如何阅读类型系统符号

Alexis King
原作者

Chuigda Whitegive
翻译
Doki Doki λ Club!

CAIMEO
翻译提议、校对

Gemini
校对
Google Deepmind

Claude
校对
Anthropic

译者前言

本文是 StackExchange 提问及回答 How should I read type system notation 的中文翻译，部分字句有所改动。译者还增补了几个常用的符号。术语 *(terminology)* 在正文中第一次出现的地方以仿宋体（中文）或 *Italic (English)* 呈现。如遇翻译或排版质量问题，请在 https://github.com/chuigda/CoreBlogPHP-NG/issues 向译者报告。

前言

用于描述类型系统的符号因不同的表示方法而异，因此不可能给出全面的概述。不过，大多数表示方法之间都有许多共通之处，本回答会尝试提供足够的基础知识，以便理解常见的各种变体。

语法和文法

应用于程序设计语言的类型系统是语法性 *(syntactic)* 系统，也就是说，类型系统是定义在程序设计语言（抽象）语法之上的一组规则。因此，对类型系统的全面论述首先会使用巴科斯-瑙尔表示法 *(Backus Naur Form, BNF)*[1] 给出类型系统所考虑的所有语法构造 *(syntactic construct)* 的文法 *(grammar)*。在最简单的类型化语言中，语法仅用于两件事：表达式和类型。

例如，考虑如下只包括布尔和整数的简单语言的文法：

$$
\begin{array}{lll}
e ::= \text{true} \mid \text{false} & & \text{布尔字面量} \\
\quad \mid 0 \mid 1 \mid -1 \mid 2 \mid -2 & & \text{整数字面量} \\
\quad \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & & \text{条件} \\
\quad \mid e + e \mid e - e \mid e \times e & & \text{算术} \\
\quad \mid e = e \mid e < e \mid e > e & & \text{比较} \\
\\
\tau ::= \text{Bool} & & \text{布尔} \\
\quad \mid \text{Int} & & \text{整数}
\end{array}
$$

这里，e 对应于表达式，而 τ 对应于类型，这是标准的符号约定。有的表示方法会为类型选用不同的符号，如 t, T, σ 或者其他小写希腊字母，但总体结构大致相同。

更复杂的语言自然会有更复杂的文法：命令式语言需要语句 *(statement)* 的文法，支持模式匹配的语言需要模式的文法，诸如此类。而上面这个简单语言里甚至连变量都没有！然而，在文法上区分词项 *(term，具有类型的东西)* 和类型是必要的，因为类型系统正是要定义词项和类型间的关系。

[1] 译注：BNF 不是一种范式 / 正规形式 *(Normal Form)*。

关系、判断、公理和推理规则

指定文法之后，下一步就是定义类型关系 *(typing relation)*。类型关系通常写作 $e : \tau$，可以读作“e 的类型是 τ”。直觉上，一些陈述是“有意义”的，另一些则没有：

- $1 + 2 : \text{Int}$ 表示“$1 + 2$ 的类型是 Int”，这当然是有意义的。
- $1 + 2 : \text{Bool}$ 表示“$1 + 2$ 的类型是 Bool”，而这是没有意义的。
- $\text{true} + 2 : \text{Int}$ 表示“$\text{true} + 2$ 的类型是 Int”，这更没有意义，因为 $\text{true} + 2$ 本就是无稽之谈，根本不该有类型。

我们应该写一些能准确捕捉“哪些陈述有意义，哪些没有”直觉的规则。为此，我们定义类型判断 *(typing judgement)*，写作：

$$\vdash e : \tau$$

这里，$\vdash$ 可以读作“以下陈述是正确的”。这里的 $\vdash$ 可能显得有点多余，并且在简单的类型系统（比如现在我们讨论的这个）当中也确实可以省略，但它接下来会发挥更重要的作用。

用这种符号，我们就可以为类型系统写一些类型规则 *(typing rule)*了：

$$\frac{}{\vdash \text{true} : \text{Bool}} \qquad \frac{}{\vdash \text{false} : \text{Bool}}$$

这两条规则上面都有一条横线，而横线上空无一物，这表示它们恒成立，也就是说它们是公理 *(axiom)*。对于整数字面量，也可以有无数这样的公理：

$$\frac{}{\vdash 0 : \text{Int}} \qquad \frac{}{\vdash 1 : \text{Int}} \qquad \frac{}{\vdash -1 : \text{Int}} \qquad \frac{}{\vdash 2 : \text{Int}} \qquad \cdots$$

当然，字面量的类型规则相当无聊。但当表达式里包含子表达式的时候，事情就变得有意思了！以下是 $+$ 和 $<$ 的类型规则：

$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \qquad \frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 < e_2 : \text{Bool}}$$

现在，横线上下都有东西了，它们就成为了推理规则 *(inference rule)*。它们表示有条件的类型规则：**如果**横线上的所有陈述成立，**则**横线下面的陈述成立。例如，第一条规则可以读作“如果 $e_1 : \text{Int}$ 成立且 $e_2 : \text{Int}$ 成立，则 $e_1 + e_2 : \text{Int}$ 成立”，希望这符合直觉。

$-$，$\times$，$=$ 和 $>$ 的规则和上面的两条规则大致相同。但 `if` ... `then` ... `else` 的规则要再复杂一点。这是因为 `if` 表达式的两个分支可以具有任何类型，只要两个分支的类型相同。也就是说，

$$\texttt{if}\ \text{true}\ \texttt{then}\ 1\ \texttt{else}\ 2$$

和

$$\texttt{if}\ \text{true}\ \texttt{then}\ \text{false}\ \texttt{else}\ \text{true}$$

是合法的，但

$$\texttt{if}\ \text{true}\ \texttt{then}\ 1\ \texttt{else}\ \text{true}$$

是不合法的。

要描述这一点，类型规则用一个变量来表示分支的类型：

$$\frac{\begin{array}{c}\vdash e_1 : \text{Bool} \\ \vdash e_2 : \tau \\ \vdash e_3 : \tau\end{array}}{\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

应用这条规则时可以为 τ 选取任何类型，只要它在两个分支之间保持一致就行。

这种符号起源于形式逻辑 *(formal logic)*，特别地，用于类型系统的符号风格上与自然演绎 *(natural deduction)* 最为接近。虽然我不会在本答案中详细介绍符号的具体细节，但以这种形式表达的规则可以被用来构建关于系统属性的形式化证明，这对于证明类型健全性 *(type soundness)* 之类的属性非常重要。

作为算法规范的判断

到目前为止，我一直在刻意避免谈及类型判断的计算解释。一般而言，判断只是逻辑规则，而某些以这种方式指定的类型系统并不直接对应于可判定的 *(decidable)* 类型检查算法。然而，如果你不习惯思考证明系统，这种纯逻辑的视角可能不太直观。

幸运的是，很多时候，你都可以用一种方法照着类型规则写出类型检查算法：规则 $\vdash e : \tau$ 可以解释成一个从表达式 e 到其类型 τ 的函数 *(function)*。通常表达式文法中的每种情况都有相应的一条规则，这样整个类型规则就能转写成一个递归的类型检查函数，每条规则对应于这个递归函数中的一条分支。

例如，考虑以上小小语言的规则。它直接对应于一个这样的 `infer` 函数：

$$\begin{array}{l}
\text{infer} : \text{Expr} \to \text{Type} \\
\text{infer}(e) = \texttt{match}\ e\ \texttt{where} \\
\quad \begin{array}{lll}
\text{true} \mid \text{false} & \mapsto & \text{Bool} \\
0 \mid 1 \mid -1 \mid 2 \mid \ldots & \mapsto & \text{Int} \\
e_1 + e_2 & \mapsto & \texttt{assert}\ \text{infer}(e_1) = \text{Int}; \\
 & & \quad \texttt{assert}\ \text{infer}(e_2) = \text{Int}; \\
 & & \quad \text{Int} \\
e_1 < e_2 & \mapsto & \texttt{assert}\ \text{infer}(e_1) = \text{Int}; \\
 & & \quad \texttt{assert}\ \text{infer}(e_2) = \text{Int}; \\
 & & \quad \text{Bool} \\
\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \mapsto & \texttt{assert}\ \text{infer}(e_1) = \text{Bool}; \\
 & & \quad \texttt{let}\ \tau = \text{infer}(e_2); \\
 & & \quad \texttt{assert}\ \text{infer}(e_3) = \tau; \\
 & & \quad \tau
\end{array}
\end{array}$$

即使无法将类型规则直接转换为类型检查算法，在对逻辑判断进行推理时考虑信息流仍然非常有用：对于判断 $\vdash e : \tau$，e 可以被视为判断的“输入”，而 τ 可以被视为“输出”。这种严格的方向性并不总适用于类型系统中的每条规则，但通常它适用于多数规则，并且是理解规则含义的一种有效方法。

变量和语境

目前为止，我们用作例子的语言异常地简单。此前我一直在有意规避变量 *(variable)* 这一复杂性，但若要为任何有用的程序设计语言编写类型规则，那么就不能逃避。所以接下来让我们扩展这个小小语言，向其中加入函数，使其成为简单类型 λ 演算 *(Simply-typed lambda calculus, STLC)* 的一个变体。这需要向语言的文法中添加如下内容：

$$
\begin{array}{rll}
e ::= & \ldots & \\
& \mid x & \text{变量} \\
& \mid \lambda x : \tau \,.\, e & \text{函数抽象} \\
& \mid e\ e & \text{函数应用} \\
\tau ::= & \ldots & \\
& \mid \tau \to \tau & \text{函数类型}
\end{array}
$$

这里 x 代表“某个变量”。如果你不熟悉 λ 演算，这些符号可能看起来有点怪，但它其实没有看上去那么陌生：简单类型 λ 演算中的语法 $\lambda x : \tau \,.\, e$ 直接对应于 TypeScript 中的 `(x : τ) => e`，而 $f\ x$ 则对应于 `f(x)`。

在扩充文法之后，类型关系所用的符号不需要改变——仍然形如 $e : \tau$。但是，类型判断的结构必须相应地进行扩展。我们会在为变量编写类型规则时遇到麻烦：

$$\frac{}{\vdash x : ???}$$

问题在于，变量的类型取决于它出现时所处的语境 *(context)*。因此类型判断需要相应地扩展，以追踪作用域内所有变量的类型，我们用使用以下符号：

$$\Gamma \vdash e : \tau$$

Γ 被称为“语境”或者“类型环境”，而 $\vdash$ 所扮演的角色现在更明确了：它把语境假设 *(contextual assumptions)* 从待证陈述里分离了出去。因此，扩展后的判断可以读作“在语境 Γ 中，表达式 e 具有类型 τ”，而在算法上，Γ 可以被视为判断的一个额外的“输入”，具有类型 `Map<Variable, Type>`。然而，正式地说，任何类型规则都必须被语法性地定义。类型规则中的语境可以显式地定义为如下的语法结构：

$$
\begin{array}{rll}
\Gamma ::= & \varnothing & \text{空语境} \\
& \mid \Gamma, x : \tau & \text{变量绑定}
\end{array}
$$

有时，空语境会使用符号 • 而不是 $\varnothing$。

在这种表示方法下，语境本质上是一个关联列表 *(association list)*，它将变量名映射到其类型。

大部分类型规则都无需关心语境：大部分推理规则只是简单地传递语境，不作任何更改；而大部分公理也同样无视语境。例如，以下是几条根据新的判断调整的类型规则：

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \qquad \frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}}$$

然而，语境对于两种新增结构——变量使用和 λ 表达式——的类型规则是必须的：

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 \,.\, e) : \tau_1 \to \tau_2}$$

右边的那条规则是最复杂的，因为它包含了所有关键的机制：在对 λ 表达式的函数体 *(body)* 进行类型检查时，语境会被扩展，新的绑定 $x : \tau_1$ 会被引入。之后在检查表达式 e 的过程中，这一

信息会被左边的规则使用，这条规则实质上是说，若当前语境中存在**变量绑定** x，其类型为 τ（因此 x 在作用域内），则**表达式** x 具有类型 τ。换句话说，语境是这两条规则之间用来传递信息的一种通信机制。

（细心的读者可能会注意到，这一模型无法处理变量遮蔽。这是因为以这种方式指定的类型系统通常假定所有变量都已被解析并唯一。）

如果你仍然感到困惑，不妨考虑一下这些新增内容对前文提到的 infer 函数的影响：

$$
\begin{aligned}
&\text{infer} : (\text{Context}, \text{Expr}) \to \text{Type} \\
&\text{infer}(\Gamma, e) = \texttt{match } e \texttt{ where} \\
&\quad x && \mapsto \text{lookup}(\Gamma, x) \\
&\quad \lambda x : \tau_1 \,.\, e' && \mapsto \texttt{let } \Gamma' = \text{extend}(\Gamma, x, \tau_1); \\
&\qquad && \quad \texttt{let } \tau_2 = \text{infer}(\Gamma', e'); \\
&\qquad && \quad \tau_1 \to \tau_2
\end{aligned}
$$

接下来还要添加一条函数应用的类型规则：

$$
\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}
$$

其他常见符号和注意事项

目前为止，本答案已经描述了绝大多数用于描述类型系统的符号，但对符号的修改和扩展极其常见。要涵盖所有修改和扩展是不可能的，但幸运的是，优秀的论文通常会先解释它们引入的任何非标准符号。然而，有些约定俗成的惯例十分普遍，以至于人们常常不加解释就使用它们，本节会尝试提供一个基本概述，并描述一些符号上的怪癖。

这不是一个详尽的列表，也永远不可能是。如果你发现此处未涵盖某些符号，请另行提问！

推理规则的布局

目前为止，本答案中所有的例子都以非常规整的垂直方式列出了推理规则。然而，“一个条件一行”并不是某种强制性的要求，多个条件可以并排出现在同一行上：

$$
\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 + e_2 : \text{Int}}
$$

在同一条规则中，垂直排列和水平排列甚至可能同时出现。

附加条件

通常，推理规则横线上面出现的条件必须是能通过公理和推理规则的某种组合来满足的判断。然而，事情并不总是如此：规则中也可能包含任意的布尔表达式，我们称之为*附加条件 (side conditions)*，只有所有附加条件都满足，才能应用推理规则。例如在上述类型规则中，$x : \tau \in \Gamma$ 就是一个附加条件。

算法性 (algorithmic) 的类型系统中有时会出现一种特殊的附加条件：α `fresh` 或者 $\text{fresh}(\alpha)$。这表示 α 是一个全新的类型变量，也就是它和既存的所有类型变量都不同。

子类型

子类型 (subtyping) 引入了一种比严格相等性更弱的类型一致性概念。子类型关系必须被显式定义，通常记作 $\tau_1 <: \tau_2$ 或者 $\tau_1 \preceq \tau_2$，可以读作“τ_1 是 τ_2 的子类型”。

子类型关系通常也使用推理规则的形式来定义。例如，一个非常简单的子类型关系可能引入两个特殊的类型，$\top$（读作"顶 / *top*"）和 $\bot$（读作"底 / *bottom*"）。$\top$ 是所有类型的*超类型 (supertype)*，而 $\bot$ 是所有类型的*子类型 (subtype)*。这一关系可以用以下三条简单公理表示：

$$\frac{}{\tau <: \tau} \qquad \frac{}{\tau <: \top} \qquad \frac{}{\bot <: \tau}$$

第一条规则是*自反性 (reflexive)* 规则，通常简写为"refl"，它声明每个类型都是自身的子类型。这条规则确保了子类型关系严格弱于全等关系。

然后，在所有允许子类型的规则中，都必须显式地使用如上定义的子类型关系。例如，支持子类型的系统可以用以下规则来实现函数应用：

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_3 \quad \Gamma \vdash e_2 : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_3}$$

多语境

有些类型系统所定义的类型判断涉及到多个语境。第二个语境通常被命名为 Δ。常用于表示多语境的符号是 $\Gamma; \Delta \vdash e : \tau$（当两个语境都是"输入"时）和 $\Gamma \vdash e : \tau \dashv \Delta$（当 Δ 是"输出"时）。

第二个语境可能有多种不同的用途。例如，某些变量可能在某些表达式中被引用，而其他表达式则不行；又或者，输出语境可在*资源感知型 (resource-aware)* 程序设计语言中被用于跟踪哪些变量被"消耗"了。

双向类型检查

双向类型检查 (Bidirectional typechecking) 是一种不依赖约束求解器的、有限的非局部类型推理技术。一个双向系统将通常的类型判断 $\Gamma \vdash e : \tau$ 分为两个特化的判断：

- $\Gamma \vdash e \Leftarrow \tau$（或作 $\Gamma \vdash e \downarrow \tau$，$\Gamma \vdash e :_{\downarrow} \tau$）是检查 *(checking)* 判断，它检查表达式 e 是否具有期望的类型 τ。算法上，τ 是判断的输入。
- $\Gamma \vdash e \Rightarrow \tau$（或作 $\Gamma \vdash e \uparrow \tau$，$\Gamma \vdash e :_{\uparrow} \tau$）是推导 *(inference)* 判断，在"不知道期望类型是什么"的时候使用。算法上，τ 是判断的输出。

两个判断以互递归的方式定义，双向传播类型信息，因此允许在某些时候省略类型注解。例如，λ 抽象的类型规则的检查变体允许省略变量绑定上的类型注解，因为变量的类型可以根据期望的类型确定：

$$\frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash (\lambda x\,.\,e) \Leftarrow \tau_1 \to \tau_2}$$