arXiv:2304.10500v1 [cs.PL] 15 Mar 2023

简单类型λ演算中的Transformer模型类型推断：深度学习代码的案例研究

BrandoMiranda *、Nathan Fulton、Avi Shinnar、
Vasily Pestun† 和 Barry Trager，IBM Research

2021年8月

摘要

尽管深度学习与形式语言交叉领域的研究成果日益丰富，但针对使用 Transformer 模型对类型化 λ 演算进行推理的系统性探索仍相对较少。这是一个值得研究的领域，原因有二。首先，类型化 λ 演算是编程语言的通用语。一套能够将各种类型化 λ 演算与有效神经网络架构联系起来的启发式方法，将为语言特性（例如，多态性、子类型化、继承等）与架构选择之间的映射提供一种系统性方法。其次，Transformer 模型被广泛应用于面向代码的深度学习架构中，但其设计和超参数空间十分庞大，而且在编程语言应用领域尚未得到充分探索。因此，我们提出一种基准测试，使我们能够通过编程语言中或许最简单、最基础的特性——项与类型之间的关系——来对这一问题进行深入探索。因此，我们通过研究 Transformer 预热和优化器选择对类型推断任务的影响，开启了对用于类型化 λ 演算的 Transformer 架构的探索：即仅使用 Transformer 来预测 λ 演算项的类型。我们发现，即使在这种简单的设置下，优化景观也颇具挑战性。一项特别的实验发现是，与 Adam 和 RAdam 相比，使用 Adafactor 进行优化能够更快地收敛。我们推测，优化器之间这种不同的性能表现可能与在形式生成的数据集上进行泛化所面临的困难有关。

*同时也隶属于伊利诺伊大学厄巴纳-香槟分校。
†同时隶属于 IHES。

1 引言

深度学习在计算机视觉 [1, 2], 自然语言处理 [3, 4], 尤其是博弈领域 [5, 6, 7] 取得的成功，推动了越来越多关于将深度学习应用于编程任务的研究。深度学习与编程语言交叉领域的研究包括：利用神经网络解决编程语言研究中的经典问题（神经引导证明搜索 [8], 神经演绎式程序合成 [9], 和神经归纳式程序合成 [10]）、利用编程语言解决深度学习在典型应用领域中的弱点，以及将编程语言与深度学习相结合以解决新颖任务（例如，NLP 到代码[11]）。

尽管深度学习与形式语言交叉领域的研究日益增多，但针对 Transformer 模型如何推理类型化 λ 演算的系统性探索仍相对有限。这个研究方向之所以引人关注，有两个原因。首先，类型化 λ 演算是编程语言的通用语。若能建立一套启发式方法，将各种类型化 λ 演算与有效的神经网络架构联系起来，就能为如何将语言特性（例如多态、子类型化、继承等）映射到架构选择提供系统方法。其次，Transformer 模型广泛应用于面向代码的深度学习架构，但 Transformer 的设计空间和超参数空间极其庞大，在编程语言应用中也鲜有探索。

本文通过探索 Transformer 预热 [12, 13, 14, 15]和优化器选择对学习类型推断的影响，初步研究了适用于简单类型化 λ 演算的 Transformer 架构。这一基准测试使我们能够在简单的设定下，探索 Transformer 推理编程语言中或许最简单、最基础的性质的能力：项与类型之间的关系。

我们的贡献如下:

- 我们针对固定的 Transformer 架构，探索了完整范围的超参数。我们发现，学习率 而非预热步数，是最重要的超参数。这一发现意义重大，因为当前文献高度强调学习率预热计划的重要性 [12, 13, 14, 15, 16]。

- 我们证明，尽管任务很简单，要达到令人满意的性能，仍需要进行大量的超参数搜索。
- 我们发现，Adafactor 优化器 [16] 在无需进行任何超参数调优的情况下，就能在我们的任务中始终实现零训练误差和零验证误差。无论是使用公式的复杂路径嵌入（类似于自然语言的字符嵌入），还是使用深度嵌入 [17]，这一结论均成立。

  鉴于 Adafactor 为我们的训练带来了令人瞩目的稳定性，我们推测，训练基于 Transformer 的模型以学习

形式规则——例如，先前的研究 [18] 首次通过“Grokking”观察到的现象——或许可以通过使用 Adafactor 优化器 [16] 来避免。因此，我们建议从业者和研究人员先尝试 Adafactor。

2 背景

本节回顾 Transformer 和简单类型λ演算。

2.1 Transformer

Transformer模型在自然语言处理（NLP）领域产生了巨大影响。它能够利用 GPT-n 系列等模型生成听起来与人类语言相似的文本 [4]。此外，Transformer模型已被用于生成代码，例如 GPT-3 [4], GitHub copilot [11], TreeGen [17], GPT-f[19], skip-trees [20] 等。在自然语言处理领域，一种常见的训练方式是将自然语言句子转换为由闭合词汇表中的词法单元组成的序列，然后向具有数百万量级参数的Transformer模型输入大量数据（例如，数百万个词元） [13]。在形式语言生成领域，Transformer通常通过自然语言生成代码 [11, 17] ，或直接基于形式语言进行训练 [19, 20, 21]。

2.2 简单类型化 λ 演算

简单类型λ演算是一种编程语言模型，能够捕捉函数定义和函数应用的本质。该语言同时包含项和类型。请注意，项也称为程序，但在本文中我们将使用“项”这一术语。

类型可以是由字母表 $t \in T$ 中的一个字母表示的基本类型，也可以是由从一个类型指向另一个类型的箭头表示的函数类型：

$$\tau ::= t \mid \tau \to \tau \tag{1}$$

简单类型λ演算的项包括变量（变量字母表中的变量 $x \in X$）、函数定义和函数应用：

$$e ::= x \mid \lambda x : \tau . e \mid e_1\ e_2$$

表达式 $\lambda x : \tau . e$ 定义了一个函数，该函数接受一个名为 x、类型为 τ 的参数； e 是该函数的函数体（当然，其中可以引用参数 x）。表达式 $e_1\ e_2$ 使用参数 e_2 调用函数 e_1 。

该语言可以通过添加基类型和基变量来丰富。例如，考虑扩展后的语言，其中包含 nat 基类型和基变量 $0, 1, 2, \ldots$。以下表达式定义了一个函数，该函数接受任意数字作为输入并返回数字 1，然后将此函数应用于数字 5：

$$(\lambda x : \texttt{nat}.1)5$$

通常，这类场景用包含变量及其类型的类型环境 Γ （也称为上下文）来表示。形式上，类型环境（或上下文）是一组配对 $x : \tau$，其中 x 是一个变量，τ 是它的类型。例如，如果执行开始时我们只有基变量，那么前一个示例的类型环境将是 $\Gamma = \{1 : \text{nat}, 2 : \text{nat} \ldots\ldots, \}$。传统上，类型判断表示如下：

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

这意味着项 e 在上下文 Γ中具有类型 τ。请注意，接下来我们将把类型环境简称为上下文。

2.3 类型推断

类型推断算法用于计算表达式的类型。类型推断在带类型的编程语言中发挥着重要作用，但随着语言表达能力的增强，类型推断也会变得更加困难；例如，OCaml 或 Scala 中的类型推断属于指数时间复杂度（但在实践中几乎总是可处理的） [22, 23]。在更具表达力的类型理论中，例如用于定理证明的类型理论，类型推断通常不可判定；即使在实践中，尽管广泛使用启发式方法，要将其自动化也很困难。

然而，简单类型化 λ演算中的类型推断非常简单。具体而言，其多项式时间完全性广为人知 [24] （即，可以使用图灵机在多项式时间内计算出类型）；第 9 节给出了证明其简单性的伪代码。这个问题之所以如此简单，正是我们假设简单类型 λ演算的类型推断是一个出色的少样本学习基准的原因。在编程语言课程中接触少量类型推断示例后，学生应该能够将所学推广到更长的项、具有不同基类型的项等。能够理解其所处理的形式语言模型的 Transformer，应该能够对简单类型 λ演算进行类型推断。

3 类型推断数据集

回想一下，我们训练一个 Transformer，旨在探索项与类型之间可能是最基本的关系：模型能否推断出一个简单类型λ演算项的类型？ （即学习如何进行类型推断）。具体而言，我们使用自己生成的合成数据集来训练 Transformer。在本节中，我们将结合一些具体的数据示例，介绍合成数据集生成的过程。

3.1 合成数据集生成

在生成简单类型化 λ 演算的合成数据集时，我们需要一个包含基变量和基类型的类型环境 Γ ，作为

如第 2.2 节所述。在我们的实验中，初始上下文中只有一个基类型为 T 的变量 x，即 ваз瓶变量 x，从而得到该上下文

$$\Gamma = \{\mathtt{x} : \mathtt{T}\} \tag{2}$$

数据集生成： 为了在数据集中生成一个样本，我们首先随机生成一个类型 τ，然后继续生成一个具有该类型的项 e。在简单类型化 λ 演算中，这一过程非常简单，因为只需检查该项的抽象语法树（AST）的构造器即可，但在一般情况下，这是不可判定的 [25]。

此外，我们还需要根据各项的抽象语法树深度对其进行界定。

随机类型生成： 输入：要生成类型，我们需要：来自全局上下文 Γ的有效基类型列表、任何类型允许的最大深度，以及用于决定何时生成基类型或进行分支并因此继续递归生成类型的概率分布。 输出： 类型列表。请注意，数据集可能包含重复类型，但我们不允许训练集、验证集和测试集之间存在重复类型的交集。

在递归生成一个类型时，我们首先抛掷一枚公平硬币，以决定是进行分支（从而继续递归地构建更深层次的项），还是返回一个基类型。如果抛硬币的结果表明应当进行分支并递归地继续构建树，我们首先检查是否已经达到最大深度；如果已经达到最大深度，则返回一个基类型，而不是继续递归构建更深层次的项。如果我们选择进行分支，则意味着我们将创建一个应用类型，并按照前面所述的方法，递归地生成箭头类型的左侧部分和右侧部分。需要注意的是，我们并不总是能够生成完整的二叉树，因为继续加深项或停止递归并为相应子树返回一个基类型，都具有一定的概率。

随机项生成： 输入： 为了生成一个随机项，我们接收以下内容作为输入：所需的类型 τ、（当前的）动态上下文 γ'（其初始值为全局上下文 Γ）、用于分支的概率分布，以及任意项所允许的最大深度。 输出： 我们输出一个具有该类型 τ 的单一项 e。为了生成一个项，我们首先检查其类型 τ 属于哪种类型，并据此（随机地）构建一个该类型的项。请注意，每当我们生成一个抽象时，都会确保将有界变量添加到上下文中，从而始终能够获取正确的上下文。我们将其称为（当前的）动态上下文 Γ'，其初始值等于包含基变量及其类型的全局上下文。我们的实现确保当嵌套抽象发生名称冲突时，绑定规则仍然正常适用。例如，如果存在两个名为 y 的变量，那么在 λ 项中，靠近根节点的抽象将被后来创建的、更深层的抽象覆盖。生成一个项时，需要考虑两种主要情况：

1. 如果给定的类型 τ 是一个基类型，或者
2. 如果 τ 是一个箭头类型 $\tau' \to \tau''$（用于抽象）。

在第一种情况下（当 τ 是基类型时），我们有两种选择，

a. 从当前动态上下文中选择一个具有该类型的变量来生成一个项 Γ'，或者

b.（递归地）生成一个应用，其形式为 $e_{left}(e_{right})$，并且在求值后最终会返回所需的类型 τ。为此，我们生成一个新的类型 $\tau' \to \tau$，用于 e_{left}，然后递归地生成一个具有此类型的 e_{left} 项。接着，我们递归地生成一个类型为 τ' 的项 e_{right}。有了这两个项后，我们返回一个形式为 $e_{left}(e_{right})$ 的应用项。对于情形 1，我们以相同的概率选择 1.a. 或 1.b.；如果在任何时候达到最大深度，就执行 1.a.，以避免项变得更深。

现在我们继续说明如何处理情况 2——当我们需要生成一个类型为 $\tau' \to \tau''$ 的项时。在这种情况下，生成此类类型的项时，我们有四种选择：

a. 我们可以在该类型的动态上下文 Γ' 中选择一个变量——前提是存在这样的变量（例如，在创建抽象时可能已将其添加进去）。

b. 我们选择创建一个具有该类型 $\tau' \to \tau''$ 的新抽象。

c. 我们选择创建一个应用，如果对其进行求值，最终将返回一个类型为 $\tau' \to \tau''$ 的项。

d. 项的当前深度已达到上限，因此我们选择返回动态上下文中的一个变量，或深度为 1 的抽象。

我们以相同的概率随机选择选项 2.a.、2.b. 或 2.c.。步骤 2.c. 的执行过程与步骤 1.b. 完全相同，不同之处在于返回的整体项的类型为 $\tau' \to \tau''$。

为了完成步骤 2.b. 中的抽象，我们需要创建一个绑定变量及其主体。创建绑定变量时，必须确保不重复使用基变量中的名称——如果发生这种情况，就相当于全局上下文中某个名称（例如 1）消失了，这将是致命的。除此之外，我们可以按照均匀概率，从所有这些有效名称中选择一个已使用过的字符串或一个新变量。为了创建其主体，我们只需按照本节概述的过程递归地创建一个项，其中输入类型为 τ''。

下面给出几个可能的数据点（$x,\ y$）的具体示例，其中 x 是项，而需要推断的目标类型是在上下文 $\Gamma = \{\mathtt{x} : \mathtt{T}\}$ 中的 y。示例 1：

$$(x, y) = (\mathtt{x}, \mathtt{T})$$

在此示例中，项是基变量 x，其基类型为 T。

示例 2：

$$(x, y) = (\mathtt{x}_0 : \mathtt{T}.\mathtt{x}, \mathtt{T} \to \mathtt{T})$$

此示例展示了一个具有绑定变量 $\mathtt{x}_0$ 和类型 T 的 λ 抽象。对于任意类型为 T 的项，该 λ 抽象都会返回基变量 x。因此，整体类型为 T →T。

示例 3：

$$(x, y) = ((\lambda.\mathtt{x}_0 : \mathtt{T}.\mathtt{x})\mathtt{x}, \mathtt{T})$$

在第二个示例中，这个项是将前一个示例中的 λ 抽象应用于基变量 x。由于该 λ 抽象对任何参数都会返回基变量 x，因此它总是具有 T。

示例 4:

$$(x, y) = (\lambda \mathtt{x}_0 : \mathtt{T} \to \mathtt{T}.\mathtt{x}, (\mathtt{T} \to \mathtt{T}) \to \mathtt{T})$$

此示例展示了一个 λ 抽象，其绑定变量 $\mathtt{x}_0$ 的类型为 T → T。特别要注意，参数明确是一个从基类型到基类型 T 的函数类型。由于主体始终返回基变量 x，因此结果类型是从输入类型 T →T 到输出类型 T。

示例 5:

$$(x, y) = ((\lambda \mathtt{x}_1 : \mathtt{T}.(\lambda \mathtt{x}_2 : \mathtt{T} \to \mathtt{T}.\mathtt{x}_2)\mathtt{x})(\lambda \mathtt{x}_0 : \mathtt{T}.\mathtt{x}_0), \mathtt{T} \to \mathtt{T})$$

这是一个稍微复杂一些的数据示例，其中我们将 λ 抽象 $(\lambda \mathtt{x}_1 : \mathtt{T}.(\lambda \mathtt{x}_2 : \mathtt{T} \to \mathtt{T}.\mathtt{x}_2)\mathtt{x})$ 应用于参数 $(\lambda \mathtt{x}_0 : \mathtt{T}.\mathtt{x}_0)$。然而，请注意，对于任何类型为 T 的参数，项 $\lambda \mathtt{x}_1 : \mathtt{T}.(\lambda \mathtt{x}_2 : \mathtt{T} \to \mathtt{T}.\mathtt{x}_2)\mathtt{x}$ $\lambda \mathtt{x}_2 : \mathtt{T} \to \mathtt{T}.\mathtt{x}_2$ 的求值结果都是。因此，总体而言，示例 5 将函数 $\lambda \mathtt{x}_2 : \mathtt{T} \to \mathtt{T}.\mathtt{x}_2$ 应用于参数 $(\lambda \mathtt{x}_0 : \mathtt{T}.\mathtt{x}_0)$，其结果为类型 T → T。

最后，我们想提醒读者，初始全局环境为 $\Gamma = \{\mathtt{x} : \mathtt{T}\}$ -，因此，我们所有的简单类型化 λ 项都只有一个基类型 T。

4 模型架构

我们的模型由一个标准的 Transformer [13] 模型组成，该模型从语法中预测规则，以为简单类型λ演算项生成类型。受到形式语言中所包含的丰富句法结构信息的启发，我们的模型接收输入的 CST 的预处理序列表示，而不是文本中的词法单元原始序列。由于我们是根据一个项预测其类型，编码器接收该项的预处理序列表示，而解码器接收类型的标准右移预处理序列表示。该模型的嵌入维度大小为 1024，Transformer 的解码器和编码器均具有 3 层。

4.1 名称预处理与开放词汇问题

在生成 CST 的序列表示之前，我们会按照本节所述，对项中的名称进行预处理。首先，我们说明进行这项预处理的动机。在程序合成中，不像自然语言那样存在闭合词汇表。例如，程序中的变量、函数、类型或任何对象都可以有任意名称。为了避免在实验中解决这个具有挑战性的开放问题，我们假设存在一个固定的全局上下文，使我们能够预先知道模型可能遇到的所有符号。因此，为了维持闭合词汇表这一假设，我们只使用 32 个不同的绑定变量名称。此外，所有绑定变量都经过预处理，按照其广度优先搜索（BFS）顺序重新命名

在 CST 中。因此，我们在训练（和推理）之前修改语法，将语法中的正则表达式（见第 8.1 节）替换为全局上下文中的名称——即 $\Gamma = \{x : T\}$ ——以及新的、唯一的 32 个 BFS 变量名称。做好这些准备后，我们接下来将说明如何为模型的编码器和解码器创建输入 CST。

4.1.1 编码器和解码器的 CST 输入

编码器输入： 编码器接收一个词元序列，该序列来自对简单类型λ演算项的 CST 进行广度优先搜索遍历。为了生成词元序列，我们按照广度优先搜索的顺序遍历该项的 CST，并使用边符号和节点符号。中间节点的节点符号是语法中的非终结符，叶节点的节点符号则是终结符。此外，对于批次中的每个项，我们会在序列开头和结尾分别添加特殊词元“sos”和“eos”。最后，我们将原始 Transformer [13] 中的位置嵌入添加到输入序列中。在输入编码器的一个批次中，各项的长度可能不同。为此，我们会在较短的项序列后添加填充标记，使其长度达到批次中最长项序列的长度。在训练和推理期间，这些填充标记会被屏蔽，不会影响模型的输出。请注意，如果要使用以序列作为输入的自回归模型（例如 Transformer），而不是使用 CST 这类结构化输入，就必须对序列采用某种遍历顺序（例如我们使用的广度优先搜索）。

解码器的输入： 我们遵循标准 Transformer [13] 中解码器所采用的相同输入格式。具体而言，由于解码器要预测类型对应的一系列语法规则，因此它接收的是右移后的语法规则嵌入序列。为生成此输入，我们对目标类型的 CST 进行广度优先搜索遍历，并据此生成语法规则序列。需要注意的是，将 CST 这类结构化输入转换为与自回归模型兼容的序列时，必须进行此类遍历。随后，我们将序列右移一个词元，并在批次中每个目标类型序列的开头和结尾分别添加特殊词元“sos”和“eos”。与编码器输入类似，我们还添加填充标记以处理类型长度不一的情况，并进行相应的掩码处理，以避免模型的输出受到影响。我们还将原始 Transformer [13] 中的位置嵌入应用于输入序列。

关于输入项中类型的存在性说明： 我们想强调的是，λ 抽象的约束变量的类型并未对模型隐藏。这些信息对于确保类型推断任务具有良好的适定性至关重要。如果没有这些信息，即使引入进一步的分布假设（例如，我们的数据来自人类生成的项），也无法解决类型推断问题。因此，输入项中包含提供给编码器的类型信息。

4.2 编码器和解码器堆栈

我们使用 PyTorch API [26] 构建 Transformer 的编码器层和解码器层，因此，所有层均严格按照原作者的描述实现[13]。我们在图 1 中展示了这一抽象，并选择仅明确呈现位置编码——这是因为我们进行了不同的消融实验。有关更多详细信息，请读者参阅原始 Transformer 论文 [13]和 PyTorch API [26]。表格和图中的实验结果所使用的超参数如下：所有嵌入维度均为 1024，编码器层和解码器层的层数均为 3，丢弃率为 0.1。表格和图中的实验结果所使用的超参数如下：所有嵌入维度均为 256，编码器层和解码器层的层数均为 1，丢弃率为 0.1。

4.3 文法规则预测

在方程 1 中的类型文法经过预处理、纳入全局上下文中允许的基本类型后，我们预测 λ演算中类型的文法规则 ID。对我们而言，这个全局上下文是 $\Gamma = \{ x : T \}$。请注意，实际应用中，这两种文法会合并在同一个文法中，如 8.1 节所示——由于项是简单类型化的，项文法包含类型文法。为了预测文法规则，我们遵循模型中图 1 所示的信息流，并将解码器的最后一个特征层作为线性层的输入，以预测类型文法规则的 ID。

4.4 训练

为了训练模型，我们将目标类型转换为按广度优先搜索顺序排列的规则序列。我们通过计算模型预测的规则序列与目标类型规则序列之间的交叉熵损失来计算损失。请记住，本研究的重点是考察优化器在各种超参数设置下的不同表现。因此，我们使用的优化器为：Adam、RAdam [14]和 Adafactor [16]。第 5 节的实验部分概述了我们训练实验中采用的不同消融研究和超参数。

4.5 推理 - 类型合成

对于类型的合成，我们遍历模型预测的规则序列分布，并通过选择最可能的语法规则 ID，以贪心方式合成一个类型。在合成类型时，我们按照广度优先搜索顺序展开模型预测的规则，因为模型正是使用按此方式排序的序列进行训练的。如果在合成过程中，预测的语法序列中某一对规则无法按照广度优先搜索顺序应用，我们就预测虚拟错误类型。更多详情请参阅算法 1。

算法 1 按广度优先搜索顺序贪心合成类型 1: 输入：术语批次 B，为其合成长度为 $|B|$ 的类型 2: $\tau = []$ 初始化一个空列表，用于存放预测的类型批次。 3: 对于 $b = 1, 2, ..., |B|$，执行 4: $T_b =$ 从术语批次 B 中获取索引为 b 的术语 5: $R_b =$ 获取术语 T_b 的预测规则序列 R_b 6: $G_b = []$ 初始化一个空列表，用于存放来自 R_b 的贪心规则 7: 对于 $t = 1, 2, ..., |R_b|$，执行 8: $p_{b,i} =$ 获取当前类型推断步骤中可能规则的分布 t 9: $\hat{r}_{b,t} = arg\max_{r \in 1,2,...,|p_b,_i|} \{p_{b,t}[r]\}$ 即，获取当前术语 T_b 在此时间步最可能的规则 t 10: $G_b[t] = r_{b,t}$ 即，存储在时间步 t 要预测的贪心规则 11: 结束循环 12: $\tau_b =$ 按广度优先搜索顺序应用来自 G_b 的贪心规则生成类型。如果某条规则无法根据广度优先搜索顺序中的前一条规则应用，则跳过该类型并追加虚拟错误类型。 13: $\tau[b] = \tau_b$ 将预测的类型 τ_b 追加到类型列表中。 14: 结束循环 15: 返回：输入术语 τ 对应的预测类型批次

5 实验

在本节中，我们将展示相关证据，以支持我们的结论：通过使用 Adafactor 优化器，我们解决了训练 Transformer 以解决类型推断问题时所面临的大规模超参数搜索问题。

5.1 预热阶段对于 Transformer 学习预测简单类型是否重要?

关于 Transformer 的文献特别强调，为了实现稳定学习，选择恰当的预热阶段至关重要 [12, 13, 14, 15, 16]。

预热阶段是训练的初始阶段，在此期间学习率从零开始逐渐增加至目标值。在本节中，我们使用 Adam [27] 探究不同学习率下预热步数的重要性。本小节的主要结果见表1。

从表1中我们可以看出，随着预热步数的变化，Transformer 的训练准确率并未发生变化。特别是当学习率为 10^{-5} 时，我们注意到无论预热步数是多少，模型始终保持完美预测。对于大小为 10^{-4} 的学习率，我们可以看到模型在大多数运行中都取得了良好的性能，但与 10^{-5} 相比，准确率并不完美——这表明，选择合适的学习率比采用预热阶段更为重要。对于最大的学习率 10^{-3}，我们原本预期预热阶段能够带来最大的收益，但即使采用较长的预热阶段，模型的性能仍被限制在 0.31——即使进行了较长的预热。

	训练准确率		
预热步数的数量	lr=10^{-3}	lr=10^{-4}	lr=10^{-5}
0	0.31	0.95	0.99
2,000	0.089	0.93	0.99
4,000	0.27	0.76	1.0
10,000	0.31	0.97	0.99

表1：展示预热步数对训练准确率影响的结果。我们使用来自 CSTs 的词元嵌入（仅采用位置编码）训练图1所示的 Transformer模型，具体方法如第10.1.1节所述。我们的模型采用批量大小为128进行训练，共迭代,000次，耗时2天4小时。模型的嵌入维度为1024，Transformer包含3个编码器层和3个解码器层。

预热阶段共进行 10,000 步。请注意，预热 $2,000$ 次迭代与先前研究 [15] 的建议一致——建议预热 $2(1-\beta_2)^{-1}$（其中 $\beta_2 = 0.999$ 是 Adam优化器的二阶矩超参数 [27]）。

这些实验表明，选择合适的学习率似乎就足以让 Transformer 学会进行类型推断。

5.2 RAdam 能解决收敛问题吗?

鉴于文献中对预热阶段的高度重视 [14, 12,13, 15]，以及调整预热超参数的复杂性，我们测试了修正Adam（RAdam）——这是一种声称能够解决 Adam 在训练初期高方差问题的优化器 [14]。

	训练准确率		
预热步数	学习率=10^{-3}	学习率=10^{-4}	学习率=10^{-5}
0	训练发散	0.91	1.0
2,000	-	0.96	0.99
4,000	-	0.80	1.0
10,000	-	0.96	0.99

表2：展示预热步数对使用 RAdam [14] 时训练准确率影响的结果。我们按照第 10.1.1 节所述，使用来自 CST 的词元嵌入（仅包含位置编码），训练图1所示的 Transformer模型。模型以批量大小 128 训练了 80，000 次迭代，耗时 2 天 4 小时。模型的嵌入维度为 1024，Transformer 的解码器和编码器均有 3 层。学习率为 10^{-3}、预热步数为 2000、4000 和 10，000 时，表中的结果尚未运行。我们选择不运行这些设置，因为预热会提高学习率，很可能导致类似的训练发散。

从表2可以看出，无论是否使用预热阶段，RAdam 的表现都与 Adam 类似。这并不意外，因为 RAdam 的设计目的就是解决训练初期的不稳定问题。然而，表1中的证据表明，预热阶段并未给我们的任务带来任何稳定性。这说明 RAdam 的影响也不会很大。对此，一个佐证是：不使用预热计划时（表2第1行），模型会发散；之后，在找到合适的学习率之前，模型始终无法达到完美准确率。

我们没有针对 10^{-3} 进行深入实验，因为在我们的实验中，训练始终会发散。为了解决这一问题，我们确实尝试将 RAdam 与两种不同的衰减调度器结合使用，以逐步降低学习率。我们尝试了一种衰减调度器（不进行预热），当训练损失进入平台期时便降低学习率，但这并未改善收敛情况。在另一项实验中，我们采用了指数衰减策略，每个训练轮次都降低一次学习率。这同样没有效果。对于学习率为 10^{-3} 的情况，这两种方法都未能解决训练发散的问题。不过，进一步开展实验仍然值得探索。

我们想强调的是，除了在训练 Transformer 时选择学习率之外，还必须选择用于训练 Adam 或 RAdam 的步数，并将其作为另一项超参数。此外，这一超参数还会与预热步数相互影响，使得这两个超参数的选择更加复杂。

5.3 使用 Adafactor 的实验

由于使用 Adam 和 RAdam 在小型数据集上训练模型以达到完美的训练准确率存在困难，我们因此在当前任务中测试了 Adafactor [16, 28] 。为了检验 Adafactor 的鲁棒性，我们还在表 3 中提供了使用模型不同变体进行的补充实验。我们在附录第 10 节中介绍了这些变体。

在所有这些设置下，训练均取得成功，并在 2–3 小时内达到完美的训练准确率，学习曲线也很平滑——与 5.4 中所示的曲线类似。为了观察退火调度器对学习率的影响，我们使用了默认的 Adafactor 调度器，发现这使达到相同训练准确率所需的时间缩短至原来的 1/2.5。每个训练轮次调用一次调度器时，训练时间从 10 小时缩短至 2–4 小时。

嵌入方法	训练准确率	验证准确率	迭代/秒	运行时间
词元（自然语言处理）	1.0	0.99	0.2	1.7 小时
路径 + f_{char}	1.0	0.99	0.2	2.1 小时
深度嵌入 + DE_{RI}	1.0	1.0	0.2	2.4 小时

表 3：结果显示，使用 Adafactor [16]通过不同方法嵌入 λ演算项训练模型的成功情况。我们按照 Hugging Face Adafactor [28]文档中概述的设置进行配置，并使用每个训练轮次调用一次的调度器。在本例中，这意味着针对我们的任务每 78 次迭代调用一次。请注意，不使用调度器时模型也会收敛，但耗时约 10 小时，而不是约 2 小时。我们按照图 1 中的描述，使用第一列所概述的不同嵌入训练 Transformer模型。第一行（nlp）指的是自然语言处理（NLP）中使用的纯词元模型，如第 10.1.1 节所述。第二行（path+f_{char}）表示按照第 10 节所述对路径进行嵌入的模型。第三行（Depth Embed +DE_{RI}）表示根据生成该词元的父语法规则进行深度嵌入的模型，如第 10 节所述。模型使用批量大小 128 进行训练，直到达到完美的训练准确率。模型的嵌入维度为 1024，Transformer的解码器和编码器均包含 3 层。

我们还想强调，**我们的模型达到了完美的验证准确率**。这表明，当用于训练和测试的项具有相同的固定深度时，模型能够完美泛化。不过，我们推测，在测试系统性泛化时需要用到搜索方法（如束搜索）。例如，当验证集中的项的深度大于用于训练模型的项的深度时。

5.4 Adam 与 Adafactor 学习曲线的比较

为了了解图 1 中 Transformer模型的训练难度，我们分析了图 5.4 所示的成功训练实验的学习情况。主要观察结果是，采用退火调度器的 Adafactor [16]仅需 2.17 小时即可完成训练，而使用线性预热阶段的 Adam 训练同一模型则耗时 1 天 15 小时。我们根据以下公式选择退火调度器的速率

$$\mathsf{scheduler_rate} = \min(\mathsf{one_epoch}, 2(1 - \mathsf{beta_2})^{-1})$$

这导致每个训练轮次都会调用一次调度器速率。不过请注意，即使没有退火调度器，我们的 Transformer 也能成功完成训练，只是耗时约 10 小时。

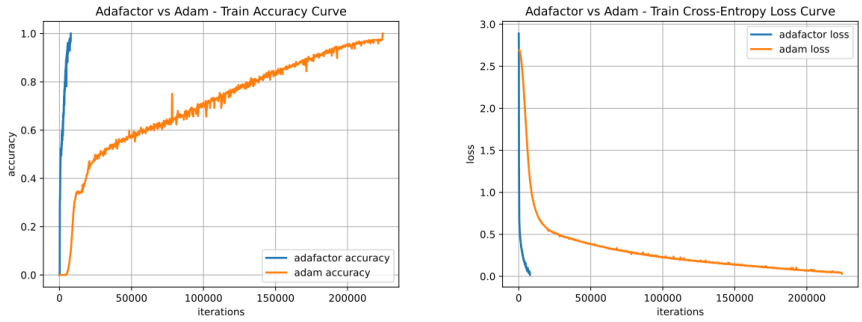

图 2：**Adafactor 达到完美训练准确率的速度比 Adam 快 11 倍。** 两种优化器都训练到训练集上达到完美准确率为止。Adafactor 使用 Hugging Face [28] 提供的默认调度器，而 Adam 使用线性预热计划。Adam 和 Adafactor 的学习率 10^{-5}均采用 Hugging Face [28] 默认的超参数，其中不包括为我们搜索学习率。Adafactor 没有进行任何超参数调优——除了每个训练轮次调用一次调度器。Adafactor 训练了 7,821 次迭代才达到完美准确率，其调度器每两个训练轮次调用一次——在本例中约为 146 次迭代。Adam 训练了 224,439 次迭代，耗时 1 天 15 小时 43 分钟才达到完美准确率。Adafactor 训练了 7,821 次迭代，耗时 2 小时 10 分钟才达到完美准确率。请注意，我们确实尝试过对 Adam 使用退火调度，但未能胜过 Adafactor。

5.5 硬件和框架参数

所有时间均基于在单个 IBM Research Cluster 虚拟节点上进行的测量结果，该节点使用 1 个图形处理器（NVidia Tesla P100，配备 16 GB GPU 内存）和 16 个虚拟 CPU 核心（Intel Xeon Platinum 8260 (R)，主频 2.40GHz，配备 118 GB CPU 内存）。实现使用了 PyTorch 1.9.1[26]。我们还使用了 ultimate-utils 库 [29] ——一个用于机器学习的工具库——来支持我们的 PyTorch 模型和训练实现。

6 讨论

本文通过使用 Adafactor 优化器[16]，解决了训练 Transformer 学习类型推断时需要进行大量超参数搜索的问题。这种方法有诸多优点：a）通过添加一个每隔

$$\text{scheduler_rate} = \min(\text{one_epoch}, 2(1 - \text{beta_2})^{-1})$$

（在本工作中，每个训练轮次）b）即使没有调度器，该方法也能正常工作

而且只需跟踪模型何时达到完美准确率（或收敛）即可。此外，Adafactor 优化器的原始作者对其进行了专门设计，使其能够以次线性方式使用内存——这一特性在大型语言模型时代尤为宝贵。此外，由于超参数的选择存在困难，Adafactor 能够加快研究进程，并建议研究人员直接在 transformers 中使用默认配置，即不搭配调度器。文中强调了选择 Adafactor 而非 Adam 的原因：Adam 的超参数搜索，尤其是迭代次数的选择，是在已经通过 Adafactor 取得成功的实验基础上进行指导的。令我们惊讶的是，我们发现 RAdam 比 Adam 更容易出现发散，因此不建议在我们的模型和任务中使用它。

最后，我们推测，与自然语言训练相比，在编程语言任务（如类型预测）上进行训练具有不同的优化景观。我们假设这是因为，当改变一个项中的单个词元时，会导致完全不同的语义；而在自然语言中，改变一个句子中的单个词元通常只会使句子变得不自然。我们的证据表明，即使是在最简单的任务上训练 Transformer——预测一个简单类型λ演算项的类型——也需要谨慎地进行优化。与此同时，相关研究独立发现了 Transformer 在形式规则训练方面的困难，并将这一现象称为“Grokking 效应”。他们观察到，经过长时间的训练后，Transformer模型能够从随机猜测水平跃升至完美泛化。基于我们的证据，我们推测 Adafactor 或许能够解决这一问题，并建议将其作为未来的研究方向。此外，我们推测，即使使用 Adafactor，Transformer 也不太可能表现出系统性泛化能力，无法真正学会底层规则，也无法泛化到预测训练时未曾接触过的、更深层次的项。

7 相关工作

自最初的 Transformer [13] 以来，人们就开始采用独特的学习方案来训练 Transformer 执行自然语言任务，其中使用了以下学习率方案：

$$\mathsf{lrate} = 0.0325 \cdot \min(\mathsf{step_num}^{-0.5}, \frac{\mathsf{step_num}}{252982})$$

这对应于先线性提高学习率，然后按照步数平方根的倒数降低学习率。

后续研究探讨了是否需要预热阶段，并通过其 RAdam 优化器发现自适应学习率 [14]存在较高方差。然而，后续研究表明，自适应学习率的方差虽然发散，却不会导致更新项 [15]发散。这项新的后续研究指出， $2 \cdot (1 - \beta_2)^{-1}$的预热步数。 然而，我们的实验表明，在我们的任务中，预热步数不会影响模型的收敛；真正起作用的是所用的优化器类型

，或者使用较低的学习率进行较长时间的训练就足够了。我们的证据表明，训练基于 Transformer 的模型来学习形式规则的难题——此前的研究将其称为“Grokking” [18] ——或许可以通过使用 Adafactor 优化器 [16] 来避免，因此我们建议从业者和研究人员优先尝试这种方法。最后，Adafactor [16] 优化器的设计目标是让 Transformer 的内存占用呈次线性增长；我们的实验表明，该优化器稳定性良好，且只需调节一个潜在的超参数。未来的工作可以针对更多编程语言挑战，检验 Adafactor 在 Transformer 上的稳定性。

致谢

我们谨向 IBM 的 Tejas I. Dhamecha 致以感谢，感谢其分享在训练 Transformer 神经网络方面的经验，并在 2021 年夏季通过 Slack 与我们进行讨论。

参考文献

[1] Alex Krizhevsky、Ilya Sutskever 和 Geoffrey E Hinton。基于深度卷积神经网络的 ImageNet 图像分类。[2] 何恺明、张祥雨、任少卿和孙剑。用于图像识别的深度残差学习。*IEEE* 计算机学会计算机视觉与模式识别会议论文集，2016 年 12 月：770–778，2015 年 12 月。

[3]Jacob Devlin、Ming Wei Chang、Kenton Lee 和 Kristina Toutanova。《BERT：用于语言理解的深度双向 Transformer 预训练》。*NAACL HLT 2019——2019* 年北美计算语言学协会分会会议：人类语言技术——会议论文集，1:4171–4186，2018 年 10 月。

[4]GPT 模型详解。OpenAI 的 GPT-1、GPT-2、GPT-3——沃尔玛全球科技博客。

[5] David Silver、Aja Huang、Chris J. Maddison、Arthur Guez、Laurent Sifre、George Van Den Driessche、JulianSchrittwieser、Ioannis Antonoglou、Veda Panneershelvam、Marc Lanctot、Sander Dieleman、Dominik Grewe、John Nham、Nal Kalchbrenner、Ilya Sutskever、Timothy Lillicrap、Madeleine Leach、Koray Kavukcuoglu、Thore Graepel 和 Demis Hassabis。利用深度神经网络和树搜索掌握围棋。《自然》*2016* 年 *529:7587*，529(7587):484–489，2016 年 1 月。

[6] Volodymyr Mnih、Koray Kavukcuoglu、David Silver、Alex Graves、Ioannis Antonoglou、Daan Wierstra 和 Martin Riedmiller。《利用深度强化学习玩转雅达利游戏》。

[7] Weirui Ye、Shaohuai Liu、Thanard Kurutach、Pieter Abbeel、Yang Gao、清华大学、加州大学伯克利分校、Shanghai Qi 和 Zhi Institute。《用有限数据精通雅达利游戏》。2021年10月。[8] Sarah Loos、Geoffrey Irving、Christian Szegedy 和 Cezary Kaliszyk。《深度网络引导的证明搜索》。46:85–63，2017年1月。[9] Ashwin K. Vijayakumar、Dhruv Batra、Abhishek Mohta、Prateek Jain、Oleksandr Polozov 和 Sumit Gulwani。《神经网络引导的演绎搜索：基于示例的实时程序合成》。第*6*届国际学习表征会议（*ICLR 2018*）—会议论文集，2018年4月。[10] Yifan Xu、Lu Dai、Udaikaran Singh、Kening Zhang 和 Zhuowen Tu。《通过自我学习进行神经程序合成》。2019年10月。[11] Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman、Alex Ray、Raul Puri、Gretchen Krueger、Michael Petrov、Heidy Khlaaf、Girish Sastry、Pamela Mishkin、Brooke Chan、Scott Gray、Nick Ryder、Mikhail Pavlov、Alethea Power、Lukasz Kaiser、Mohammad Bavarian、Clemens Winter、Philippe Tillet、Felipe Petroski Such、Dave Cummings、Matthias Plappert、Fotios Chantzis、Elizabeth Barnes、Ariel Herbert-Voss、William Hebgen Guss、Alex Nichol、Alex Paino、Nikolas Tezak、Jie Tang、Igor Babuschkin、Suchir Balaji、Shantanu Jain、William Saunders、Christopher Hesse、Andrew N Carr、Jan Leike、Josh Achiam、Vedant Misra、Evan Morikawa、Alec Radford、Matthew Knight、Miles Brundage、Mira Murati、Katie Mayer、Peter Welinder、Bob McGrew、Dario Amodei、Sam McCandlish、Ilya Sutskever 和 Wojciech Zaremba。《评估基于代码训练的大语言模型》。[12] 教程6：Transformer与多头注意力 — UvA深度学习笔记本 v1.1 文档。[13] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N. Gomez、 ·Lukasz Kaiser 和 Illia Polosukhin。《Attention Is All You Need》。载于神经信息处理系统进展，第2017年12月卷，5999–6009页。神经信息处理系统基金会，2017年6月。[14] Liyuan Liu、Haoming Jiang、Pengcheng He、Weizhu Chen、Xiaodong Liu、Jianfeng Gao 和 Jiawei Han。《关于自适应学习率的方差及其他问题》。2019年8月。[15] Jerry Ma 和 Denis Yarats。《未调优预热对自适应优化是否足够》。2019年10月。[16] Noam Shazeer 和 Mitchell Stern。《Adafactor：具有次线性内存开销的自适应学习率》。第*35*届国际机器学习会议（*ICML 2018*），10:7322–7330，2018年4月。

[17] Zeyu Sun、Qihao Zhu、Yingfei Xiong、Yican Sun、Lili Mou 和 Lu Zhang。TreeGen：一种用于代码生成的基于树的 Transformer 架构。技术报告。[18] Alethea Power、Yuri Burda、Harri Edwards、Igor Babuschkin 和 Vedant Misra。Grokking：在小型算法数据集上超越过拟合的泛化。[19] Stanislas Polu 和 Ilya Sutskever。用于自动化定理证明的生成式语言建模。2020 年 9 月。[20] Markus N Rabe、Google Research、Dennis Lee、Kshitij Bansal 和 Christian Szegedy。通过自监督跳树训练进行数学推理。技术报告。[21] Kaiyu Yang 和 Jia Deng。通过与证明助手交互学习证明定理。2019 年 6 月：12079–12094，2019 年 5 月。[22] A. J. Kfoury、J. Tiuryn 和 P. Urzyczyn。ML 类型可判定性是指数时间完全的。计算机科学讲义（包括人工智能讲义和生物信息学讲义子系列），431 LNCS：206–220，1990 年。[23] Harry G. Mairson。判定 ML 类型可判定性是确定性指数时间完全的。*ACM 编程语言原理年度研讨会会议论文集*，第 382–401 页，1990 年。[24] Harry G Mairson。线性 λ 演算与多项式时间完全性。[25] Adam Chlipala。使用依赖类型进行认证编程。技术报告，2019 年。[26] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga、Alban Desmaison、Andreas Kopf、Edward Yang、Zachary DeVito、Martin Raison、Alykhan Tejani、Sasank Chilamkurthy、Benoit Steiner、Lu Fang、Junjie Bai 和 Soumith Chintala。PyTorch：一种命令式风格的高性能深度学习库。载于*神经信息处理系统进展 32*，第 8024–8035 页。Curran Associates, Inc.，2019 年。[27] Diederik P Kingma 和 Jimmy Lei Ba。ADAM：一种随机优化方法。技术报告。[28] 优化——transformers 4.7.0 文档。[29] Brando Miranda。机器学习与人工智能终极实用工具库，2021 年。

8 附录

8.1 λ演算语法

这里我们使用代码中采用的 Lark Python语法，以扩展巴科斯范式（EBNF）格式给出 λ演算语法。回顾一下，除了全局上下文外，该语法还通过附加 32 个约束变量扩展为一个闭合词汇表。根据我们的语法，空白字符和括号均不作为词元处理。由此得到的语法结果如下[1]

```
项 : "lambda" 项 ":" 类型 "." 项  "[" 项 项 "]"
|  /[A-Za-z0-9_]+/ 类型 : 类型"->" 类型 // "->"
|  /[A-Za-z0-9_]+/ IGNORE_TOKENS : "("  |
")" %import common.WS %ignore WS %
ignore IGNORE_TOKENS
```

请注意，项和类型的语法交织在一起，因为项带有类型注释。

9 简单类型λ演算中的类型推断

要推断简单类型λ演算项 e 在上下文Γ中的类型，需要遍历该项的抽象语法树，并根据抽象语法树 e各层的构造器类别递归地构建类型。总体而言，类型推断过程如算法 2 所示。

CST 的位置编码的 10 种变体

为了测试 Adafactor 的鲁棒性，我们还为 CST 的初始输入序列添加了额外的结构信息，这与 TreeGen 模型 [17] 的做法类似。我们通过两种方式实现这一点：1. 基于 CST 中每个词法单元的路径，为 CST 中的词法单元提供嵌入；2. 在位置编码层之外使用深度编码层。我们将回顾这些方法并说明其动机，同时阐明我们为鲁棒性实验所做的调整。

[1] https://github.com/FormalML/type-parametric-synthesis/blob/master/tinfer/simply_type_lambda_calc.ebnf

算法 2简单类型 λ 演算的类型推断

定义 推断类型(e, Γ):
输入： e 我们想要推断类型的输入项。上下文 Γ 项-类型对 x 个配对：τ。
如果项 e 是一个 $Variable$ **那么**
    $\tau_e = \Gamma.get_type(e)$
    返回： τ e
否则如果 项 $e = \lambda x$: $\tau.body$ 是一个 $Abstraction$ **则**
    $\Gamma' = \Gamma \cup \{x : \tau\}$
    $\tau_{abs} = infer\ type\ (body,\ \Gamma')$
    返回： τ_{abs}
否则 项 $e = (e_{left})e_{right}$ 是一个 $Application$ **然后**
    $\tau_{left} = infer\ type(e_{left},\ \Gamma)$
    $\tau_{right} = infer\ type(e_{right},\ \Gamma)$
    $\tau_{apply} = \tau_{left} \rightarrow \tau_{right}$
    返回： τ_{apply}
结束 if

10.1 动机

我们的初步实验揭示了一个出乎意料的挑战：训练 Transformer 学习执行类型推断。我们发现，Adafactor 优化器是一种效果出乎意料地好的模型训练策略。为了检验这一策略在我们的实验设置中是否具有普遍有效性，我们决定在使用 Adafactor 的同时，改变输入模型的嵌入方式，以测试 Adafactor 的鲁棒性。主要实验结果的表 3 显示，该模型确实保持了鲁棒性，但收敛时间并未得到改善。

10.1.1 编码器和解码器输入的变体

路径嵌入方法：受这样一个事实的启发：使用符号的广度优先搜索排序生成序列会丢失项的结构信息，因此我们实现了类似于 [17] 的路径嵌入。该方法将 CST 中的每个节点替换为通向该节点的路径，以便在输入的分词过程中编码这些结构信息。路径是遍历 CST 到达特定节点时所经过的符号序列，其中包括最终访问节点的符号。因此，我们使用前馈神经网络，通过节点对应的路径对词元进行嵌入，其方式类似于 [17] 中所述的方法。路径嵌入的计算过程是

$$\mathsf{embed}(x_{\mathrm{path}}) := \mathsf{FFD}[x_1; \ldots; x_L] \tag{3}$$

其中，FFD 是具有可训练权重的前馈网络， L 是路径的最大长度， x_i 是路径中符号的嵌入。FFD 在 R^M 中输出路径嵌入。这里， $[x_1;\ ...;\ x_L]$ 表示字符向量嵌入的按行拼接操作。具体而言，给定一个

由 L 个列向量组成的列表，；拼接操作会将它们堆叠起来。因此，如果 $x_i \in R^D$，那么矩阵 $[x_1; ...; x_L]$ 属于 $R^{L \times D}$。

在我们的实验中，我们设置为 $L = 13$，因为长度为 13 的路径能够表示相当有意思的项——对于二叉树来说，它允许表示约有 8,192 个节点的项。对于短于 L 的路径，我们会用一个特殊的“填充”词元补齐；之后该词元会被屏蔽，不会影响模型的预测。请注意，路径嵌入类似于在自然语言处理中使用字符嵌入。在自然语言处理中，它们会用组成词法单元的单个字符替换该词法单元，然后对其进行嵌入。类似地，我们会用对应于每个词法单元的节点路径替换该词法单元。

深度嵌入方法：遗憾的是，路径嵌入方法的计算开销较大，因为它使用一个符号序列来嵌入每个词元，这类似于自然语言中对单词进行字符嵌入。由于代码并不总是能够轻易地完全向量化以实现路径嵌入，因此我们转而尝试使用深度嵌入，将结构信息注入输入序列（类似于 [17]）。

假设我们有一个长度为 L 的 BST 排序后的 CST 嵌入向量序列 $x_{cst\ seq} = [x_{token_1}, \ldots, x_{token_L}]$，其嵌入维度为 D。我们有 $x_{cst\ seq} \in R^{L, D}$。那么，该序列的深度嵌入 DE 是指使用 CST 中每个词元的父节点所构成的嵌入序列。因此，$DE_{x_{cst\ seq}} = [DE_{token_1}, \ldots, DE_{token_L}] \in R^{L, D}$。接下来，我们将这个序列向量加到 $x_{cst\ seq}$ 上——其方式与添加位置嵌入的方式相同：

$$x_{seq} = [x_{token_1}, \ldots, x_{token_L}] + [DE_{token_1}, \ldots, DE_{token_L}] \tag{4}$$

其中，$x_{token_i} \in R^D$ 是 $token_i$ 的嵌入，$DE_{token_1} \in R^D$ 是 CST 中 $token_i$ 的父节点的嵌入。此外，我们始终会添加原始 Transformer 模型 [13] 中提出的常规位置编码。

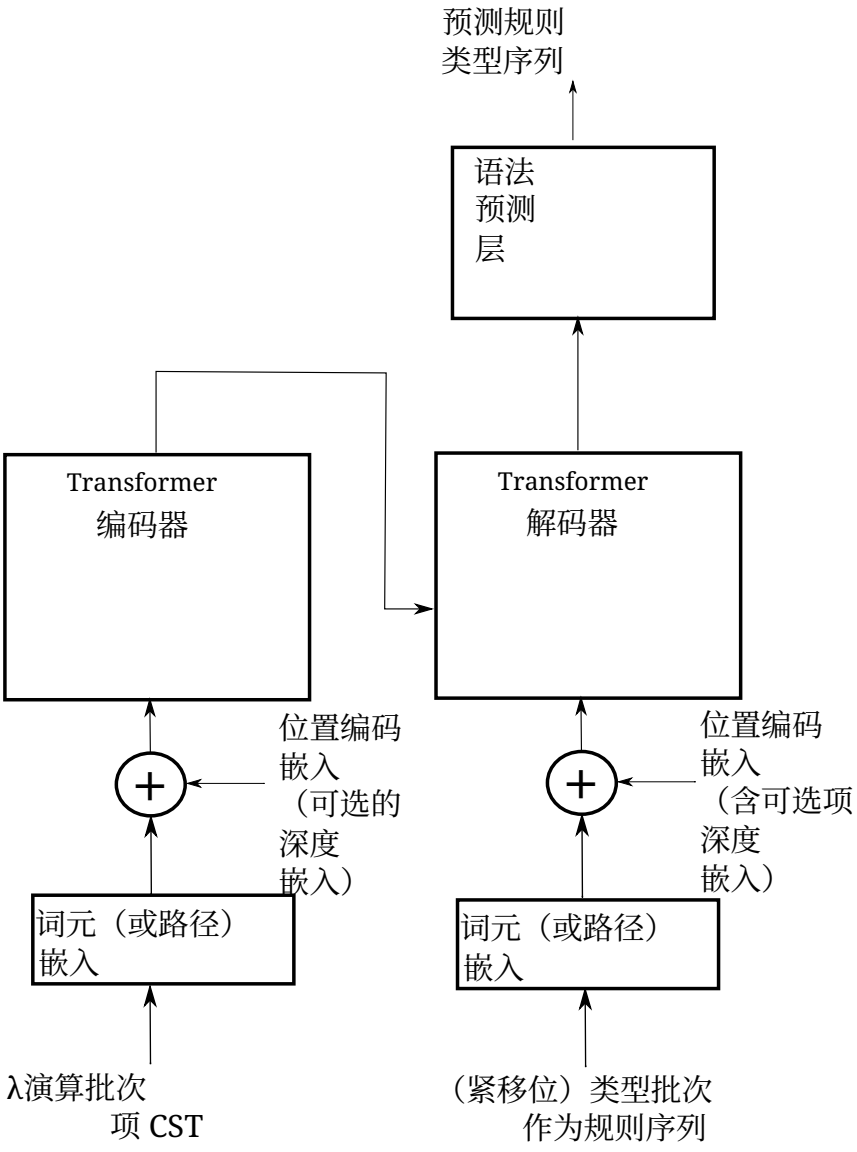

图1：我们用于简单类型λ演算的 Transformer 模型。