length` 的类型可以按照如下方式推出：

$$\pi = \mu \to \nu \text{ by } [25, 23]$$
$$\nu = \varphi = \beta \textbf{ list} \text{ by } [21, 13, 12, 2]$$
$$\nu = \gamma = \textbf{int} \text{ by } [22, 8, 6, 3]$$

还需要更多工作才能证明 β 完全不受约束并且整个系统是一致的。下一节中描述的类型检查算法会系统性地完成这一工作，像约束系统上的定理证明器一样简单。

下面是一个自下而上的 `length` 函数的类型推导，与类型检查算法实际的操作步骤更为接近；约束的一致性（即没有类型错误）也在过程中得到了检验：

[26]	l	: δ	[10]
[27]	**null** l	: **bool**	
	$\varepsilon = \textbf{bool}; \delta = \alpha$ **list**		[11, 9, 1]
[28]	0	: **int**	
	$\gamma = \textbf{int}$		[6, 3]
[29]	**tl** l	: β **list**	
	$\varphi = \beta$ **list**; $\xi = \beta$ **list**; $\beta = \alpha$		[26, 27, 12~14, 2]
[30]	**length** (**tl** l)	: ι	
	$\theta = \beta$ **list**		[15~17, 29]
[31]	**succ** (**length** (**tl** l))	: **int**	
	$\iota = \kappa = \textbf{int}$		[18~20, 4, 30]
[32]	**if null** l **then** 0		
	else succ (**length** (**tl** l))	: **int**	[5~8, 27, 28, 31]
[33]	**fun** l. **if null** l **then** 0		
	else succ (**length** (**tl** l))	: β **list** $\to$ **int**	
	$\mu = \beta$ **list**; $\nu = \textbf{int}$		[21~23, 26, 27, 32]
[34]	**length**	: β **list** $\to$ **int**	
	$\pi = \beta$ **list** $\to$ **int**		[24, 25, 33, 15, 30, 31]

注意上面的过程已经处理了递归：34 比较了 `length` 函数的两个实例（函数定义处和递归调用处）的类型。

5. 类型检查

基本的算法可以被描述为：

1. 当 lambda 绑定引入一个新变量 `x` 时，赋予它一个全新的类型变量 α，表示它的类型有待根据它出现的上下文进一步确定。将序对 <x, α> 存储到一个环境（在程序中称作 `TypeEnv`）中。每当变量 `x` 出现时，从环境中检索 `x` 并提取出 α（或是它的任何一个介入实例化）作为 `x` 在此处的类型。
2. 在条件结构中，将 `if` 条件的类型与 **bool** 进行匹配，将 `then` 和 `else` 分支的类型归一化以确保整个表达式具有一个类型。
3. 在函数抽象 `fun x. e` 中，创建一个新的上下文而，为 `x` 赋予一个全新的类型变量，然后在这个上下文中推断 `e` 的类型。
4. 在函数应用 `f a` 中，将 `f` 的类型与 $A \to \beta$ 归一化，其中 A 是 `a` 的类型，而 β 是一个全新的类型变量。也就是说，`f` 必须具有函数类型，并且其定义域可以和 A 归一化；整个函数应用的类型是 β（或是其任何一个实例化）。

要描述 `let` 表达式及其引入变量的类型检查，我们需要引入泛型（*generic*）类型变量记号。考虑以下表达式：

```
fun f. pair (f 3) (f true) [Ex1]
```

在 Milner 的类型系统中，这个表达式无法被类型检查，而上述算法会给出一个类型错误。事实上，`f` 的第一次出现（`f 3`）先给出 `f` 的类型是 $\mathbf{int} \to \beta$，随后第二次出现（`f true`）又给出了一个类型 $\mathbf{bool} \to \beta$，而它无法与第一个类型归一化。

像 `f` 这样由 lambda 引入的标识符，它的类型变量是非泛型（*non-generic*）的。因为就像这个例子中所展现的，类型变量在 `f` 出现的所有地方共享，并且他们的实例可能会有冲突。

你可以试着给表达式 *Ex1* 一个类型，例如你可以试着给它 $(\alpha \to \beta) \to (\beta \times \beta)$。这在 `f (fun a. 0)` 这样的场景下是可以的，它能正常得到结果 `(pair 0 0)`。然而这种类型系统总的来说是不健全（*unsound*）的：例如 `succ` 的类型可以与 $\alpha \to \beta$ 匹配，因而会被 `f` 接受，并被错误地应用到 `true` 上。Milner 的类型系统有一些健全的扩展能处理 *Ex1* 这样的表达式，但它们不在讨论本文的范围内。

因此，为 lambda 引入的标识符的异质（*heterogeneous*）使用标定类型是一个基本问题。实践中这基本上是可以接受的，因为像 *Ex1* 这样的表达式不是很有用，也不是特别必要。我们需要在 `let` 引入的标识符的异质使用问题上做得更好。考虑以下表达式：

```
let f = fun a. a
in pair (f 3) (f true) [Ex2]
```

必须找到一种办法来为上述表达式标定类型，否则多态函数就没法在同一个上下文里被应用到不同的类型上，多态也就没有用了。好在我们这里的局面比 *Ex1* 要好，因为我们确切地知道 `f` 是什么，并且可以利用这一信息来分别处理它的两次出现。

在这个例子中，`f` 具有类型 $\alpha \to \alpha$；像 α 这样出现在 `let` 引入的标识符的类型（并且不在外围的 lambda 引入的标识符类型中出现）的类型变量，我们称它为泛型（*generic*）的。在 `let` 引入的标识符的不同实例化中，泛型的类型变量可以有不同的值。实现时，可以为 `f` 的每次出现创建一个 `f` 的拷贝。

不过，在复制一个类型的时候，我们必须小心，不要复制非泛型的类型变量。非泛型的类型变量必须被共享。以下表达式和 *Ex1* 一样是不合法的，因为 `g` 的类型是非泛型的，而这个类型会被传播到 `f` 中：

```
fun g. let f = g
in pair (f 3) (f true) [Ex3]
```

同样地，你不能给这个表达式一个像 $(\alpha \to \beta) \to (\beta \times \beta)$ 这样的类型（考虑对这个表达式应用 `succ`）。

泛型变量的定义是：

表达式 `e` 的类型中的一个类型变量是泛型的，当且仅当它不出现在任何表达式 `e` 外围的 lambda 表达式的绑定器中。（原文：*A type variable occurring in the type of an expression `e` is generic iff it does not occur in the type of the binder of any lambda-expression enclosing `e`.*）

注意一个在 lambda 表达式内部非泛型的类型变量，在表达式外部可能是泛型的。例如在 *Ex2* 中，`a` 的类型 α 是非泛型的，而 `f` 的类型 $\alpha \to \alpha$ 是泛型的。

要确定一个变量是否是泛型的，我们随时维护一个非泛型变量的列表：若类型变量不在这个列表中，则它是泛型的。每当进入 lambda 表达式时扩充该列表；而每当离开 lambda 表达式时恢复原先的列表，这样 lambda 表达式引入的类型变量就又泛型了。当复制一个类型的时候，我们必须只赋值泛型的变量，非泛型的变量则应该被共享。在将非泛型类型变量归一化到一个项目（term）上时，该项目中包含的所有类型变量都会变为非泛型的。

最后我们需要考虑递归定义：

```
let rec f = ... f ...
in ... f ...
```

我们可以将 `rec` 用 Y 不动点（具有类型 $(\alpha \to \alpha) \to \alpha$）展开：

```
let f = Y fun f. ... f ...
in ... f ...
```

显然可知，在递归定义中的 `f` 的实例（中的类型变量）必须是非泛型的，而在 `in` 中的实例是泛型的。

5. 要对 `let` 作类型检查，我们首先检查它的声明部分，获得一个包含标识符与类型的环境，这个环境之后会用于 `let` 体（*body*）的检查
6. 检查 `let` 的声明部分时，检查其中所有的定义 $x_i = t_i$，每个定义会在环境中引入一个序对 <x_i, T_i>，其中 T_i 是 t_i 的类型。对于存在（互）递归的声明 $x_i = t_i$，我们首先为所有要定义的 x_i 创建一个包含序对 <x_i, α_i> 的环境，这里 α_i 是非泛型的类型变量（它们在进入声明作用域的时候被插入到非泛型变量的列表中）。所有的 t_i 在这个环境中进行类型检查，然后再将 T_i 与 α_i（或是其实例化）进行匹配。

6. 关于模型、推理系统和算法的题外话

有两种方法能形式化地描述类型的语义。最基本方式的是给设计一个数学模型，通常是将每个类型表达式映射到一组值（具有该类型的值）；这里的基本困难在于找到 $\to$ 算子的数学含义 [Scott 76] [Milner 78] [MacQueen 84]。

另一种与之互补的方法是定义一个由公理（axiom）和推理规则组成的形式系统，在这个系统中可以证明一个表达式具有某种类型。语义模型和形式系统之间的联系非常紧密。语义模型通常是定义一个形式化系统的指南，并且每个形式系统必须是自洽的，这通常通过展示它的语义模型来体现。

在一个好的形式系统中，我们基本上可以证明我们“已知”的所有东西为真（根据直觉，或者因为它在语义模型里就是为真的）。在有了一个好的形式系统之后，我们“差不多”就可以忘记语义模型而，后续的工作就会轻松很多。

相比于语义模型，类型检查和形式系统的关系更加严格，这是其句法的（syntactic）性质所决定的。类型检查算法一定程度上来说就是在实现一个形式系统，通过向该系统提供一个证明定理的过程。形式系统比任何算法都更简单也更基础，所以类型检查算法的最简形式就是它所实现的形式系统。此外，在设计一个类型检查算法时，最好首先为其定义一个形式系统。

并非所有的形式系统都能支持类型检查算法。如果一个形式系统过于强大（比如说，我们可以用它证明很多东西），那么它很可能是不可决断的（*undecidable*），因而无法为其设计一个决策过程。类型检查通常仅限于可决断的类型系统，在这些类型系统上可以设计类型检查算法。不过有时不可决断的类型系统可以用不完备的启发式（*heuristcs*）类型检查处理（截至发文为止尚无

实践），这些类型检查算法只尝试在这个系统中证明命题，但可能在某些时候放弃。实践上来说这是可以接受的，因为程序的复杂度总得有个限制：its meaning could get out of hand long before the limits of the typechecking heuristics are reached.

即使是可决断的类型系统，所有类型检查算法也可能是指数复杂度的，这也需要用启发式方法来处理。人们已经在多态类型系统中处理重载时成功地运用了这一方法 [Burstall 80]。

下一节展示了一个用于多态类型检查的推理系统。现在我们对类型检查有了两种不同的视角：一种是“求解由类型方程组成的系统”，我们在前面已经看到了，而另一种就是“在形式系统中证明命题”，我们接下来会看到的。这两种视角之间可以互换，但后者提供了更多的见解，因为它将类型的语义和算法连接了起来。

7. 一个推理系统

在接下来的推理系统中，类型的语法被扩展为类型量词 $\forall\alpha.\tau$. 在 Milner 的类型系统中，所有类型变量都是隐式地在顶层被量化的。例如，$\alpha \to \beta$ 实际上是 $\forall\alpha.\forall\beta.\alpha \to \beta$。然而，量词不能在类型表达式中嵌套。

如果一个类型具有形式 $\forall\alpha_1....\forall\alpha_n.\tau$，其中 $n \geq 0$ 且 τ 中没有量词，则我们说这个类型是浅显（*shallow*）的。我们的推理系统允许构造非浅显的类型，但不幸的是没有类型检查算法能被用在它们上面。因此，我们只关心浅显类型的推理。我们引入量词是因为这有助于解释泛型/非泛型类型变量的行为，它们可以与自由/量化类型变量完全对应。[Damas 82] 中则提出了一种略微不同的推理系统，其中规避了非浅显的类型。

以下是推理规则集合。其中 *VAR* 是公理，其他规则都是推理。横线读作“蕴含”。记号 $A \vdash e : \tau$ 表示给定一个假设集合 A，我们可以推导出表达式 e 具有类型 τ。一个假设就是表达式 e 中一个变量的类型，它可以是自由类型。记号 $A.x : \tau$ 表示将集合 A 与假设 $x : \tau$ 取并集；记号 $\tau[\sigma/\alpha]$ 表示将 τ 中 α 的所有自由出现替换为 σ。

[VAR] $\quad A.x : \tau \vdash x : \tau$

[COND] $$\frac{A \vdash e : \mathbf{bool} \quad A \vdash e' : \tau \quad A \vdash e'' : \tau}{A \vdash (\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'') : \tau}$$

[ABS] $$\frac{A.x : \sigma \vdash e : \tau}{A \vdash (\lambda x.e) : \sigma \to \tau}$$

[APP] $$\frac{A \vdash e : \sigma \to \tau \quad A \vdash e' : \sigma}{A \vdash (e\ e') : \tau}$$

[LET] $$\frac{A \vdash e' : \sigma \quad A.x : \sigma \vdash e : \tau}{A \vdash (\mathbf{let}\ x = e'\ \mathbf{in}\ e) : \tau}$$

[REC] $$\frac{A.x : \tau \vdash e : \tau}{A \vdash (\mathbf{rec}\ x.e) : \tau}$$

[GEN] $$\frac{A \vdash e : \tau}{A \vdash e : \forall\alpha.\tau} \quad (\alpha \text{ 不是 } A \text{ 中的自由变量})$$

[SPEC] $$\frac{A \vdash e : \forall\alpha.\tau}{A \vdash e : \tau[\sigma/\alpha]}$$

（译者注：*APP* 原作 *COMB*，依 Wikipedia 改）

我们首先来看这样一个例子，我们可以推导出恒等函数最泛型的类型：`fun x. x` : $\forall\alpha.\alpha \to \alpha$

$$\frac{\dfrac{x : \alpha \vdash x : \alpha \quad [VAR]}{\vdash (\textbf{fun}\ x.x) : \alpha \to \alpha \quad [ABS]}}{\vdash (\textbf{fun}\ x.x) : \forall\alpha.\alpha \to \alpha \quad [GEN]}$$

一个特化版本的恒等函数可以从泛型的类型中被推导出来：

$$\frac{\vdash (\textbf{fun}\ x.x) : \forall\alpha.\alpha \to \alpha}{\vdash (\textbf{fun}\ x.x) : \textbf{int} \to \textbf{int}} \quad [SPEC]$$

或者更直接地：

$$\frac{x : \textbf{int} \vdash x : \textbf{int} \quad [VAR]}{\vdash (\textbf{fun}\ x.x) : \textbf{int} \to \textbf{int} \quad [ABS]}$$

我们可以扩展上述推理过程，来揭示 `(fun x. x)(3)` $: \textbf{int}$：

$$\frac{\dfrac{x : \textbf{int}, 3 : \textbf{int} \vdash x : \textbf{int}\ [VAR]}{3 : \textbf{int} \vdash (\textbf{fun}\ x.x) : \textbf{int} \to \textbf{int}\ [ABS]} \quad 3 : \textbf{int} \vdash 3 : \textbf{int}\ [VAR]}{3 : \textbf{int} \vdash (\textbf{fun}\ x.x)(3) : \textbf{int}\ [APP]}$$

以下是一个非法的派生，其中包含非浅显的类型。这种类型可以被用于 `(fun x. x x)`，而我们的算法无法处理这种表达式。

$$\frac{\dfrac{\dfrac{x : \varphi \vdash x : \varphi\ [VAR]}{x : \varphi \vdash x : \varphi \to \varphi\ [SPEC]} \quad x : \varphi \vdash x : \varphi\ [VAR]}{x : \varphi \vdash xx : \varphi\ [APP]}}{\vdash (\textbf{fun}\ x.x\ x) : \varphi \to \varphi\ [ABS]}$$

$$\text{here } \varphi = \forall\alpha.\alpha \to \alpha$$

注意观察 $\forall\alpha.\alpha \to \alpha$ 是经规则 *SPEC* 将 α 替换为 $\forall\alpha.\alpha \to \alpha$，最终被实例化成 $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$ 的。

现在我们来展示如何推导出 `(let f = fun x.x in pair (f 3) (f true))` $: \textbf{int} \times \textbf{bool}$：

令 $A = \{3 : \textbf{int}, \textbf{true} : \textbf{bool}, \textbf{pair} : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \times \beta\}$

$\varphi = \forall\alpha.\alpha \to \alpha$

则
$$\frac{\dfrac{A.f : \varphi \vdash f : \varphi}{A.f : \varphi \vdash f : \textbf{int} \to \textbf{int}} \quad A.f : \varphi \vdash 3 : \textbf{int}}{A.f : \varphi \vdash f\ 3 : \textbf{int}}$$

$$\frac{\dfrac{A.f : \varphi \vdash f : \varphi}{A.f : \varphi \vdash f : \textbf{bool} \to \textbf{bool}} \quad A.f : \varphi \vdash \textbf{true} : \textbf{bool}}{A.f : \varphi \vdash f\ \textbf{true} : \textbf{bool}}$$

$$\frac{\begin{array}{l} A.f : \varphi \vdash f\ 3 : \textbf{int} \\ A.f : \varphi \vdash f\ \textbf{true} : \textbf{bool} \\ \underline{A.f : \varphi \vdash \textbf{pair} : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha \times \beta} \\ \cdots \end{array}}{A.f : \varphi \vdash \textbf{pair}\ (f\ 3)(f\ \textbf{true}) : \textbf{int} \times \textbf{bool}}$$

$$\frac{A \vdash \textbf{fun}\ x.x : \varphi \quad A.f : \varphi \vdash \textbf{pair}\ (f\ 3)(f\ \textbf{true}) : \textbf{int} \times \textbf{bool}}{A \vdash (\textbf{let}\ f = \textbf{fun}\ x.x\ \textbf{in}\ \textbf{pair}\ (f\ 3)(f\ \textbf{true})) : \textbf{int} \times \textbf{bool}}$$

注意对于假设 $f : \forall\alpha.\alpha \to \alpha$，我们可以分别将 **int** 和 **bool** 代入 α 来进行两次独立的实例化。也就是说，f 拥有泛型类型。与此相反，在表达式 `(fun f. pair (f 3) (f true))(fun x. x)` —— 即以上 `let` 表达式的函数应用版本中，我们无法为 `(fun f. pair (f 3) (f true))(fun x. x)` 推导出一个浅显类型。

如果一个变量没有在外围的 lambda 绑定器中出现，那么这个变量就是泛型的。这些绑定器必须被加入假设集合中，这样它们就能在之后创建这些外围 lambda 的时候被 *ABS* 丢弃。因此，如果一个变量没有出现在假设集合中，那么它就是泛型的。也正因如此，如果一个变量是泛型的，我们可以对其应用 *GEN* 规则来给它添加一个量词。这确定了泛型变量和量词之间的准确关系。

有一种形式化的方式来将上述推导系统和前面小节中的类型检查算法联系起来。显然，如果类型检查算法成功地得出了一个表达式的类型，那么这个类型也可以从推理系统中被推导出来（见 [Milner 78]）。我们会采取一种不同的、非形式化的方式来直观地证明类型检查算法。我们将会展示如何从一个推理系统中抽取出类型检查算法。从这个视角来看，类型检查算法就是一个证明启发式（*proof heuristic*），也就是说它是一种确定推理规则的应用次序的策略。如果证明启发式成功了，我们就知道类型可以被推导出来。不过如果证明启发式失败了，还是有可能推导出一个类型的。例如，我们的启发式无法应用于需要非浅显类型变换（non-shallow type manipulation）的表达式，例如它推不出 `(fun x. x x)(fun x. x)`：$\forall\alpha.\alpha \to \alpha$。

这个启发式有两个部分：如何确定假设集合，以及应用推理规则的次序。如果语言要求所有的标识符都有类型声明，获得假设集合的过程就显而易见了；否则，我们就需要进行类型推断（type inference）。

在进行类型推断时，首先将 lambda 绑定的标识符与类型变量关联，随后在类型检查中不断收集信息，以确定这些标识符应该到底是什么类型。也就是说，我们从最宽泛的假设出发，并按某种次序应用推理规则以构建证明。某些规则要求两个子表达式具有相同的类型。通常情况并非如此，因此我们归一化它们各自的类型，使得它们相等。归一化的过程会导致某些标识符的类型被特化出来。此时我们可以想象重复证明步骤，但此时我们已经有了一个更详细的假设集合：这次两个子表达式就会有相同的类型，我们就能继续类型检查了。

应用推理规则的顺序应该能让我们自左至右，自下而上地构建我们正在进行类型检查的表达式。例如，之前我们想要揭示 `(fun x. x)`：$\forall\alpha.\alpha \to \alpha$。我们首先将 $x : \alpha$ 作为我们的假设集合，要自下而上地推导出 `(fun x. x)` 的类型，我们就从 x 的类型开始，x 的类型可以由 *VAR* 取得，然后我们就可以运用 *ABS* 得到 `(fun x. x)` 的类型。

那么，目前为止，除了 *GEN* 和 *SPEC* 规则之外，所有其他规则都可以被从左到右，自下而上地被应用，每次只根据接下来的语法结构应用一条规则。这样，问题的范围就缩小到决定何时运用 *GEN* 和 *SPEC* 了；我们和把它们跟 *LET* 规则放在一起讨论。

在应用 *LET* 之前，我们先派生出 $A \vdash e' : \sigma'$（参见 *LET* 规则），然后再尽可能地应用 *GEN* 规则，得到 $A \vdash e' : \sigma$，其中 σ 可以是量化类型。现在我们可以开始派生 $A.x : \sigma \vdash e : \tau$，并且每当我们需要用到 *VAR* 规则而 x 和 τ 是量化类型时，我们马上使用 *SPEC* 类型移除所有量词，将量词变量替换为全新的类型变量。然后，如上所述，这些新变量将被实例化，从而确定使用 *SPEC* 规则的更精细的方式。

作为练习，你可以试着利用上述启发式方法来推导 **length** 的类型，并观察这个过程如何与类型检查算法相对应，特别要注意非泛型变量列表如何与假设集合以及 *GEN* 和 *SPEC* 规则的应用相对应。

8. 程序（ML）

以下 ML 程序实现了多态类型检查算法，也诠释了多态是如何在 ML 语言中应用的。ML 的语法和语义在 [Milner 84] 中有所描述。以下是对程序和 ML 语言的一些注解：

~~关键字以粗体表示，标识符以正体表示，而类型构造器以斜体表示。~~类型构造器在表达式中被用于创建数据，并在模式匹配中被用于分析并选择数据分量。类型变量~~以正体表示~~并且以单引号开头，例如 `'a` 是一个类型变量，通常读作“阿尔法”。

- 程序以标准列表操作函数开始，这些函数通过模式匹配来定义；注意这里没有使用类型声明。
- 指针，也就是可以赋值的对值的引用，并没有内建到 ML 语言中。指针可以用 `ref`（内建的可变引用）和 `Option` 实现。一个指针就是一个可写的引用，指向一个可选的值；如果值不存

在（none），则指针为空指针，否则为非空指针（some）。`Void` 创建一个新的空指针，`Access` 解引用指针，而 `Assign` 更新指针。类型 `'a Pointer` 是参数化的，不过在本程序中它只会被用于 `Type Pointer`。

- 时间戳被用于区分不同的类型变量。时间戳是一种抽象类型（因而无法被伪造），内部有一个变量 `Counter`，每次创建新时间戳都会自增这个变量。
- 类型 `Ide`、`Term` 和 `Decl` 从我们语言的抽象语法演变而来。一个类型表达式 `Type` 可以是一个类型变量或一个类型算子。一个类型变量使用一个时间戳来标识。如果类型变量的类型指针为空，则它是未实例化的类型变量，否则就是已实例化的类型变量。一个类型算子（例如 **bool** 或者 $\rightarrow$）具有一个名字和一个类型参数列表（例如 **bool** 没有参数，而 $\rightarrow$ 应有两个参数）。
- 当审视一个类型表达式时，函数 `Prune` 会被调用：它总是返回一个类型表达式，其要么是一个未初始化的类型变量，要么是一个类型表达式。也就是说，它会跳过已经实例化的变量，并且从表达式中将它们修剪掉，从而移除由已实例化变量构成的长链。
- 函数 `OccursInType` 用于检查一个类型变量是否曾在一个类型表达式中出现过。
- 类型 `NGVars` 是非泛型变量列表的类型。函数 `FreshType` 会创建一个类型表达式的拷贝，复制其中的泛型变量，并共享其中非泛型的类型变量。
- 类型归一化可以被很简单地定义。将非泛型变量与项目归一化时，项目中所有变量都会变为非泛型的。这已经由非泛型变量列表自动处理了，因此归一化过程中不需要添加特殊的代码。
- 随后类型环境也被定义。注意函数 `RetrieveTypeEnv` 总是创建新的类型；一些拷贝是不必要的，并且可以被消除。
- 最后就是我们的类型检查过程，它维护一个类型环境和一个非泛型变量列表。递归声明的分析被分解为两个阶段进行。第一个阶段 `AnalyzeRecDeclBind` 简单地创建一个新的非泛型类型变量集合，并将它们和标识符关联起来。第二个阶段 `AnalyzeRecDecl` 分析声明并调用 `UnifyType` 来确保递归类型约束。

```
{# ------ 列表操作（标准库函数） ------ #}

val rec
  map f nil = nil |
  map f (head :: tail) = (f head) :: (map f tail);

val rec
  fold f nil x = x |
  fold f (head :: tail) x = f (head, fold f tail x);

val rec
  exists p nil = false |
  exists p (head :: tail) = if p head then true else exists p tail;

{# ------ Option 类型 ------ #}

type 'a Option = data none | some of 'a;

{# ------ 指针类型 ------ #}

type 'a Pointer = data pointer of 'a Option ref;

val Void() = pointer(ref none);
val Access(pointer(ref(some V))) = V;
val Assign(pointer P, V) = P := some V;
```

```
{# ------ 时间戳 ------ #}

abstype Stamp = data stamp of int;
with local Counter = ref 0
  in val NewStamp() = (Counter := !Counter + 1; stamp(!Counter));
     val SameStamp(stamp S, stamp S′) = (S = S′)
  end
end;

{# ------ 标识符 ------ #}

type Ide = data symbol of String;

{# ------ 表达式 ------ #}

type rec Term = data
  ide of Ide |
  cond of Term * Term * Term |
  lamb of Ide * Term |
  appl of Term * Term |
  block of Decl * Term

and Decl = data
  defDecl of Ide * Term |
  andDecl of Decl * Decl |
  recDecl of Decl;
```

```
{# ------ 类型 ------ #}

type rec Type = data
  var of Stamp * Type Pointer |
  oper of Ide * Type list;

val NewTypeVar() = var(NewStamp(), Void());
val NewTypeOper(Name, Args) = oper(Name, Args);

val SameVar(var(Stamp, _), var(Stamp′, _)) =
  SameStamp(Stamp, Stamp′)

val rec Prune (Type: Type): Type =
  case Type of
    var (_, Instance) =>
      (case Instance of
        pointer(ref none). Type |
        pointer(_).
          let val Pruned = Prune(Access Instance)
          in (Assign(Instance, Pruned); Pruned) end
      ) |
    oper(_) => Type;

val rec OccursInType(TypeVar: Type, Type: Type): bool =
  let val Type = Prune Type
  in case Type of
    var(_) => SameVar(TypeVar, Type) |
    oper(Name, Args) =>
      fold(fun Arg, Accum => OccursInType(TypeVar, Arg) orelse Accum) Args false
    end;

val OccursInTypeList(TypeVar: Type, TypeList: Type list): bool =
  exists (fun Type => OccursInType(TypeVar, Type)) TypeList;

{# ------ 泛型变量 ------ #}

type NGVars = data nonGenericVars of Type list;

val EmptyNGVars = nonGenericVars [];

val ExtendNGVars(Type: Type, nonGenericVars NGVars): NGVars =
  nonGenericVars(Type :: NGVars);

val Generic(TypeVar: Type, nonGenericVars NGVars): bool =
  not(OccursInTypeList(TypeVars, NGVars));
```

```
{# ------ 复制类型 ------ #}

type CopyEnv = (Type * Type) list;

val FreshType(Type: Type, NGVars: NGVars): Type =
  let val rec Fresh (Type: Type, Env: CopyEnv ref): Type =
    let val Type = Prune Type
    in case Type of
      var(_) =>
        if Generic(Type, NGVars) then FreshVar(Type, !Env, Env) else Type |
      oper(Name, Args) =>
        NewTypeOper(Name, map (fun Arg => Fresh(Arg, Env)) Args)
    end
  and FreshVar (Var: Type, Scan: CopyEnv, Env: CopyEnv ref): Type =
    if null Scan
    then let val NewVar = NewTypeVar()
         in (Env := (Var, NewVar)::(!Env); NewVar) end
    else let val (OldVar, NewVar)::Rest = Scan
         in if SameVar(Var, OldVar) then NewVar else FreshVar(Var, Rest, Env) end
  in Fresh(Type, ref []) end;

{# ------ 原语算子 ------ #}

val BoolType =
  NewTypeOper(symbol "bool", []);

val FunType (From: Type, Into: Type): Type =
  NewTypeOper(symbol "fun", [From; Into])

{# ------ 归一化 ------ #}

val rec UnifyType (Type: Type, Type′: Type): unit =
  let rec Type = Prune Type and Type′ = Prune Type′
  in case Type of
    var (Stamp, Instance) =>
      if OccursInType(Type, Type′)
      then case Type′ of var(_) => () | oper(_) => escape "Unify"
      else Assign(Instance, Type′) |
    oper(Name, Args) =>
      case Type′ of
        var(_) => UnifyType(Type′, Type) |
        oper(Name′, Args′) =>
          if Name = Name′ then UnifyArgs(Args, Args′) else escape "Unify"
  end

and UnifyArgs ([], []) = () |
    UnifyArgs (Hd::Tl, Hd′::Tl′) = (UnifyType(Hd, Hd′); UnifyArgs(Tl, Tl′)) |
    UnifyArgs (_) = escape "Unify";
```

```
{# ------ 环境 ------ #}

type TypeEnv = data TypeEnv of (Ide * Type) list;

val EmptyTypeEnv = typeEnv [];

val ExtendTypeEnv (Bind: Ide, Type: Type, typeEnv TypeEnv) =
  typeEnv((Bind, Type)::TypeEnv);

val RetrieveTypeEnv (Ide: Ide, typeEnv TypeEnv, NGVars: NGVars): Type =
  let val rec
    Retrieve ([]: (Ide * Type) list): Type = escape "Undefined identifier" |
    Retrieve ((Bind, Type)::Rest: (Ide * Type) list): Type =
      if Ide = Bind then FreshType(Type, NGVars) else Retrieve Rest
  in Retrieve TypeEnv end;
```

```
{# ------ 类型检查 ------ #}

val rec
  AnalyzeTerm (ide Ide, TypeEnv, NGVars): Type =
    RetrieveTypeEnv(Ide, TypeEnv, NGVars) |
  AnalyzeTerm (cond(If, Then, Else), TypeEnv, NGVars): Type =
    let val () = UnifyType(AnalyzeTerm(If, TypeEnv, NGVars), BoolType);
        val TypeOfThen = AnalyzeTerm(Then, TypeEnv, NGVars);
        val TypeOfElse = AnalyzeTerm(Else, TypeEnv, NGVars)
    in (UnifyType(TypeOfThen, TypeOfElse); TypeOfThen) end |
  AnalyzeTerm(lamb(Bind, Body), TypeEnv, NGVars): Type =
    let val TypeOfBind = NewTypeVar();
        val BodyTypeEnv = ExtendTypeEnv(Bind, TypeOfBind, NGVars);
        val BodyNGVars = ExtendNGVars(TypeOfBind, NGVars);
        val TypeOfBody = AnalyzeTerm(Body, BodyTypeEnv, BodyNGVars)
    in FunType(TypeOfBind, TypeOfBody) end |
  AnalyzeTerm(appl(Fun, Arg), TypeEnv, NGVars): Type =
    let val TypeOfFun = AnalyzeTerm(Fun, TypeEnv, NGVars);
        val TypeOfArg = AnalyzeTerm(Arg, TypeEnv, NGVars);
        val TypeOfRes = NewTypeVar()
    in (UnifyType(TypeOfFun, FunType(TypeOfArg, TypeOfres)); TypeOfRes) end |
  AnalyzeTerm (block(Decl, Scope), TypeEnv, NGVars): Type =
    let val DeclEnv = AnalyzeDecl(Decl, TypeEnv, NGVars)
    in AnalyzeTerm(Scope, DeclEnv, NGVars) end

and
  AnalyzeDecl (defDecl(Bind, Term), TypeEnv, NGVars): TypeEnv =
    ExtendTypeEnv(Bind, AnalyzeTerm(Term, TermEnv, NGVars), TypeEnv) |
  AnalyzeDecl (andDecl(Left, Right), TypeEnv, NGVars): TypeEnv =
    AnalyzeDecl(Right, AnalyzeDecl(Left, TypeEnv, NGVars), NGVars) |
  AnalyzeDecl (recDecl Rec, TypeEnv, NGVars): TypeEnv =
    let val TypeEnv, NGVars = AnalyzeRecDeclBind(Rec, TypeEnv, NGVars)
    in AnalyzeRecDecl(Rec, TypeEnv, NGVars) end

and
  AnalyzeRecDeclBind(defDecl(Bind, Term), TypeEnv, NGVars): TypeEnv * NGVars =
    let val Var = NewTypeVar()
    in ExtendTypeEnv(Bind, Var, TypeEnv), ExtendNGVars(Var, NGVars) end |
  AnalyzeRecDeclBind(andDecl(Left, Right), TypeEnv, NGVars): TypeEnv * NGVars =
    let val TypeEnv, NGVars = AnalyzeRecDeclBind(Left, TypeEnv, NGVars)
    in AnalyzeRecDeclBind(Right, TypeEnv, NGVars) end |
  AnalyzeRecDeclBind(recDecl Rec, TypeEnv, NGVars): TypeEnv * NGVars =
    AnalyzeRecDeclBind(Rec, TypeEnv, NGVars)

and
  AnalyzeRecDecl(defDecl(Bind, Term), TypeEnv, NGVars): TypeEnv =
    (UnifyType(RetrieveTypeEnv(Bind, TypeEnv, NGVars),
               AnalyzeTerm(Term, TypeEnv, NGVars));
     TypeEnv) |
  AnalyzeRecDecl(andDecl(Left, Right), TypeEnv, NGVars): TypeEnv =
    AnalyzeRecDecl(Right, AnalyzeRecDecl(Left, TypeEnv, NGVars), NGVars) |
  AnalyzeRecDecl(recDecl Rec, TypeEnv, NGVars): TypeEnv =
    AnalyzeRecDecl(Rec, TypeEnv, NGVars);
```

9. 结论

本文给出了一些关于多态类型检查的实用知识，尝试用非形式化的方式将其与理论背景联系起来。这些思想已经在一些人之间传播了些许年月，并且通过我和 Luis Damas, Mike Gordon, Dave MacQueen, Robin Miller，Rave Sethi 的讨论传给了我。

参考文献

[Burstall 80] R.Burstall, D.MacQueen, D.Sannella: “Hope: an Experimental Applicative Language”, Conference Record of the 1980 LISP Conference, Stanford, August 1980, pp. 136-143.

[Damas 82] L.Damas, R.Milner: “Principal type-schemes for functional programs”, Proc. POPL 82, pp. 207-212.

[Gordon 79] M.J.Gordon, R.Milner, C.P.Wadsworth: “Edinburgh LCF”, Lecture Notes in Computer Science, No. 78, Springer-Verlag, 1979.

[Hindley 69] R.Hindley: “The principal type scheme of an object in combinatory logic”, Transactions of the American Mathematical Society, Vol. 146, Dec 1969, pp. 29-60.

[MacQueen 84] D.B.MacQueen, R.Sethi, G.D.Plotkin: “An ideal model for recursive polymorphic types”, Proc. Popl 84.

[Milner 78] R.Milner: “A theory of type polymorphism in programming”, Journal of Computer and System Sciences, No. 17, 1978.

[Milner 84] R.Milner: “A proposal for Standard ML”, Proc. of the 1984 ACM Symposium on Lisp and Functional Programming, Aug 1984.

[Robinson 65] J.A.Robinson: “A machine-oriented logic based on the resolution principle”, Journal of the ACM, Vol 12, No. 1, Jan 1965, pp. 23-49.

[Scott 76] D.S.Scott: “Data types as lattices”. SIAM Journal of Computing, 4, 1976.