TypeT5：基于静态分析的序列到序列类型推断

JiayiWei, GregDurrett, IsilDillig德克萨斯大学奥斯
汀分校计算机科学系{jiayi,gdurrett, isil}
@cs.utexas.edu

arXiv:2303.09564v1 [cs.SE] 16 Mar 2023

摘要

人们对自动预测 Python 和 JavaScript 程序中缺失的类型注解越来越感兴趣。尽管先前的方法在预测最常见的类型时已取得令人瞩目的准确率，但在处理罕见或复杂类型时往往表现不佳。本文提出一种新的类型推断方法，通过利用面向代码的先进序列到序列预训练语言模型 CodeT5，将类型预测视为代码填充任务。我们的方法使用静态分析为模型将要预测类型签名的每个代码元素构建动态上下文。我们还提出一种迭代解码方案，将先前的类型预测纳入模型输入上下文，从而使相关代码元素之间能够交换信息。评估结果表明，所提出的方法 TypeT5 不仅实现了更高的整体准确率（尤其是在罕见且复杂的类型上），还能生成更连贯的结果并减少类型错误，同时便于用户干预。

1 引言

在 Python 和 JavaScript 等语言中，由于缺乏静态类型系统，维护和分析代码库变得更加困难。为了解决这一问题，渐进式类型 （Siek & Taha，2007年）应运而生，它允许逐步为无类型代码库添加类型注释，从而兼具静态类型的优势与快速原型开发的便利。因此，包括 Python 和 JavaScript 在内的许多主流编程语言已经采用了这一理念，研究人员也开发了基于学习的技术来预测缺失的类型注释（Raychev 等人，2015；Hellendoorn 等人，2018；Wei 等人，2020年；Pradel 等人，2020年；Allamanis 等人，2020年；Pandi 等人，2020年；Jesse 等人，2021年；Mir 等人，2022年；Jesse 等人，2022年；Peng 等人，2022年）。

与此同时，随着大规模预训练的兴起和 Transformer 架构的蓬勃发展，序列到序列模型已被证明在代码注释生成（Panthaplackel et al., 2020）、代码补全（Wang et al., 2021; Ahmad et al., 2021）和代码合成（Li et al., 2022）等编程任务中非常有效。这类模型一项尤为吸引人的特性是，由于采用了子词分词（Gage，1994；Schuster & Nakajima，2012；Sennrich et al., 2016），它们能够在测试时生成任意代码表达式——包括新颖的标识符名称和抽象语法树结构。然而，与通常只需将周边代码作为上下文便能很好完成的代码补全任务不同，有效的类型推断通常需要非局部信息，包括可能属于完全不同文件的代码片段。例如，考虑一个函数 f，它将一个名称泛泛的参数 x 直接传递给另一个函数 g。仅查看 f 的函数体，可能很难判断 x 的类型。当程序员遇到这种情况时，他们通常会检查 f 的调用方和被调用函数，有时甚至会沿调用关系递归检查，以确定 x 的预期类型。因此，在许多情况下，仅查看给定变量的直接上下文，可能不足以准确预测其类型。

我们的方法 TypeT5 通过使用静态分析来识别代码库中对每次预测有用的部分，从而解决了这一挑战。具体而言，我们构建了一种所谓的 使用图，其中节点对应代码元素（即我们希望预测其类型的函数或变量），边表示它们之间潜在的使用者—被使用者关系。给定这样一张图后，我们将给定代码元素的使用者和被使用者编码为类似普通代码的形式，并将其作为

额外的上下文输入 Transformer 模型。为了充分利用序列到序列范式，我们还提出了一种迭代解码方案，通过上下文传入先前的类型预测，使信息能够在整个代码库中跨越相距较远的代码元素进行传播。

我们基于广受欢迎的 CodeT5 模型实现了 TypeT5，并用它为未标注类型的 Python 代码合成类型注释。我们的评估将 TypeT5 与三种最先进的类型推断工具（Allamanis 等人，2020年；Mir 等人，2022年；Peng 等人，2022年）以及一个不利用静态分析的 CodeT5 基准进行比较。结果表明，TypeT5 大幅优于所有基线，同时显著提升了对罕见且复杂的类型的准确率。我们的消融研究证实了我们对 CodeT5 基线所做各种修改的益处；此外，一项类型检查实验表明，所提出的迭代解码方案还能提高生成的类型赋值的一致性，从而减少类型约束违反。最后，我们探索了模型的一种替代用法：用户可以交互式地检查模型的预测并进行必要的修正。结果证明，我们的方法作为开发者工具，可用于为完全没有类型注释的项目添加注释——平均而言，用户只需修正模型每五项预测中的一项。

总而言之，本文作出了以下贡献：

• 我们将 CodeT5 应用于推断 Python 类型注释，结果表明其性能显著优于以往方法。据我们所知，这是首个能够预测参数化类型和用户定义类型的基于机器学习的技术。

• 我们通过应用静态分析技术改进了原始 CodeT5 模型，帮助模型推理局部上下文之外的信息，进一步提升了其性能。
• 我们提出了一种迭代解码方案，尤其有助于提升连贯性，其衡量标准是类型检查器报告的类型错误数量。此外，我们还提出了一种新颖的设置，将序列到序列解码方案与用户干预相结合。

2 概述

本节通过图 1 所示的示例，说明 TypeT5 的设计动机。该示例包含一个方法 `predict` 和两个函数 `evalon dataset` 与 `chuck srcs`，它们分别实现在不同的文件中。给定这段代码的无类型版本，我们的目标是自动推断类型注释（以绿色突出显示）。由于大量使用用户定义类型（如 `ChunkedDataset`、`PythonType` 和`ModelWrapper`），以及复杂的参数化类型，例如`dict[int,list[PythonType]]`，现有的类型推断技术难以处理此示例。

model.py

```python
class ModelWrapper:
    ... # other methods and attributes omitted

    def predict(
        self,
        data: ChunkedDataset,
        n_seqs: Optional[int] = None,
    ) -> dict[int, list[PythonType]]:
        pred_types = dict()
        for batch in data.data:
            batch["input_ids"] = batch["input_ids"].to(device)
            preds, _ = self.predict_on_batch(batch, n_seqs)
            for i, c_id in enumerate(batch["chunk_id"]):
                if n_seqs is None:
                    pred_types[c_id] = preds[i]
                else:
                    span = i * n_seqs : (i + 1) * n_seqs
                    pred_types[c_id] = preds[span]
        return pred_types
```

eval.py

```python
def eval_on_dataset(
    model: ModelWrapper,
    data: TokenizedSrcSet,
    window_size: Optional[int] = None,
) -> dict[int, list[PythonType]]:
    window = copy(model.DefaultWindow)
    if scale_window is not None:
        window.left_tokens = window_size
        window.right_tokens = window_size

    chunks = chunk_srcs(data, window)
    return model.predict(chunks, n_seqs=None)
```

data.py

```python
def chunk_srcs(
    data: TokenizedSrcSet,
    window: WindowArgs
) -> ChunkedDataset:
    ...  # implementation omitted
    return ChunkedDataset(...)
```

图 1：摘自我们自有代码库的简化代码片段。数据集评估函数首先调用`chunk srcs` 函数，将给定的文本数据转换为大小相同的数据块，然后将这些数据块传递给`ModelWrapper.predict` 方法。

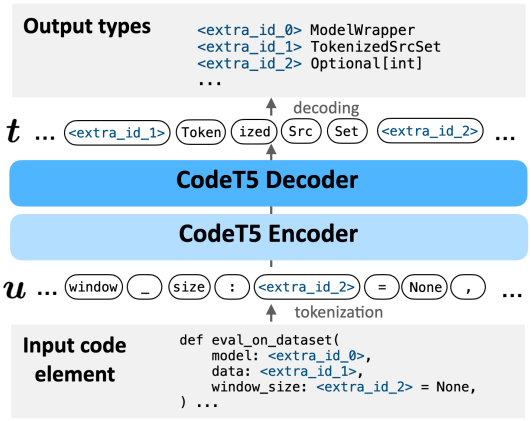

图2：CodeT5 如何使用字节对编码对源代码进行编码和解码。标记令牌（以蓝色突出显示）表示空缺（输入）及其对应的填充内容（输出）。

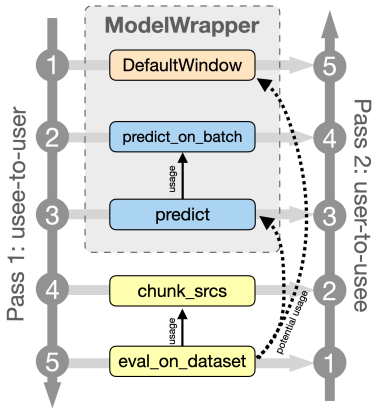

图3：两遍迭代式解码过程。

将类型推断视为代码填充 在本研究中，我们提出一种新方法，将类型推断视为代码填充任务。由于类型注释可以看作缺失的代码片段，我们对当前最先进的代码填充模型 CodeT5 进行微调，并以此作为起点。CodeT5 使用字节对编码（Radford 等人；Gage，1994年）生成子词令牌序列（见图 2），因此原则上可以预测任意代码片段来填充掩码空缺，包括包含复杂参数化类型和用户定义类的类型注释。

通过静态分析引入上下文 通过将周围代码用作预测上下文，我们微调后的 CodeT5 可以相对轻松地预测某些变量的正确类型注释。例如，基于`n seqs`（model.py，第 7 行）或`window size`（eval.py，第 4 行）的名称和默认值，CodeT5 可以推断出这些参数的正确类型。然而，对于其他参数，例如 eval.py 第 2 行中的`model`，周围的上下文并未包含足够的信息，无法做出合理的预测。要理解原因，请注意`ModelWrapper`是在单独文件中定义的新类，因此（1）它在训练过程中从未出现过，并且（2）其定义不在上下文中。同样，也很难预测`eval on dataset`的返回类型，因为它直接返回`model.predict`的结果，而模型同样无法获取其定义。

为了解决这一问题，我们的方法利用静态分析来增强 CodeT5 的预测上下文。我们将在第 3.3 小节详细介绍如何根据静态分析构建上下文。简而言之，我们的方法会分析代码元素之间的用户-被使用者关系，并将相关定义纳入上下文。例如，当模型对`eval on dataset`进行预测时，上下文会包含`DefaultWindow`和`predict`的定义；它们分别在`ModelWrapper`中定义，并在 eval.py 的第 6 行和第 12 行调用。

TypeT5 架构由于不同代码元素之间存在许多依赖关系，因此为每个代码元素进行独立类型预测并不理想。例如，要推断`eval on dataset`的返回类型，我们需要知道`ModelWrapper.predict`的返回类型，而它又取决于`self.predict on batch`的返回类型（model.py，第 12 行）。然而，由于模型可接收的上下文有限，因此无法纳入所有传递使用者和被使用者。为解决这一问题，TypeT5 使用迭代解码方案，使模型能够以先前的类型预测为条件。具体而言，我们的解码方案首先对使用者—被使用者图进行拓扑排序，然后进行两轮连续的预测遍次：先从被使用者到使用者，再沿相反方向进行，如图3所示。要了解这两个方向为何都有用，可以看到，`eval on dataset`的返回类型取决于`predict`，而后者又取决于`predicton batch`。因此，将信息从被调用函数传播到调用方显然很有用。反过来，考虑预测`data`的类型，它是`predict`的第一个参数。由于`chunk srcs`的返回值会作为第一个参数传入`predict`，沿相反方向传播信息也会有所帮助。

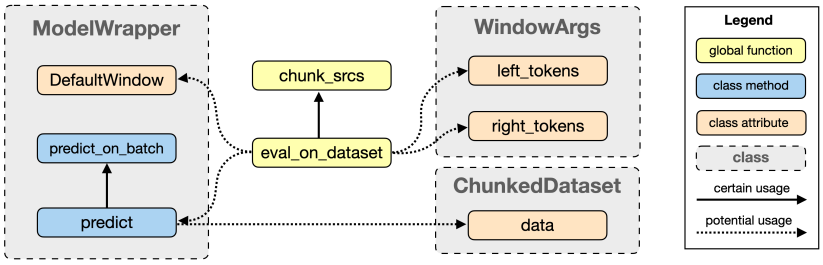

图4：图 1 中代码片段对应的使用图。`WindowArgs`和`ChunkedDataset`假定在同一项目的其他位置定义。

3 方法

3.1 使用 CodeT5 进行类型预测

我们将类型预测表述为一个序列到序列（seq2seq）任务。令 $\boldsymbol{u} = (u_1, \ldots\ldots, u_n)$ 表示由代码词元组成的序列，其中每个词元都是一个无类型的代码元素 e（函数或变量）。我们在希望预测类型的每个位置插入带索引的标记令牌 $\boldsymbol{u}$ `<extra id i>`，并让模型预测 $\boldsymbol{t} = (t_1, \ldots\ldots, t_m)$，即为 $\boldsymbol{u}$ 中标记位置的类型编码的词元序列。注意，$\boldsymbol{t}$ 仅包含类型，不含其他代码词元，其格式为 `<extra id 1>` `[type 1 tokens]` `<extra id 2>` `[type 2 tokens]`, , 依此类推。我们使用与 CodeT5 相同的分词器，它可以将任意 Python 表达式编码为一串（通常较短的）子词令牌。

我们的基线 CodeT5 模型是一个 Transformer 序列到序列模型 $P(\boldsymbol{t} \mid \boldsymbol{u}, \bar{\boldsymbol{u}})$，它在给定原始代码序列 $\boldsymbol{u}$ 及其周围代码 $\bar{\boldsymbol{u}}$的情况下，为类型序列 $\boldsymbol{t}$ 生成一个概率分布。该过程如图 2 所示；为清晰起见，省略了 $\bar{\boldsymbol{u}}$。

我们改进后的模型 TypeT5 使用通过静态分析构造的上下文替代周围代码 $\bar{\boldsymbol{u}}$，并将其记为 $\boldsymbol{s}$。因此，我们可以将模型写作 $P(\boldsymbol{t} \mid \boldsymbol{u}, \boldsymbol{s})$。请注意，在这两个模型中，我们都像先前的工作一样，在预处理步骤中删除了所有注释和 Python 文档字符串。现在，我们将展示如何通过静态分析构造 $\boldsymbol{s}$。

3.2 构建使用图

为克服仅将周围代码用作上下文的局限性，我们的方法依赖静态分析来提取有关代码库的相关全局信息。具体而言，我们的分析会构建一个使用图，其中节点对应代码元素，边表示直接使用。例如，如果 x 和 y 都是函数，那么从 x 到 y 的一条边意味着 x 调用了 y。类似地，对于变量，从 x 到 y 的一条边意味着 x 的值取决于 y。

我们前述示例的使用图如图4所示。我们展示了两种使用关系：确定使用（以实线箭头表示）和潜在使用（以虚线箭头表示）。确定使用是指无需知道相关变量的类型即可通过静态分析确定的使用。例如，全局函数`chunk srcs` 被`eval on dataset`直接使用；我们也知道`predict on batch` 由`predict` 调用（第12行对`self`的方法调用）。对于其他情况，例如非`self` 属性访问或方法调用，其目标取决于接收者的类型，因此我们首先从当前项目中收集所有名称匹配的属性和方法，然后为每个对象生成一条潜在使用关系。关于使用图构建的更多细节，请参见 A.1 小节。

3.3 构造模型输入

TypeT5 利用使用图 $\mathcal{G}$ 构造模型的输入。具体而言，我们将代码元素 e 的模型输入定义为以下四部分的拼接：前导部分 $\boldsymbol{s}_{\text{pre}}$、被使用者 $\boldsymbol{s}_{\text{usee}}$、主代码 $\boldsymbol{u}$和使用者 $\boldsymbol{s}_{\text{user}}$，构造方式如下（具体示例见 A.9 小节）。

• **前导部分**：$\boldsymbol{s}_{\text{前导部分}}$ 的主要作用是帮助模型将每个局部名称映射到其所指向的定义，因此前导部分包含所有导入语句。此外，为了帮助模型了解当前作用域中有哪些类型可用，前导部分还包含所有类定义的头部，以及当前文件中的所有类型变量声明。

• **主代码**：对于我们关注的代码元素 e，我们使用其定义来构造 $\boldsymbol{u}$。如果 e 是函数，$\boldsymbol{u}$ 就是其源代码；如果 e 是变量，$\boldsymbol{u}$ 则包含所有将 e 作为左侧的顶层赋值语句。此外，如果 e 是类 C的方法或属性，我们还会缩进 $\boldsymbol{u}$，并在其前面添加 C的头部，如图 7 所示。

• **使用者**：给定使用图 $\mathcal{G}$，我们通过纳入 users($\mathcal{G}$, e) 中所有元素的源代码词元来构造 $\boldsymbol{s}_{\text{用户}}$。由于这些元素可能来自不同的源文件，我们还会在每个元素前添加一条特殊注释，注明其所属的 Python 模块。请注意，如第 3.4 小节后文所述，这些元素还可以选择包含 TypeT5 模型预测的类型。

• **被使用者**：$\boldsymbol{s}_{\text{被使用者}}$ 不仅包含 e的直接使用者，还包含用户上下文$\boldsymbol{s}_{\text{用户}}$中使用的任何内容。也就是说，$\boldsymbol{s}_{\text{被使用者}}$ 包含 usees($\mathcal{G}$, e) $\cup \bigcup_{e' \in \text{users}(\mathcal{G},\ e)}$ usees($\mathcal{G}$, e') 中的元素。由于其中通常包含的元素远多于 $\boldsymbol{s}_{\text{用户}}$，因此我们只使用这些元素（有类型或无类型的）签名来构造$\boldsymbol{s}_{\text{被使用者}}$。

我们将输入总长度限制为最多 4096 个子词令牌，并从左右两侧截去超出部分，围绕主代码居中。涉及特定用法的上下文元素会安排得更靠近中心，以便始终优先于潜在用法。

3.4 迭代解码推理

We 现在描述如何在我们的基准模型以及上下文增强模型中进行推理 d mo，使用迭代解码方案。

CodeT5 解码： 给定训练好的模型 $P(\boldsymbol{t} \mid \boldsymbol{u},\ \bar{\boldsymbol{u}})$，我们可以使用束搜索推断出最可能的类型集合 $\boldsymbol{t}$，适用于 $\boldsymbol{u}$（以及周围上下文 $\bar{\boldsymbol{u}}$）。我们的实现会对单个代码块 $\boldsymbol{u}$ 的输出类型进行联合预测，因为块中后续的类型取决于先前类型的预测结果。不过请注意，$\boldsymbol{u}$ 和 $\bar{\boldsymbol{u}}$ 始终是完全未标注类型的代码：虽然我们在预测时会以先前的类型为条件，但不会将这些类型插入下一个元素的预测上下文中。[1]

TypeT5 迭代解码： 我们纳入上下文 $\boldsymbol{s}$ 的部分动机，是为了在推理时利用其中的类型信息。关键在于，这要求 $\boldsymbol{s}$ 已被赋予类型。然而，取自原始代码库的上下文并未赋予类型，因此 TypeT5 会利用自身的预测，迭代地为这些上下文添加类型签名。设 $\mathcal{M}$ 为模型生成的类型赋值，它将每个代码元素 e 映射到其预测类型 $\boldsymbol{t}_e$；并将 $\mathcal{M}(\boldsymbol{s})$ 记作根据 $\mathcal{M}$为 $\boldsymbol{s}$ 中的元素添加注释后得到的上下文。从空的 $\mathcal{M}$ 开始（这会使任何上下文$\boldsymbol{s}$ 保持不变），TypeT5 随后采用迭代解码方案更新 $\mathcal{M}$：该方案按使用图 $\mathcal{G}$ 遍历两遍，如图3所示。第一遍预测按被使用者到使用者的顺序进行，[2] 而第二遍则按相反方向进行，以实现双向信息流。在任意给定的时间步，我们可以将模型对元素 e 的预测记作取自$P(\boldsymbol{t}_e \mid \boldsymbol{u}_e,\ \mathcal{M}(\boldsymbol{s}_e))$，随后利用预测类型 $\boldsymbol{t}'_e$ 更新 $\mathcal{M}$，使得 $\mathcal{M}(e) = \boldsymbol{t}'_e$。

3.5 训练

为了节省计算资源并提升训练期间的并行度，我们使用现有的人类标注作为金标准类型赋值 $\mathcal{M}^*$，而不是让模型以自身的预测为条件。请注意，这种类型赋值通常并不完整，可能并未为每个缺失的类型都提供标签。我们使用教师强制训练模型，使其最大化对真实标注的预测对数似然，即对于每个 e（其中 $\boldsymbol{t}^*_e$ 可用），最大化 log $P(\boldsymbol{t}^*_e \mid \boldsymbol{u}_e,\ \mathcal{M}^*(\boldsymbol{s}_e))$。我们在相同的数据集上以类似方式训练 CodeT5 基线模型。[3]

[1]在我们的实验中，纳入预测类型实际上会因曝光偏差而损害性能。[2]这需要对 $\mathcal{G}$进行拓扑排序。当 $\mathcal{G}$ 不是有向无环图时，我们会任意打破其中的环。[3]请注意，如果不进行这一步训练（微调），原始 CodeT5 模型的表现会非常差，因为它倾向于 predict 任意的、并非类型的 Python 表达式。

表 1：两个数据集的基本统计信息。

	BetterTypes4Py			**InferTypes4Py**
	训练集	验证集	*test*	*test*
项目	573	40	50	3
非空文件	16.5K	1098	949	99
代码行数	2.4M	174K	139K	21K
顶层类型槽位	541K	38.2K	28.4K	4.6K
顶层用户添加类型	275K	19.3K	15.8K	2.7k
稀有类型比例	25.7%	23.3%	35.0%	33.8%
复杂类型比例	20.4%	16.6%	20.8%	33.4%
平均类型大小	1.42	1.33	1.43	1.72

4 实验

我们使用 Python 实现了 TypeT5，其源代码和模型权重可在 GitHub[4] 上找到。超参数和硬件详情列于 A.6 小节。下文首先将 TypeT5 与三个先进的 Python 类型推断系统进行比较，分别是 Typilus（Allamanis 等人，2020年）、Type4Py（Mir 等人，2022年）和 HiTyper（Peng 等人，2022年）。随后，我们通过消融研究评估影响模型性能的不同因素。

4.1 评估设置

数据集 在评估中，我们为每个 Python 项目的顶层 代码元素预测类型注释。这些元素包括所有类属性、方法、顶层函数和全局变量。我们将用户添加的现有类型注释视为真实标注，并将逐类型准确率作为主要性能指标。按照与 Allamanis 等人（2020）和 Wei 等人（2020）相似的设置，我们按 Python 项目划分数据集。这样一来，测试项目中新定义的类型在训练期间就不会被见到。与按文件划分数据集的做法相比（Type4Py 及其他研究均采用此方法；Pradel 等人，2020；Mir 等人，2022），这种设置更具挑战性，但也更能反映模型在现实世界中的性能（Gruner 等人，2022）。

我们的主要数据集 **BetterTypes4Py** 是从 Many-Types4Py 数据集（Mir 等人，2021年）中筛选出高质量子集构建而成的，该数据集曾用于训练 Type4Py。我们在 A.2 小节中介绍筛选标准。由于我们的模型是在 CodeT5 模型的基础上进行微调的（该模型可能已经在上述数据集中的某些测试代码仓库上进行了预训练），我们还构建了**InferTypes4Py**，这是一个从 Typilus、Type4Py 和我们自己的工具的源代码中衍生出的测试集；这些源代码均未用作 CodeT5 的（预）训练数据。我们还在附录中进一步讨论潜在的代码重复问题（A.3 小节）和标签质量问题（A.4 小节）。

我们在表 1 中总结了两个数据集的主要属性。我们将类型的大小定义为其主体中类型构造器的数量[5]，并将大小为 1 的类型归类为**简单**类型，否则归类为**复杂**类型。我们还根据类型是否包含一种罕见的类型构造器，将其归类为**罕见**或**常见**类型；这里的罕见类型构造器是指训练集中出现频率未进入前 100 的类型构造器。

准确率指标。 由于 Python 允许同一种类型采用不同的语法形式书写，[6]我们首先进行类型规范化，将预测类型和真实类型都转换为规范形式。有关此规范化步骤的详细信息，请参见 A.5 小节；我们将规范化后针对所有人工标注计算的准确率称为**完全准确率** 。为了更好地与先前的工作进行比较，我们还定义了**调整后准确率** （我们的主要指标），其做法是：**(1)** 过滤掉所有`None` 和`Any` 标签（与先前的工作相同）；**(2)** 将全限定名称转换为简单名称（例如，使用`Tensor` 而非`torch.Tensor`），因为一些先前的方法无法输出正确限定的类型；**(3)** 将任何最外层的`Optional[T]` 和`Final[T]` 改写为`T`，因为它们

[4]可在 https://github.com/utopia-group/TypeT5 获取。[5]例如，int 和 foo.Bar 的大小都是 1，而 dict[字符串、foo.Bar] 的大小为 3。[6]例如，Union[int,None] 和 Optional[int] 都表示一个也可以为 None 的整数；list 和 List[Any]都表示元素未指定类型的 Python 列表

表 2：**常见**类型上的准确率比较。

	BetterTypes4Py					InferTypes4Py				
	full	**调整后**			**base**	**full**	**调整后**			**base**
	all	all	简单	复杂	all	all	all	简单	复杂	all
Typilus	n/a	54.05	55.12	33.23	60.37	n/a	52.33	52.19	53.91	64.67
Type4Py	n/a	50.34	51.91	32.14	47.51	n/a	32.08	33.47	16.54	29.83
HiTyper	59.20	54.28	57.70	26.44	59.01	45.67	43.54	46.00	19.27	47.99
CodeT5	76.74	78.04	82.43	53.03	82.44	77.83	78.06	85.31	63.41	81.87
TypeT5	**79.24**	**81.43**	**85.69**	**56.75**	**84.82**	**81.75**	**82.95**	**87.62**	**72.78**	**84.17**

表3：罕见**类型**的准确率比较。

	BetterTypes4Py					InferTypes4Py				
	full	**调整后**			**base**	**full**	**调整后**			**base**
	all	all	简单	复杂	all	all	all	简单	复杂	all
Type4Py	n/a	12.37	13.17	4.05	14.15	n/a	0.25	0.14	0.98	0.17
HiTyper	10.30	25.51	27.59	9.79	29.33	9.36	9.36	10.79	1.19	12.33
CodeT5	49.47	52.95	57.28	34.26	57.65	51.64	53.28	59.97	30.62	66.47
TypeT5	**58.56**	**61.47**	**65.21**	**40.22**	**68.44**	**53.44**	**56.27**	**61.50**	**36.92**	**69.23**

在不同程序员之间往往用法不一致。[7] 最后，我们还定义了**基准准确率**指标，它与调整后准确率相同，但只检查最外层类型（例如，`Dict[str,List]`将匹配任何`Dict`，但不会匹配`Mapping`）。

4.2 将 TypeT5 与其他方法进行比较

我们将 TypeT5 与第 3.4 小节所述的基础 CodeT5 模型，以及先前工作中另外三种先进方法的已发布版本进行比较。[8] Typilus（Allamanis 等人，2020年）将程序建模为图，并使用图神经网络计算变量的类型。原始 Typilus 模型既可以从一组常见类型中进行预测，也可以预测（非参数化的）用户定义类型，但其已发布模型只能预测常见类型，因此我们仅评估其在常见类型上的表现。Type4Py（Mir 等人，2022年）使用变量名和周围的代码计算嵌入，并通过在类型嵌入空间中进行最近邻搜索来执行类型预测。我们通过其 Web 界面运行已发布的 Type4Py 模型。HiTyper（Peng 等人，2022年）结合了基于规则的类型推断算法与机器学习类型推断模型的优势：仅在推断算法遇到阻碍时才调用机器学习模型，并在其他位置利用类型约束推导类型。其实现默认使用 Type4Py 作为机器学习后端，我们用它来运行 HiTyper。

我们在表2（常见类型）和表3（稀有类型）中展示了各工具的准确率，并在附录中列出合并结果（表6）。由于 HiTyper 只能对约 67% 的所有标签给出预测，因此我们报告其在这个子集上的表现。从表2中，我们可以得出以下观察结果。(1) 我们的 CodeT5 基线模型已经大幅优于所有先前方法，展现了使用序列到序列预训练语言模型的优势。(2) TypeT5 在两个数据集上的调整后准确率分别进一步提高了 3.4% 和 4.9%。(3) Type4Py 在 InferTypes4Py 数据集上的表现明显低于在 BetterTypes4Py 上的表现，这可能是因为 Type4Py 最初是在 BetterTypes4Py 中的一些测试文件上训练的。[9] 查看表6可知，CodeT5 和 TypeT5 在稀有类型上的表现都远超其他方法。此外，对于既稀有又复杂的类型，TypeT5 的提升幅度最大：在两个数据集上都比 CodeT5 提高约 6%，这表明全局信息在这些情况下尤为重要。作为定性分析，我们还在 A.9 小节中展示了 TypeT5 在一个真实示例上的输出。

[7]例如，类型检查器 MyPy 提供了启用隐式 Optional 类型的选项，因此模型无法仅凭未标注类型的代码判断应该输出 Optional[T] 还是 T。[8]由于我们的方法受益于 CodeT5 在 8 种不同编程语言上的大规模预训练，因此我们使用的训练集比以往工作更小，也没有在我们的数据集上重新训练这些已有方法。[9]原始 Type4Py 论文报告的准确率远高于我们在此处测得的结果。我们在 A.7 小节中分析了这一差异。

表 4：不同模型修改方案的性能。所有模型均使用相应的输入重新训练。

修改	准确率	类型错误
无前导文本	64.20	6067
无使用者	71.20	7053
无使用对象	67.25	7332
非增量	72.52	5720
原始（TypeT5）	**73.02**	**5087**

表5：不同解码策略的性能。不同解码策略使用相同的 TypeT5 模型权重。

策略	准确率	类型错误
独立	71.68	6876
随机	71.66	6215
使用者到被使用者	70.67	7415
UseeToUser	72.65	6402
两遍（TypeT5）	**73.02**	**5087**

4.3 TypeT5 的消融实验

接下来，我们将介绍一系列消融实验，评估各种因素对 TypeT5 性能的影响。除了准确率（针对所有类型）外，我们还报告**类型错误计数**，以此估算模型预测的一致性。[10] 请注意，我们只报告与类型一致性直接相关的错误（因为类型检查器还会报告与类型一致性无关的错误，例如未解析的导入语句。关于此指标的更多详情，请参见 A.8 小节。）。

不同组件如何影响模型的性能？ 为了评估每个上下文元素的影响，我们每次移除一个组件，并据此重新训练模型。具体而言，**无前导文本**、**无使用者**和**无使用对象**消融实验分别对应于从模型输入中移除 $\boldsymbol{s}_{\text{pre}}$、$\boldsymbol{s}_{\text{user}}$ 和 $\boldsymbol{s}_{\text{usee}}$ 上下文元素（在第 3.3 小节中介绍）。**非增量**模型不执行迭代解码，并经过训练以基于未标注类型的上下文进行条件化。我们对所有模型使用相同的输入大小限制（4096 个子词令牌），因此不利用某类信息的模型可以为其他类别的信息留出更多空间。我们在表 4 中同时展示了所有类型上的调整后准确率和类型错误计数（其他准确率指标见表 7）。我们可以看到：（1）所有组件都能提升整体准确率，其中前导部分的影响最大；（2）尽管迭代解码方案仅略微提升了整体准确率，但它显著改善了模型的一致性，使类型错误减少了 12%。

不同的解码策略有何比较？ 除了第 3.4 小节介绍的两遍迭代解码策略外，我们还测试了另外四种解码策略：(1) **独立**，对每个元素独立预测其类型签名，不以模型自身的预测为条件（与非增量模型相同，但未重新训练）；(2) **随机**，按随机顺序依次访问每个元素；(3) **使用者到被使用者**，先访问使用者，再访问被使用者；(4)**UseeToUser**，先访问被使用者，再访问使用者。结果见表5（其他准确率指标见表8）。可以看出，与独立策略相比，我们提出的两遍解码方案在准确率和类型错误方面的提升最大；而使用者到被使用者的表现不如独立策略，这表明不佳的解码顺序可能会对模型性能产生负面影响。

4.4 用户引导的交互式解码

与以往工作相比，我们的方法具有一项独特优势：由于模型可以根据之前的类型进行条件预测，因此用户可以通过让模型参考自己所做的任何更正，轻松地进行干预。因此，我们探索了该模型的另一种使用场景：用户交互式地检查模型预测的每个类型签名，并在模型继续进行下一项预测之前做出任何必要的更正。我们使用人工真实标注 $\mathcal{M}^*$ 模拟这一交互过程，并修改用户到用户的解码过程，让模型先预测每个元素的类型 e（与之前相同），然后在满足 $e \in \mathcal{M}^*$ 时，将预测类型 $\boldsymbol{t}_e$ 替换为相应的人工标注 $\mathcal{M}^*(e)$。在 BetterTypes4Py 数据集上，这种交互式解码过程的完全准确率达到 **78.04%**，调整后准确率达到 **79.37%**——这意味着

[10] 准确率并不总是与一致性相关。例如，当我们有 x = y，且 x 和 y 是字符串或 int 的可能性相同时，一致的类型赋值需要确保 x 和 y 始终具有相同的类型，即使这一要求并不会提高期望准确率。

平均而言，用户只需更正模型预测类型中每五个里的一个，就能从头开始为整个完全未标注的项目添加完整注释。请注意，测试集中只有 59% 包含用户类型注释，因此某些错误的类型注释可能无法立即得到更正，并可能传播到其他位置。因此，在交互式的真实场景中，我们可以预期实际准确率会更高。

5 相关工作

深度学习类型推断 大多数已有方法都通过某种形式的类型嵌入匹配来预测用户定义类型。例如，LambdaNet（Wei 等人，2020年）结合图神经网络和指针网络来应对这一挑战。Typilus（Allamanis 等人，2020年）在类型嵌入空间中执行最近邻搜索。然而，这两种方法都无法处理参数化类型所产生的无限类型空间。TypeBert（Jesse 等人，2021年）是首个基于预训练 Transformer 模型的类型推断方法，并且在预测常见类型方面表现出色。随后，Jesse 等人（2022年）通过深度相似性学习改进了 TypeBert，使其能够更好地支持用户定义类型。与我们的工作不同，TypeBert 不使用序列到序列模型，也不会基于静态分析构建上下文。除了使用源代码之外，其他形式的信息也被用于类型预测。TypeWriter（Pradel 等人，2020年）和 HiTyper（Peng 等人，2022年）都将类型检查与推理时搜索相结合，以避免生成类型错误。OptTyper（Pandi 等人，2020年）通过将类型约束转化为连续优化目标来对其进行显式建模。NL2Type（Malik 等人，2019年）则专注于根据代码注释和文档等自然语言信息预测类型。

基于检索的自然语言处理模型与我们通过静态分析增强上下文的做法类似，人们开发了许多方法，为自然语言处理（NLP）中的预训练模型补充上下文。在开放域问答任务中（Chen et al., 2017），REALM（Guu et al., 2020）和 DPR（Karpukhin et al., 2020）等方法可以检索与查询相关的知识；检索增强生成（Lewis et al., 2020）及其扩展方法（例如 Fusion-in-Decoder（Izacard & Grave, 2020））则表明，可以利用这些信息生成更长的输出。RETRO（Borgeaud et al., 2021）和 WebGPT（Nakano et al., 2021）更是将这一做法推向了网络级规模的极致。不过，我们还能够利用基于使用图的静态分析，而文本中并没有与之对应的东西。

与 T5 结合据我们所知，我们将 CodeT5 用于类型预测尚属首创，而 T5（Raffel et al., 2019）已被应用于摘要生成等多种自然语言处理任务。与我们的工作最相似的是将其用于实体链接（Petroni et al., 2021）：这类系统逐个 token 生成实体名称（De Cao et al., 2020），并且能够泛化到 TypeT5 这样的新实体。然而，在代码场景中，每个新上下文都存在临时定义的类型，且所需的上下文线索类型与自然语言场景大不相同。

结构化预测 我们的迭代解码过程可以被视为应用学习策略来执行结构化预测（BakIr 等，2007年；Daum´e 等，2009年）。在这项工作中，我们的训练方案可以被视为行为克隆，因为模型直接以人工真实标注为条件，这可能导致训练时与推理时之间的分布失配。采用更先进的训练方案，例如计划采样（Bengio 等，2015）、DAgger（Ross 等，2011）或强化学习（Sutton 和 Barto，2018），可能有助于进一步提升迭代解码的性能。

6 结论

在这项工作中，我们展示了 TypeT5 是一种将 CodeT5 与轻量级静态分析及一种新的解码方案相结合的系统，能够有效地为无类型代码库预测类型。我们表明，对预训练代码模型进行微调本身就能得到强大的基线，其表现优于以往工作中的系统；此外，借助静态分析融入恰当的上下文，尤其有助于准确预测更具挑战性的案例。我们认为，更大规模的模型还可以进一步提高准确率，但要在罕见且复杂的类型上实现高准确率，它们仍需利用本工作中的方法。

致谢

本工作部分由美国国家科学基金会奖项 CCF-1918889 资助。我们衷心感谢 Miltiadis Allamanis 和 Amir Mir 慷慨地帮助我们配置他们的工具并完成对比实验。我们也感谢 UTOPIA 团队的成员，尤其是 Jocelyn Chen，感谢他们协助管理实验所用的 GPU 服务器。

参考文献

Wasi Ahmad、Saikat Chakraborty、Baishakhi Ray 和 Kai-Wei Chang。面向程序理解与生成的统一预训练。载于计算语言学协会北美分会 *2021* 年会议论文集：人类语言技术，第 2655–2668 页，2021年。

Miltiadis Allamanis。代码机器学习模型中代码重复的不利影响。载于*2019* 年 *ACM SIGPLAN* 编程与软件新思想、新范式与反思国际研讨会论文集，第 143–153 页，2019 年。

Miltiadis Allamanis、Earl T Barr、Soline Ducousso 和 Zheng Gao。Typilus：神经类型提示。载于编程语言设计与实现会议，2020年。

G. BakIr，神经信息处理系统基金会，T. Hofmann、A.J. Smola、B. Sch¨olkopf 和 B. Taskar。《结构化数据预测》。神经信息处理系统进展。麻省理工学院出版社，2007年。ISBN 9780262026178。URL https://books.google. com/books?id=b1EFKUoFF8IC。

Samy Bengio、Oriol Vinyals、Navdeep Jaitly 和 Noam Shazeer。使用循环神经网络进行序列预测的计划采样。载于 C. Cortes、N. Lawrence、D. Lee、M. Sugiyama 和 R. Garnett（编），神经信息处理系统进展，第 28 卷。Curran Associates, Inc.，2015年。URL https://proceedings.neurips.cc/paper/2015/file/ e995f98d56967d946471af29d7bf99f1-Paper.pdf。

Sebastian Borgeaud、Arthur Mensch、Jordan Hoffmann、Trevor Cai、Eliza Rutherford、Katie Millican、George van den Driessche、Jean-Baptiste Lespiau、Bogdan Damoc、Aidan Clark、Diego de Las Casas、Aurelia Guy、Jacob Menick、Roman Ring、Tom Hennigan、Saffron Huang、Loren Maggiore、Chris Jones、Albin Cassirer、Andy Brock、Michela Paganini、Geoffrey Irving、Oriol Vinyals、Simon Osindero、Karen Simonyan、Jack W. Rae、Erich Elsen 和 Laurent Sifre。通过从数万亿个词元中检索来改进语言模型。载于*arXiv*，2021年。URL：https: //arxiv.org/abs/2112.04426。

Danqi Chen、Adam Fisch、Jason Weston 和 Antoine Bordes。阅读维基百科以回答开放领域问题。载于计算语言学协会第*55*届年会论文集（第*1*卷：长篇论文），2017年。URL：https://aclanthology.org/ P17-1171。

Hal Daum´e、John Langford 和 Daniel Marcu。基于搜索的结构化预测。机器学习，75(3)：297–325，2009年。

Nicola DeCao、GautierIzacard、Sebastian Riedel 和 Fabio Petroni。自回归实体检索。载于国际学习表征会议（*ICLR*）论文集，2020年。URL：https://arxiv.org/abs/2010.00904。

Philip Gage。数据压缩新算法。《*C Users Journal*》，12(2)：23–38，1994年。

Bernd Gruner、Tim Sonnekalb、Thomas S Heinze 和 Clemens-Alexander Brust。基于深度学习的类型推断系统的跨领域评估。*arXiv* 预印本 *arXiv:2208.09189*，2022年。

Kelvin Guu、Kenton Lee、Zora Tung、Panupong Pasupat 和 Ming-Wei Chang。REALM：检索增强语言模型预训练。载于第*37*届国际机器学习会议论文集，2020年。

Vincent J Hellendoorn、Christian Bird、Earl T Barr 和 Miltiadis Allamanis。深度学习类型推断。载于 *2018*年第*26*届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集，第152–162页，2018。

Gautier Izacard 和 Edouard Grave。利用段落检索和生成模型进行开放域问答。载于 *arXiv*，2020年。网址：https://arxiv.org/abs/2007. 01282。

Kevin Jesse、Premkumar T Devanbu 和 Toufique Ahmed。学习类型注释：大数据够用吗？载于第 *29* 届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集，第 1483–1486 页，2021年。

Kevin Jesse、Premkumar Devanbu 和 Anand Ashok Sawant。学习预测用户定义类型。*IEEE* 软件工程汇刊，2022年。

Vladimir Karpukhin、Barlas Oguz、Sewon Min、Patrick Lewis、Ledell Wu、Sergey Edunov、Danqi Chen 和 Wen-tau Yih。用于开放域问答的密集段落检索。载于*2020*年自然语言处理实证方法会议论文集（*EMNLP*），2020年。网址：https://aclanthology.org/2020.emnlp-main.550。

Patrick Lewis、Ethan Perez、Aleksandra Piktus、Fabio Petroni、Vladimir Karpukhin、Naman Goyal、Heinrich K¨uttler、Mike Lewis、Wen-tau Yih、Tim Rockt¨aschel、Sebastian Riedel 和 Douwe Kiela。《面向知识密集型自然语言处理任务的检索增强生成》。载于第*34*届神经信息处理系统国际会议论文集，2020年。

Yujia Li、David Choi、Junyoung Chung、Nate Kushman、Julian Schrittwieser、R´emi Leblond、Tom Eccles、James Keeling、FelixGimeno、Agustin Dal Lago 等人。《AlphaCode：具备竞赛水平的代码生成》。*arXiv*预印本 *arXiv:2203.07814*，2022年。

Rabee Sohail Malik、Jibesh Patra 和 Michael Pradel。《NL2Type：从自然语言信息推断JavaScript函数类型》。载于*2019*年 *IEEE/ACM* 第*41*届软件工程国际会议（*ICSE*），第304–315页。IEEE，2019年。

A. M. Mir、E. Latoskinas 和 G. Gousios。Manytypes4py：用于基于机器学习的类型推断的基准Python数据集。载于*IEEE/ACM*第*18*届软件仓库挖掘国际会议（*MSR*），第585–589页。IEEE计算机协会，2021年5月。doi：10.1109/MSR52588. 2021.00079。Amir M Mir、Evaldas Lato˘skinas、Sebastian Proksch 和 Georgios Gousios。Type4py：面向Python的实用型基于深度相似性学习的类型推断。载于第*44*届软件工程国际会议论文集，第2241–2252页，2022年。Reiichiro Nakano、Jacob Hilton、Suchir Balaji、Jeff Wu、Long Ouyang、Christina Kim、Christopher Hesse、Shantanu Jain、Vineet Kosaraju、William Saunders、Xu Jiang、Karl Cobbe、Tyna Eloun- dou、Gretchen Krueger、Kevin Button、Matthew Knight、Benjamin Chess 和 John Schulman。WebGPT：结合人类反馈的浏览器辅助问答。载于*arXiv*，2021年。URL https://arxiv.org/abs/2112.09332。J ohn-Paul Ore、Sebastian Elbaum、Carrick Detweiler 和 Lambros Karkazis。评估类型注释负担。载于第*33*届*ACM/IEEE*自动化软件工程国际会议论文集，第190–201页，2018年。Irene Vlassi Pandi、Earl T Barr、Andrew D Gordon 和 Charles Sutton。Opttyper：通过优化逻辑约束和自然语言约束进行概率类型推断。*arXiv*预印本 *arXiv:2004.00348*，2020年。Sheena Panthaplackel、Pengyu Nie、Milos Gligoric、Junyi Jessy Li 和 Raymond Mooney。学习根据代码变更更新自然语言注释。载于第*58*届计算语言学协会年会论文集，第1853–1868页，2020年。Yun Peng、Cuiyun Gao、Zongjie Li、Bowei Gao、David Lo、Qirun Zhang 和 Michael Lyu。静态推断与深度学习相结合：一种面向Python的混合类型推断方法。载于第*44*届软件工程国际会议论文集，ICSE '22，第2019–2030页，美国纽约州纽约，2022年。计算机协会。ISBN 9781450392211。doi：10. 1145/3510003.3510038。URL https://doi.org/10.1145/3510003.3510038。

Fabio Petroni、Aleksandra Piktus、Angela Fan、Patrick Lewis、Majid Yazdani、Nicola De Cao、James Thorne、Yacine Jernite、Vladimir Karpukhin、Jean Maillard、Vassilis Plachouras、Tim Rockt¨aschel 和 Sebastian Riedel。KILT：知识密集型语言任务基准。载于*2021*年北美计算语言学协会分会会议：人类语言技术论文集，2021年。URL https: //aclanthology.org/2021.naacl-main.200。

Michael Pradel、Georgios Gousios、Jason Liu 和 Satish Chandra。《TypeWriter：通过基于搜索的验证进行神经类型预测》。载于第 *28* 届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集，第 209–220 页，2020年。

Alec Radford、JeffreyWu、Rewon Child、David Luan、Dario Amodei、Ilya Sutskever 等。语言模型是无监督的多任务学习者。

Colin Raffel、Noam Shazeer、Adam Roberts、Katherine Lee、Sharan Narang、Michael Matena、Yanqi Zhou、Wei Li 和 Peter J. Liu。探索统一文本到文本 Transformer 的迁移学习极限。载于*arXiv*，2019年。网址：https://arxiv.org/abs/1910.10683。

Veselin Raychev、Martin Vechev 和 Andreas Krause。从“大规模代码”中预测程序属性。*ACM SIGPLAN* 通讯，50(1):111–124，2015年。

St´ephane Ross、Geoffrey Gordon 和 Drew Bagnell。将模仿学习和结构化预测简化为无悔在线学习。载于第十四届国际人工智能与统计学会议论文集，第 627–635 页。JMLR 研讨会与会议论文集，2011 年。

Mike Schuster 和 Kaisuke Nakajima。日语和韩语语音搜索。载于*2012* 年 *IEEE* 国际声学、语音与信号处理会议（*ICASSP*），第 5149–5152 页。IEEE，2012 年。

Rico Sennrich、Barry Haddow 和 Alexandra Birch。使用子词单元进行低频词的神经机器翻译。载于计算语言学协会第 *54* 届年会论文集（第 *1* 卷：长篇论文），第 1715–1725 页，德国柏林，2016 年 8 月。计算语言学协会。doi：10.18653/v1/P16-1162。URL：https: // aclanthology.org/P16-1162。

Jeremy Siek 和 Walid Taha。《面向对象的渐进式类型》。载于欧洲面向对象编程会议，第 2–27 页。施普林格，2007年。

Richard S Sutton 和 Andrew G Barto。《强化学习：导论》。麻省理工学院出版社，201 8.

Yue Wang、Weishi Wang、Shafiq Joty 和 Steven CH Hoi。CodeT5：面向代码理解与生成的标识符感知型统一预训练编码器-解码器模型。载于*2021*年自然语言处理实证方法会议论文集，第 8696–8708 页，2021年。

Jiayi Wei、Maruth Goyal、Greg Durrett 和 Isil Dillig。LambdaNet：使用图神经网络进行概率类型推断。载于国际学习表征会议，2020年。网址：https://openreview.net/forum?id=Hkx6hANtwH。

Thomas Wolf、Lysandre Debut、Victor Sanh、Julien Chaumond、Clement Delangue、Anthony Moi、Pierric Cistac、Tim Rault、R´emi Louf、Morgan Funtowicz、Joe Davison、Sam Shleifer、Patrick von Platen、Clara Ma、Yacine Jernite、Julien Plu、Canwen Xu、Teven Le Scao、Sylvain Gug- ger、Mariama Drame、Quentin Lhoest 和 Alexander M. Rush。《Transformers：自然语言处理领域的最先进技术》。载于*2020*年自然语言处理实证方法会议：系统演示，第 38–45 页，线上，2020年10月。计算语言学协会。网址：https://www.aclweb.org/anthology/ 2020.emnlp-demos.6。

A 附录

A.1 构建使用图

为了解析项目层级的用户-被使用者关系（在第 3.2 节中使用），我们基于`libcst`库构建了一个自定义静态分析流水线。我们使用`libcst`解析 Python 文件，并借助其工具解析同一文件中的符号引用（也就是说，它会告诉我们某个本地名称是指向当前文件中定义的函数，还是来自某条导入语句等）。然后，我们实现了自定义导入解析逻辑（遵循 Python 的模块规则），并将其与`libcst`结合，以解析当前项目中的所有特定用法。对于语法形式与类属性/方法使用相匹配但无法解析的用法，我们会为每个名称匹配的类成员生成一个潜在用法。由于所涉及的静态分析操作相当轻量，而且我们在中央处理器上并行构建不同的图，因此构建使用图所花费的时间仅占训练或推理总时间的一小部分。例如，使用 28 个 Python 进程处理整个 BetterTypes4Py 训练集（包括解析、代码转换和分词等其他任务所花费的时间）只需约 8 分钟。

虽然我们的实现将分析范围限制在当前项目内，但将分析扩展到也包含涉及库定义的用法并不困难，这或许有助于模型了解不常见库 API 的类型签名。在我们的实验中，我们没有这样做，主要是因为为数据集中的所有项目安装正确的库依赖需要投入大量人工工作。

A.2 构建 BetterTypes4Py 数据集

我们的数据集由 ManyTypes4Py 数据集按以下步骤构建：首先，我们筛除已无法访问或无法在 10 秒内下载的 GitHub 代码仓库。这样便从 5996 个原始项目中保留了 4890 个。接着，我们舍弃超过一年未更新的项目（以避免使用过时的库 API），代码仓库数量因此减少至 1218 个。为限制任何特定项目的影响，我们还筛除了代码行数超过 50K 的 37 个项目。最后，为排除类型注释过少的项目，我们将每个项目的类型注释数量 n_t 与其代码行数 n_c 进行比较，并筛除 n_t-to-n_c 比率低于 1:20 的项目。由此得到最终的 663 个项目。随后，我们随机选取 50 个测试集项目和 40 个验证项目，其余项目用于训练。

A.3 代码重复

代码重复（Allamanis，2019）已被证明会对机器学习模型的性能产生不利影响，并可能模糊评估结果。因此，先前的工作（如 Type4Py）会采用文件级去重来移除重复的源文件。然而，在我们以项目为单位的设置中，这很难实现，因为推理期间需要所有文件都存在。为了解决这一问题，我们已手动核实 InferTypes4Py 数据集不包含从其他位置复制粘贴而来的文件。我们还在测试集上运行了流行的代码重复检测工具 `jscpd`[11]，以检测重复的代码块。分析结果表明，数据集中的重复相对较少（重复行仅约占 4%），且其中大多数重复内容来自同一项目，而非跨项目，因此我们认为，在按项目进行评估时，代码重复并非主要问题。

A.4 标签质量

在我们的评估中，开发者提供的类型注释被用作真实标注，但它们并非总是准确或一致（Ore 等人，2018）。这在一定程度上促使我们构建 InferTypes4Py 数据集，其中包含高质量、错误率相对较低的类型注释。具体而言，我们自己的代码库（已纳入 InferTypes4Py）在整个开发过程中大量使用类型注释，并持续通过 VSCode 进行类型检查。作为粗略估算标签质量的指标，我们报告了两个数据集中的人工标签一致性错误（定义见 A.8 小节）：在 BetterTypes4Py 和 InferTypes4Py 中，每个人工标注的平均一致性错误分别为 0.019 和 0.045。

[11] https://github.com/kucherenko/jscpd

表 6：所有类型（**常见 + 罕见**）的准确率比较。

	BetterTypes4Py					InferTypes4Py				
	full	**调整后**			**base**	**full**	**调整后**			**base**
	all	all	简单	复杂	all	all	all	简单	复杂	all
Type4Py	n/a	34.52	35.87	19.68	35.61	n/a	21.11	22.35	9.61	22.64
HiTyper	42.53	41.95	44.86	19.02	47.88	34.83	32.50	35.11	11.40	39.23
CodeT5	67.07	67.47	72.12	44.05	73.44	68.80	68.95	75.14	54.04	77.78
TypeT5	**71.89**	**73.02**	**77.07**	**49.72**	**78.87**	**71.98**	**73.15**	**77.16**	**62.66**	**80.20**

A.5 类型规范化

为计算第 4.1 小节中的准确率指标，我们递归地应用以下步骤来规范化 Python 类型：

1. 将任意 `Optional[T]`重写为 `Union[T,None]`。2. 对 `Union` 类型的参数排序，并展平任何嵌套的`Union`。例如，将 `Union[B,Union[C,A]]`重写为 `Union[A,B,C]`。3. 如果所有类型参数都是 `Any`，则将它们全部省略。例如，将`List[Any]` 重写为 `List`。4. 将基本类型的名称首字母大写。例如，将`list` 重写为 `List`。

A.6 超参数与运行时间

我们使用 Huggingface Transformers 库提供的 CodeT5 模型初始化模型权重（Wolf 等，2020年），并使用该库的默认优化器配置（AdamW，基准学习率为 2e-5，权重衰减为 0.01）训练模型，恰好训练一轮。在推理期间，我们使用束宽度为 16、多样性惩罚为 1.0 的束搜索。我们将最大输出序列长度自适应地设为 $16n + 10$，其中 n 是输入中待预测的类型数量。

在训练和测试期间，我们分别将前导部分、usee 上下文、主代码和用户上下文的大小限制设为 1000、2048、512 和 1536。请注意，前导部分占用 usee 上下文的空间，因此输入总大小上限为 4096。

在配备 48GB 内存的单张 Quadro RTX 8000 图形处理器上，训练模型耗时约 11 小时。在 BetterTypes4Py 测试集上执行两遍推理约需 4 小时，而执行单遍推理（例如 UseeToUser）则耗时约为一半。作为对比，在同一台机器上训练 CodeT5 模型约需 3.7 小时，相应的评估约需 0.5 小时。

A.7 WHYTYPE4PY 在我们的数据集上的性能为何低得多

我们与 Type4Py 的作者讨论了这些实验，并认为我们观察到的差异很可能是由两个原因共同造成的：首先，我们的评估只统计所有顶层 API 的类型注释，而 Mir 等人（2022年）的评估还包括所有局部变量。其次，我们只评估人工标注，而他们还将机器推断的类型注释（通过类型检查器 Pyre 获得）作为真实标签。因此，两篇论文报告的标签分布显著不同：在我们的设置中，函数注释（参数 + 返回类型）占标签的大多数（87%）；而在 Mir 等人（2022年）的研究中，这一比例要小得多，仅为 18%。这表明，他们的标签集可能被类型检查器能够推断出的简单标签所膨胀，这也解释了为何在我们的数据集上评估时性能会下降。

A.8 使用类型错误衡量类型一致性

为评估类型一致性（第 4.3 小节），我们在用模型 predict 的类型进行标注的代码库上运行类型检查器`MyPy`[12] 。由于`MyPy` 报告的错误并非全部都是类型错误或与类型一致性相关，因此我们只统计以下 5 种错误代码：根据`MyPy`文档，它们的含义如下：

[12]http://mypy-lang .org/.

表 7：BetterTypes4Py 上不同模型修改的详细比较。所有模型均使用相应的输入格式重新训练。准确率在所有类型上测量。

修改	**full** 准确率 all	**调整后** 准确率 all	简单	复杂	**base** 准确率 all	类型错误
无前导文本	63.03	64.20	71.22	33.51	69.98	6067
无使用对象	67.70	67.15	70.27	48.09	73.01	7053
无使用者	70.22	71.20	**77.87**	41.34	77.26	7332
非增量	71.86	72.52	77.23	47.26	78.47	5720
原始（TypeT5）	**71.89**	**73.02**	77.07	**49.72**	**78.87**	**5087**

表8：BetterTypes4Py 上不同解码策略的详细比较。不同解码策略使用相同的 TypeT5 模型权重。准确率在所有类型上进行测量，用户引导解码见第 4.4 小节。

修改	**full** 准确率 all	**调整后** 准确率 all	简单	复杂	**base** 准确率 all	类型错误
独立	70.87	71.68	75.99	46.92	77.58	6876
随机	70.68	71.66	75.68	47.89	77.65	6215
使用者到被使用者	69.98	70.67	74.48	47.60	76.46	7415
UseeToUser	71.58	72.65	76.80	48.88	78.48	6402
两遍（TypeT5）	**71.89**	**73.02**	**77.07**	**49.72**	**78.87**	**5087**
用户引导	78.09	79.36	83.27	58.61	84.02	n/a

错误，或者与类型一致性相关的错误，因此我们只统计以下 5 种错误代码：根据`MyPy`文档，它们的含义如下：

- **`attr-defined`** 用于检查使用点运算符时，目标类或模块中是否定义了某个属性。
- **`arg-type`** 用于检查调用中的参数类型是否与被调用函数签名中声明的参数类型相符。
- **`return-value`** 用于检查返回值是否与函数的类型签名兼容。
- **`assignment`**用于检查被赋值表达式是否与赋值目标兼容。
- **`name-defined`** 用于检查名称是否已在当前作用域中定义。

请注意，这项指标确实有其局限性。我们发现，它有一个不理想的特性：相比预测错误类型，它可能更倾向于预测不存在的类型。例如，当模型为某个函数预测了错误类型时，类型检查器会根据这个声明类型检查该函数的所有用法，因此很可能报告多个错误。然而，如果模型预测了一个不存在的类型，类型检查器只会在声明位置报告一个`name-defined` 错误，并跳过对其用法的检查。这一影响使表 4 中的无前导内容变体的错误计数低于另外两个变体，因为它往往会预测出更多不存在的类型。但我们已验证，这并非我们的两遍迭代解码模型能够降低类型错误计数的原因。

A.9 TypeT5 生成的真实示例

我们在图 1 所示示例对应的实际代码上运行 TypeT5，并展示针对`ModelWrapper.predict` 元素得到的前导部分（图 5）、usee 上下文（图 6）、主代码（图 7）和用户上下文（图 8）。对于模型预测的每种类型，我们用绿色或红色标记表示其是否正确；若不正确，还会用红色显示真实类型。请注意，这些输出由迭代解码过程第二遍生成（详见第 3.4 小节），因此所有元素都至少已被标注过一次（但有些元素可能在第二遍中未被标注）。

model_input.py **Preamble**

```python
import random
from copy import copy, deepcopy
import numpy as np
from collections import Counter
from mypy_extensions import mypyc_attr
from torch import Tensor
from transformers.data.data_collator import DataCollatorForSeq2Seq
from typing import NamedTuple, overload
from datasets.arrow_dataset import Dataset
from torch.utils.data import DataLoader, RandomSampler
from.data import (
    ChunkedDataset,
    CtxArgs,
    TokenizedSrcSet,
    output_ids_as_types,
    preds_to_accuracies,
)
from.type_env import AccuracyMetric, PythonType
from.utils import *
@dataclass
class DecodingArgs:
    ...
@dataclass
class DatasetPredResult(Generic[T1]):
    ...
@dataclass
class ModelWrapper:
    ...
```

图 5：前导部分汇集了当前文件中的所有重要语句和类头部。这有助于模型了解有哪些类型可用，以及每个符号来自何处。

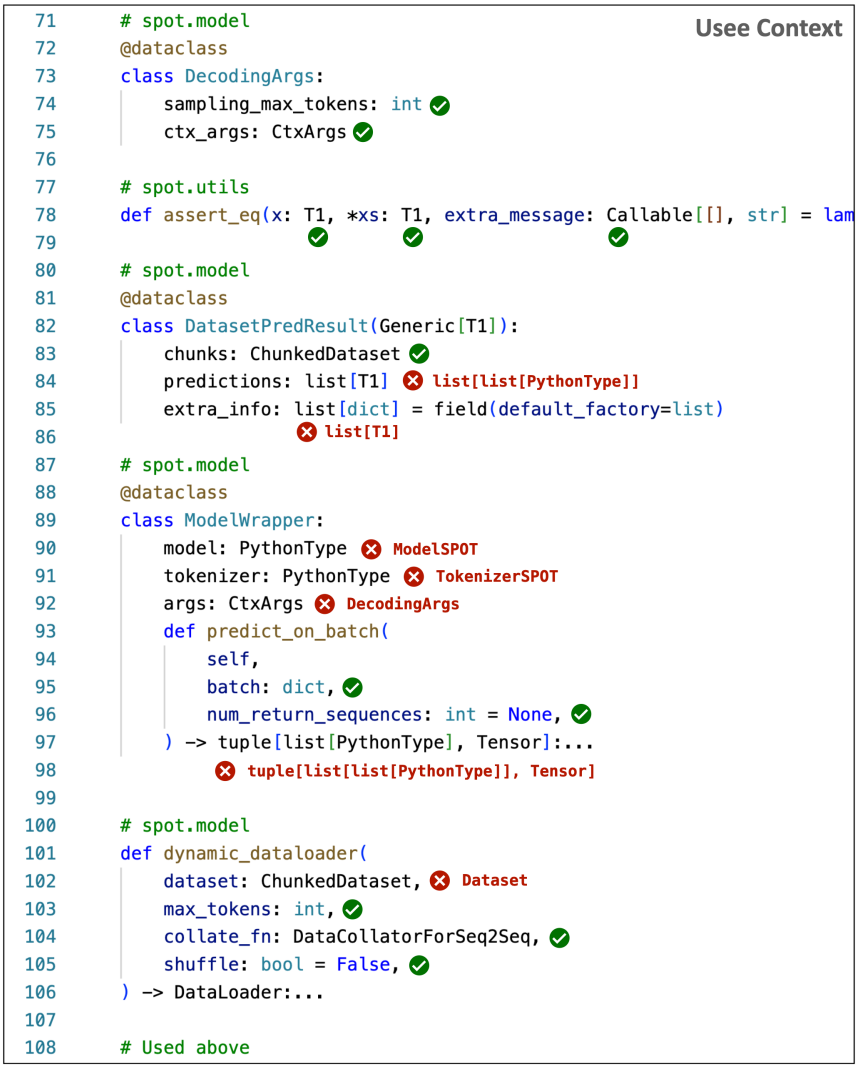

图 6：usee 上下文展示主代码或用户上下文中的元素所使用的元素的签名。通过查看它们预测的类型签名，模型无需深入了解这些定义的实现，就能理解其类型级行为。

Main Code

```python
# spot.model
@dataclass
class ModelWrapper:
    def predict(
        self,
        dataset: ChunkedDataset,  Dataset
        tqdm_args: dict = {},
        num_return_sequences: int = None,
    ) -> list[PythonType]:  list[list[PythonType]]
        model = self.model
        collator = DataCollatorForSeq2Seq(self.tokenizer, model)
        loader = dynamic_dataloader(
            dataset,
            max_tokens=self.args.sampling_max_tokens,
            collate_fn=collator,
            shuffle=True,
        )
        device = model.device
        pred_types = dict[int, list]()
        with tqdm(
            total=len(dataset), desc="predict", smoothing=0.01, **tqdm_args
        ) as tqdm_bar:
            for batch in loader:
                n_chunks = batch["input_ids"].shape[0]
                batch["input_ids"] = batch["input_ids"].to(device)
                preds, _ = self.predict_on_batch(batch, num_return_sequences)
                for i, c_id in enumerate(batch["chunk_id"]):
                    c_id = int(c_id)
                    if num_return_sequences is None:
                        pred_types[c_id] = preds[i]
                    else:
                        pred_types[c_id] = preds[
                            i * num_return_sequences : (i + 1) * num_return_seque
                        ]
                tqdm_bar.update(n_chunks)
        return [pred_types[int(c_id)] for c_id in dataset["chunk_id"]]

```

图 7：主代码是模型在当前解码步骤中进行标注的元素。模型在此示例中犯了两个错误，这两个错误都可直接归因于 usee 上下文中的前两个错误（第 102 行和第 97 行）。这表明模型会根据上下文做出连贯的预测；如果用户更正了先前的错误（如第 4.4 小节所述），就可以避免此类错误。

User Context, first half

```python
# Users below
# spot.model
@dataclass
class ModelWrapper:
    def eval_on_dataset(
        self,
        src_data: TokenizedSrcSet,
        max_labels: int = None,
        tqdm_args: dict = {},
    ) -> DatasetPredResult:
        ctx_args = self.args.ctx_args
        if max_labels is not None:
            ctx_args = copy(ctx_args)
            ctx_args.max_labels = max_labels

        chunks = src_data.to_chunks(ctx_args, tqdm_args=tqdm_args)
        preds = self.predict(
            chunks.data, num_return_sequences=None, tqdm_args=tqdm_args
        )
        return DatasetPredResult(chunks, preds)

```

User Context, second half

```python
# spot.decode
def sample_candidates(
    wrapper: ModelWrapper,
    src_data: TokenizedSrcSet,
    n_samples: int,                tuple[ChunkedDataset, list[list[list[PythonType]]]]
) -> tuple[ChunkedDataset, list[PythonType]]:
    ctx_args = wrapper.args.ctx_args
    do_sample = wrapper.args.do_sample
    if not do_sample:
        assert wrapper.args.num_beams is not None, "num_beams needs to be set"
        assert n_samples <= wrapper.args.num_beams

    chunks = src_data.to_chunks(ctx_args)
    n_chunks = len(chunks.data)

    if do_sample:
        samples = [
            wrapper.predict(chunks.data, tqdm_args={})
            for _ in tqdm(range(n_samples), desc="Sampling")
        ]
    else:…

    def get_preds(chunk_id, sample_id):
        return (
            samples[sample_id][chunk_id] if do_sample else samples[chunk_id][sampl
        )

    pred_candidates = [
        [get_preds(cid, sid) for sid in range(n_samples)] for cid in range(n_chunk
    ]
    return chunks, pred_candidates

```

图 8：用户上下文展示了主代码中 `predict` 方法的两个调用方。我们可以看到，尽管这些用户定义类型都是当前项目中新定义的类，模型仍成功预测了所有用户定义类型。