渐进式类型中用于性能提升的类型注解高效选择

Senxi Li[a] ,Feng Dai[a] , etsuro Yamazaki[a] 和higeru Chiba[a]a 东京
大学，日本

摘要 渐进式类型作为一种在单一语言中整合静态类型和动态类型的设计选择，日益受到青睐。多种实用语言已采用渐进式类型，使程序员能够灵活地根据需要为程序添加类型注解。与此同时，部分类型化程序中意外的性能下降仍是一项关键挑战。仅仅增加更多类型注解，就可能导致执行速度显著降低。以往研究探索了通过选择性添加类型注解来提升性能的策略。然而，这些策略受限于较长的编译时间，妨碍了其实际应用。

本文提出一种新技术，用于从类型推断得出的类型注解中选取一个子集，以提升渐进类型程序的执行性能。该方法的优点是采用轻量级摊销方法，从而缩短编译时间。它沿着数据流选择类型注解，预期可避免因值在无类型代码和有类型代码的边界之间反复传递而产生的高昂运行时类型转换。

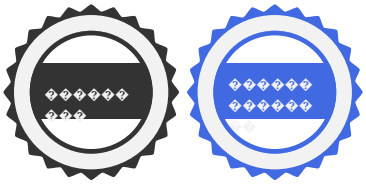
我们展示了该提案的适用性，并通过实验验证其在 Reticulated Python 上提升执行时间方面的有效性。我们的实现支持 Python 子集，用于选择由所实现的外部类型推断引擎推导出的类型注解。实验结果表明，在基准程序中，我们的提案优于使用类型推断所推导的全部类型注解这一朴素策略。与现有方法相比，该提案实现了相当的执行速度，并且在维护推导和选择类型注解时更稳定的编译时间方面展现出优势。我们的实验结果表明，所提出的技术在 Reticulated Python 中切实可行，可用于缓解渐进类型程序的性能瓶颈。

ACM CCS 2012

- **软件及其工程** → **软件设计工程**；**运行时环境**；**计算理论** → 类型理论；关键词 渐进式类型、性能优化、静态分析、类型推断

编程的艺术、科学与工程

提交 2025年10月2日发表 2026年2月15日DOI
10.22152/programming-journal.org/2026/11/3





© Senxi Li、Feng Dai、Tetsuro Yamazaki 和 Shigeru Chiba 本作品采用“知识共享署名 4.0 国际许可协议”授权。载于《编程的艺术、科学与工程》，第 11 卷第 1 期，2026 年，文章 3；共 35 页。

1 引言

渐进式类型为动态类型语言与静态类型语言之间提供了灵活的桥梁，同时也带来了执行性能下降这一严峻挑战。在渐进类型语言中，程序员可以在程序中添加类型注解，以便在需要时使用静态类型。为确保健全性，即使用类型注解 T 标注的变量只能被赋予类型为 T 的值，必须在动态（无类型）代码与静态类型代码的边界之间执行运行时类型转换。执行性能下降通常是由这些运行时类型转换造成的；它们会检查值的类型是否符合预期，否则就会引发错误。在具有非擦除语义的健全渐进式类型语言中，运行时类型转换是强制执行的 [10, 13]。健全的渐进式类型已应用于多种实用语言，包括 Racket [22, 42], 、Python [43] 和 ActionScript [37]。

提升执行性能的一种常见方法是通过类型推断自动添加附加类型注解，使更多变量获得静态类型。然而，不加审慎地添加类型推断所建议的所有类型注解，可能会意外导致执行性能下降。这是因为添加某些类型注解可能会使值反复跨越无类型代码和有类型代码之间的边界，从而引入额外的运行时类型转换。例如，当一个无类型的值被赋给变量 v，且 v 是无类型函数的一个参数时，为 v 添加附加类型注解，会使该值先从无类型代码跨入有类型代码，再返回无类型代码。另一种常见方法是利用即时编译（JIT）避免运行时多余的类型转换。尽管 JIT 一直是提升渐进式类型性能的有效方法 [3, 38, 39],，但对于资源充足的系统以外的场景，其适用性有限，因为 JIT 动态编译需要额外的内存。例如，MicroPython [11] 等微控制器以及其他嵌入式系统常因内存受限而不支持 JIT。

研究表明，天真地添加更多类型注解并不能可靠地提升性能。尤其是，Takikawa 等人指出，程序在部分类型化时会出现不可忽视的性能下降 [41]。他们的研究显示，除非程序完全采用静态类型，否则与无类型程序相比，即使程序大部分已类型化，执行速度也会显著下降，甚至超过 100 倍。这一性能陷阱会阻碍渐进式类型的使用，因为一种简单的规避方法就是直接删除所有类型注解。

已有一些研究探索了如何从类型推断得到的类型注解中选择要添加的类型注解，以提升渐进类型程序的性能。然而，这些方法在推导和选择类型注解时可能会产生不可忽视的编译时间开销。例如，我们观察到，现有工具 Herder [6] 在处理若干基准程序时会产生大量编译时间，例如超过 10 分钟，尽管与真实程序相比，这些程序的代码行数相对较少。在考虑实际使用的情况下，如何选择能够提升性能的有效类型注解仍然具有挑战性。

我们提出了一种名为 *TypePycker* 的轻量级方法，用于有选择地添加类型注解，以改善渐进类型程序的执行时间。与现有方法相比，TypePycker 的新颖之处在于，它采用轻量级方法，在无需长时间编译的情况下实现了相当的执行性能。给定一个程序，它会执行类型推断，并仅选择由类型推断得到的类型注解的一个子集。为避免值在无类型代码和有类型代码之间反复跨越边界而导致运行时类型转换的显著开销，TypePycker 使用一种快速的摊销方法，沿着数据流选择要放置的类型注解。

为评估所提方法的有效性，我们在渐进式类型语言 Reticulated Python [43] 的一组收集程序上开展实验。实验结果表明，与朴素地使用由类型推断得到的全部推断类型相比，所提方法实现了性能提升。在收集的 41 个基准程序中，我们的方法在 32 个程序上优于使用全部推断类型注解。与附加全部推断类型注解的程序相比，其加速比最高可达 5 倍以上。在 6 个程序中，我们的方法给出了与使用全部推断类型注解相当的执行时间。为评估编译时间，我们将 TypePycker 与现有工具 Herder [6] 进行比较。我们观察到，TypePycker 保持了更稳定且可接受的编译时间，并在各基准程序上实现了相当的执行时间。我们还发现，TypePycker 在若干程序上具有明显优势：其编译时间远快于 Herder，同时我们的选择所带来的性能也优于使用全部推断类型。研究结果表明，该方案是一种切实可行的方法，可缓解 Reticulated Python 中渐进类型程序的性能下降。

本文其余部分安排如下。第 2 节通过介绍渐进式类型中的性能挑战，阐明本文的研究动机。第 3 节介绍所提出的方法。第 4 节介绍我们的实验。第 5 节讨论有效性威胁。第 6 节回顾相关工作，并以简要结论结束本文。

2 渐进类型程序中的性能下降

预测渐进类型程序的执行性能颇具挑战。原因在于，当值跨越无类型代码和有类型代码之间的边界时，渐进类型语言会执行运行时类型转换。这通常会导致执行性能严重下降。运行时类型转换是在运行时执行的一项操作，用于检查某个值是否具有预期类型。执行运行时类型转换是为了确保程序 [7, 37, 40]所指定的类型不变量。许多渐进式类型语言都会执行运行时类型转换，以保证静态类型的健全性 [22, 33, 43]。多项研究观察到，运行时类型转换会造成显著的运行时开销；与无类型版本相比，部分类型化程序的运行速度可能慢 100 倍以上 [15,41]。

考虑清单 1 中给出的一个渐进类型语言程序。程序员为局部变量z 添加了类型注解。局部变量、

函数参数或未添加类型注解的返回类型会被隐式标记为未知类型*。我们假设succ 是一个类型为Function([*],*)的库函数，表示一个函数类型，其参数类型为* ，返回类型为*。

清单1 示例程序。succ 假定其类型为Function([*], *)。* 表示未知类型。

```
def f(x, y): u  = x z: Bool  = y 返回 如果
f(succ(1), 真)
```

示例程序中的运行时类型转换如清单 2 所示。请注意，该语言会为1 和真这样的字面量隐式赋予静态类型。因此，即使程序不包含任何显式类型注解，它也会成为部分类型化程序，并涉及运行时类型转换。为便于说明，我们使用记法{T_2⇐ T_1} e 来显式表示发生了运行时类型转换。其含义是：表达式e 在编译时的类型为T_1 ，而在运行时e 预期具有类型T_2 。例如，第 12 行发生了运行时类型转换{Bool ⇐*}y ，因为y 被隐式赋予类型*，但它被赋给了一个类型为Bool的局部变量。该运行时类型转换会在运行时执行，以检查y 是否为布尔值，从而产生运行时开销。

清单 2 带有运行时类型转换的示例程序。succ 假定其类型为 a s 函数([*], *)。

```
def f(x, y): u  =x z: Bool  = {Bool
⇐'
(u)
```

一种广泛使用的渐进式类型执行速度下降缓解方法，是通过类型推断自动添加附加类型注解，以消除部分运行时类型转换。假设运行类型推断引擎，为清单 1 中的示例程序推导局部变量、函数参数和返回类型的附加类型注解。它推断出参数 y 的类型为Bool。这一附加类型注解预计能提升执行速度，因为它可以消除清单 2 第 15 行的运行时类型转换 {*⇐ Bool}y ， 以及第 12 行的运行时类型转换 {Bool ⇐*}y 。

然而，若不加审慎地按照类型推断引擎的建议添加类型注解，可能会导致性能下降，这与程序员的直觉预期相悖。假设在清单 1 中为参数x 额外添加类型注解 I nt，则运行时类型转换将发生如下：

```
def f(x: Int, y): u ={* ⇐ Int}x z: Bool  = {Bool
⇐
回
f({整型 ⇐*}后继({* ⇐整型}1), {* ⇐ Bool}真)
```

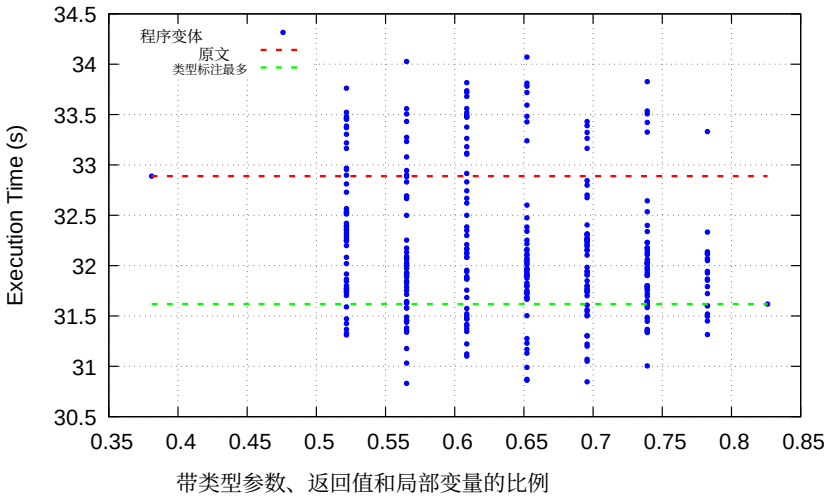
与清单 2 中展示的运行时类型转换相比，由于一个值反复跨越无类型代码 succ(1) 到有类型代码 x 的边界，然后又返回无类型代码 u 以及succ(x)中的参数，因此会额外发生三次运行时类型转换。请记住，假定 succ 的类型为 Function([*],*)。

过度类型注解所导致的意外性能下降已为现有研究所熟知。例如，Takikawa 等人的开创性论文 [41], 表明，除非程序完全采用静态类型，否则即使程序大部分已标注类型，执行时间仍可能较慢。我们还观察到，仅仅增加更多由类型推断得到的附加类型注解，并不一定会带来更好的性能。作为初步研究，我们进行了与 Takikawa 等人相似的实验。在初步研究中，我们在 Reticulated Python [43]上测量了若干带有不同数量类型注解的公开 Python 程序的执行时间。Reticulated Python 是 Python 的一种渐进类型方言，允许程序员使用静态类型为程序添加注解。它会为健全的静态类型执行运行时类型转换。Given 一个 Python 程序，我们为其局部变量、函数参数和返回类型给出一组类型注解。这些类型注解由外部类型推断引擎推导得出。请注意，我们修改了 Reticulated Python，以支持局部变量的静态类型。原始的 Reticulated Python 不支持局部变量的显式类型注解。随后，我们生成使用这些类型注解的随机子集进行标注的程序变体。我们测量了类型注解数量为零的程序、包含由类型推断得到的全部类型注解的程序，以及带有这些类型注解不同子集的所生成程序变体的执行时间。

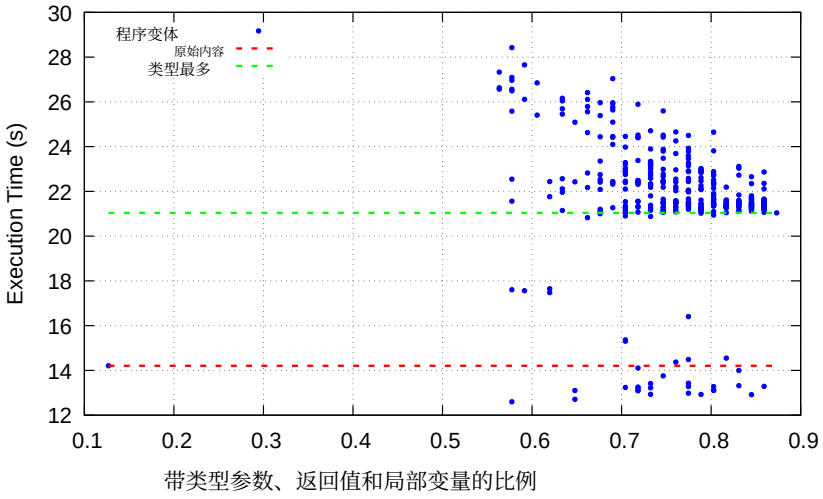
图 1 展示了两个受测程序的结果。[1] 对于图 1a 中的程序，相较于完全没有类型注解的程序，包含了类型推断得出的全部类型注解的程序性能更好。这说明添加类型注解可以提升性能。不过，也可以观察到，使用更少的类型注解还能进一步提升性能。对于图 1b 中的程序，尽管类型推断为 80% 以上的变量等生成了类型注解，但包含全部这些注解的程序表现不如完全没有注解的程序。舍弃部分类型注解可以缩短执行时间。

一个有趣的研究方向是开发技术，从类型推断得出的类型注解中选出合适的注解，以提升执行性能。然而，现有方法存在局限。一项研究 [6] 通过分析不同类型注解子集所带来的运行时类型转换开销，来推荐合适的类型注解。该方法的缺点是，对于若干基准程序（包括真实世界的程序），需要耗费极长的编译时间。本文所说的编译时间，是指通过类型推断得出附加类型注解并将选定的类型注解插入原始程序所需的时间。另一项近期研究 [21] 尝试通过基于机器学习的方法预测最优执行性能，以解决这一问题。然而，

[1]其余所有结果均包含在我们的制品中。



(a) 存储



(b) nbody

图 1 Python 程序变体的执行时间。每个蓝色圆点是一个生成的程序变体，具有不同的类型注解子集。X 轴显示带类型函数参数、返回值和局部变量的比例。Y 轴显示以秒为单位的执行时间（越低越好）。红色虚线表示类型注解数量为零的程序。绿色虚线表示包含由类型推断得出的全部类型注解的程序。

这需要执行多个带有不同类型注解子集的程序来进行训练。因此，考虑实际使用情况来选择类型注解候选项仍然颇具挑战。

3 高效选择新增的类型注解

我们提出了 TypePycker，这是一种新的轻量级方法，通过有选择地添加类型注解来提升渐进类型程序的执行时间。我们的方法首先执行类型推断，然后仅选择类型推断得出的部分类型注解，使得包含所选类型注解的程序比添加类型注解之前运行得更快。与以往的方法不同，TypePycker 十分轻量，因此无需耗费很长时间进行选择。尽管方法轻量，根据我们的实验结果，其带来的性能提升与现有方法相当，在某些情况下甚至优于现有工具。

我们的核心思路是根据类型注解在数据流中的位置来选择注解。当一个值反复从有类型变量传递到无类型（或类型未知）变量，再传回有类型变量时，运行时类型转换的开销往往会增加。因此，当类型推断为所有能够沿数据流到达 v 的其他无类型变量（和函数调用）推断出类型时，TypePycker 就会选择为变量 v添加推断出的类型注解。我们采用轻量级摊销方法快速检查这一条件。该方法不会反复对数据流进行开销高昂的遍历，也不会以蛮力方式枚举所有可能发生运行时类型转换的位置。

3.1 SimpliPy

为了介绍所提出的方法，我们首先定义一种名为*SimpliPy*的小型渐进类型语言。SimpliPy 是 Python 的简化示例版本。为简洁起见，我们将使用 SimpliPy 来介绍这一方案。

代码清单3 语言语法

```
程序: =语句
语句: = 表达式 |返回语句 |定义语句
        | 变量 = 表达式 |表达式[表达式] = 表达式
        | stmt; stmt
returnStmt: = 返回表达式
defStmt = def id(param, ...) -> T: stmt
变量, 参数: = 标识符: T
表达式: = 表达式 +表达式 |表达式(表达式, ...)
        | 如果 表达式 则 表达式 否则 表达式
        | [表达式, ...] |表达式[表达式]
        | 标识符 |数字 |真 | 假

T := * |Bool |整型 |数组(T) | 函数([T, ...], T)
```

SimpliPy 的语法见代码清单3。程序是一系列语句，表达式支持加法、数组、条件和函数应用。

类型可以是布尔类型、整数类型、函数、数组或未知类型。未知类型用 * 表示。未知类型的表达式通常称为 无类型表达式。[expr, ...] 和 (expr,...) 分别表示由零个或多个 expr 实例组成的序列，这些实例以逗号分隔，并分别置于方括号 [] 和 () 中。[expr,...] 是数组字面量。在 SimpliPy 中，程序员不得省略类型注解。相反，他们会使用类型注解 :* 明确指定未知类型。不过，当用具体类型注解 : Int 替换 :* 时，为便于阅读，下文将其称为向无类型变量或参数添加或附加类型注解。

3.2 图构建

所提出的方法首先为给定的 SimpliPy 程序构建一个有向图。该图表示可能影响渐进式类型运行时类型转换的数据流。我们根据给定程序的语法静态构建该图。此外，该方法假设类型推断引擎会为给定程序生成推断类型。

一个顶点表示变量、函数参数、函数名、字面量或表达式。当一个表达式包含子表达式时，所有表达式都由顶点表示。例如，表达式 x + y 会为x、y和x + y生成三个顶点。表达式 a[i] 会为a、i和a[i]生成三个顶点。

每个不同的变量、参数或函数名都对应一个唯一的顶点，无论它在程序中出现多少次。如果这些不同的出现形式在程序中由静态名称解析确定的名称作用域绑定到同一个变量、参数或函数名，它们都由同一个顶点表示。此外，每个函数定义都会生成一个用于表示return 语句的顶点。即使一个函数定义包含多个return 语句，也只会生成一个顶点。

每个顶点都有两个属性。其中一个属性是给定类型。我们将顶点 v 的给定类型表示为$given(v)$。如果一个顶点表示变量或函数参数，其给定类型就是程序员所写类型注解指定的类型。例如，对于 x: Int = 3，表示 x 的顶点的给定类型是整型。对于 x: * = 3，表示 x 的顶点的给定类型是*。如果一个顶点表示返回语句，其给定类型就是函数定义的类型注解所指定的返回类型。如果一个顶点表示函数名，其给定类型就是其类型注解所指定的函数类型。如果一个顶点表示数字或布尔字面量，那么其给定类型分别是整型 和Bool。如果一个顶点表示空数组字面量 [], ，其给定类型是Array(*)。如果一个顶点表示非空数组字面量，例如 [x,y],，其给定类型是Array(*)。其他顶点的给定类型均为*。请注意，有些顶点（例如表示数字字面量的顶点）并不对应程序员所写的类型注解，但它们的给定类型并非*。

顶点的另一个属性是推断类型。我们将顶点v 的推断类型表示为$infer(v)$。我们假设外部类型推断引擎已经运行，其生成的推断类型反映在图中。例如，当表达式 f(x) 的类型被推断为Int时，表示该函数应用的顶点的推断类型就是Int 。类型推断引擎还可以推断

其他类型表达式的类型。这些类型被用作与这些表达式对应的顶点的推断类型。如果类型推断引擎未推断出任何类型，则将未知类型* 视为推断类型。表示return 语句的顶点的推断类型，是其函数定义的返回类型所推断出的类型。表示函数名的顶点的推断类型，是为该函数推断出的函数类型。数字字面量和布尔字面量的推断类型是它们的值类型，即Int 和Bool。我们假设*infer(v)* 比*given(v)*更具体，这意味着如果*given(v)*中至少有一个* ，则在*infer(v)*中会将其中的该* 替换为具体类型（如果存在），因为类型推断引擎在计算*infer(v)*时会考虑*given(v)* 。例如，如果类型推断引擎无法为*v*推导出额外的类型信息，那么*infer(v)* 将等价于*given(v)*。

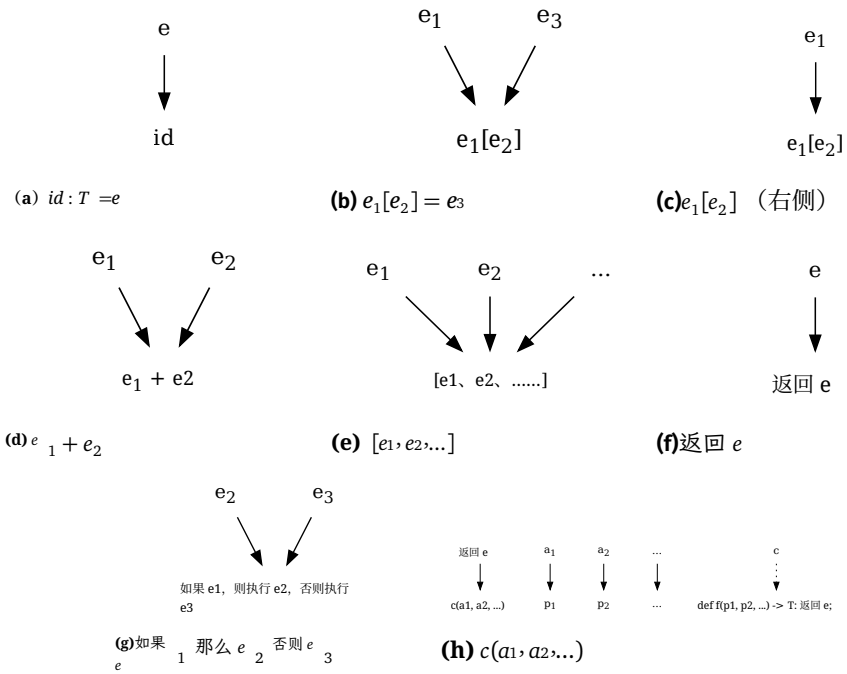


图2 边创建。实线箭头表示图边。虚线箭头表示通过指向分析发现的被调用函数。

创建有向边来表示程序中的数据流。图 2 展示了有向边的创建方式。边创建基本遵循数据流分析中的标准做法。对于函数应用 $c(a_1, a_2, ...)$，会根据 c 可能指向的被调用函数来创建边。为了发现被调用函数，我们会运行任意的指向分析。当分析发现多个潜在的被调用函数时，会在函数应用与每个潜在的被调用函数之间创建边。当分析未发现任何被调用函数时，则不会创建边。指向分析不必具有很高的精确度。不精确的分析只会导致过多的类型注解，而这可能会降低运行时性能。

3.3 选择推断类型

完成图构建后，所提出的方法会确定是否将推断类型作为类型注解追加到原始源代码中。

首先，我们提出的方法会选择表示变量、参数或返回语句的顶点v，且其*given(v)*是包含*的类型。若某类型本身为*、是包含此类元素的数组类型，或是将此类类型作为参数类型或返回类型的函数类型，则该类型包含*。Array(*) 和Function([*],整型)即为示例。这些被选中的顶点v 是根据*infer(v)*追加新类型注解的候选对象。请注意，我们假定*infer(v)* 始终比*given(v)*更为具体。

然后，对于每个被选中的顶点v，该方法会枚举所有可到达v的最近源顶点。如果不存在最近源顶点，或者所有最近源顶点w 都满足*given(w)* 不是*，则将*infer(v)* 作为v的类型注解追加。此处，源顶点是指没有入边的顶点，或者是满足*given(w)*不包含*的顶点w。若存在从w 到v 的路径，且该路径除v和w 外不包含其他源顶点（$w \neq v$），则源顶点w 是可到达v 的最近源顶点。

考虑清单 1 中的示例程序。首先，选取表示变量u、参数 x 和y以及函数f 的返回语句的顶点作为候选项。表示变量z 的顶点不作为候选项，因为其给定类型为Bool，其中不包含*。对于表示y的顶点，其最近源顶点是表示true的顶点。由于true 的给定类型不是*，因此将y 的推断类型附加为类型注解。对于表示x 和u的顶点，它们最近的源顶点是表示succ(1)的顶点。对于表示函数f的返回语句的顶点，其最近源顶点是表示succ(x) 和succ(u)的顶点。由于succ 函数应用的给定类型为*，因此不将x、u 的推断类型以及函数f 的返回类型附加为额外的类型注解。回顾一下，函数succ 的类型为Function([*],*)。

3.4 正确性

TypePycker 的正确性由正确的类型推断引擎和底层渐进类型语言保证。其正确性定义如下：添加附加类型注解后的优化程序，必须产生与原始程序相同的值，或者在同一位置失败 [37]。这种正确性遵循渐进类型语言的渐进保证 [40]，也就是说，移除部分类型注解后，渐进类型程序仍然是良类型的，或能够正确执行。由于 TypePycker 只会添加*infer(v)* 或*given(v)*类型注解，因此，TypePycker 对附加类型注解的选择，等同于移除一个类型注解均为*infer(v)*的渐进类型程序中的部分类型注解。如果类型推断引擎只推导出正确的附加类型注解，那么 TypePycker 也能保证正确。

4 实验

我们使用渐进类型语言 Reticulated Python，对收集到的一组基准程序进行实验。实验的目的是验证所提出的方法能否改善渐进类型程序的执行时间，同时保持推导和选择附加类型注解所需的编译时间稳定且可接受。

在以下实验中，我们首先展示所提方法能够缩短执行时间。我们比较每个基准测试程序在通过类型推断附加所有附加类型注解后的执行时间，与通过所提方法仅附加筛选出的类型注解后的执行时间。随后，我们展示 TypePycker 的编译时间比已有工作 Herder [6]更加稳定。我们还展示，TypePycker 的执行时间与 Herder 相当。此外，我们证明所提方法在 Reticulated Python 的不同实现中均能带来性能提升。另外，我们在搭配 PyPy 的 Reticulated Python 上测试了若干基准测试程序，以展示 TypePycker 是否可能有益于采用即时编译的渐进类型语言。

4.1 实现与数据集

我们的实验配置包括一个用于为程序附加类型注解的编译器，以及一个扩展版 Reticulated Python，用于运行编译后的程序。为生成附加类型注解，我们为 Python 子集实现了一个基于约束的类型推断引擎 [36] 。我们使用现有的编译器工具包 *InferType* [26]。InferType 是一种嵌入式领域专用语言，基于 Java 实现，用于实现基于约束的类型推断。我们的引擎支持推断函数、局部变量、类方法和类字段的类型。给定一个 Python 程序，我们的类型推断引擎会为每个表达式分配一个唯一的类型变量，并通过遍历程序的抽象语法树，收集关于类型和类型变量的类型约束。随后，它通过求解收集到的类型约束，计算出一个将类型变量与其推断类型关联起来的绑定。InferType 会调用 SMT 求解器来计算该绑定。为简洁起见，我们省略对所适配类型系统的详细描述。关于类型规则的完整细节，请参见已有研究 [16, 20]。

我们在一个 Python 子集上实现了所提出的方法。我们通过遵循基于图的表示上的常见数据流分析，将 SimpliPy 中介绍的技术扩展到 Reticulated Python。例如，Reticulated Python 中其他二元运算的图构建与 SimpliPy 中介绍的方法类似。为了在图构建过程中识别函数应用的潜在被调用函数，我们采用了上下文不敏感、流不敏感的指向分析。该分析为每个函数应用近似计算一组被调用函数，而不考虑条件分支或执行顺序。例如，考虑以下代码片段：

```
1 e =f2  e(真)3 e  = 如果
co
```

其中，f、g 和h 是三个函数定义的标识符。分析将计算得出，在第 2 行和第 4 行，e 可能分别指向函数f、g 或h ，尽管在执行过程中，true 从未传递给h 或g，而9 也从未传递给f 。按照第 3 节中的描述，在图构建过程中，它会从参数true 和9 向f、g 和h 的参数创建边。尽管所采用的指向分析简单且不精确，但在我们的实验中，它对于提升基准程序的性能是实用的。

清单 4 带有生成的快速函数的示例程序

```
1  定义f_快速(x: *, y: Bool):
2      u = x
3      z: Bool = y
4      如果 z 则返回 succ(x)，否则返回 succ(u)
5
6  定义f(x, y):
7      u = x
8      z: Bool = y
9      返回 如果 z 则 后继(x) 否则 后继(u)
10
11 f_fast(后继(1), 真)
```

我们的实现对 Reticulated Python 进行了两项主要修改。首先，我们改编了近期一项研究中的技术 [7]。改编该技术的目的是在保留程序行为的同时，最大限度地发掘优化机会。这里所说的保留程序行为，是指优化程序必须产生与原程序相同的值，或者在相同位置失败 [37]。其主要思路是为每个函数生成一个附加了额外类型注解的优化版本，同时保留原函数。随后，根据参数和实参的静态类型来分派函数应用。例如，清单 1 中带有附加类型注解的示例程序，将按清单 4 所示生成。生成了一个名为f_fast 的优化函数。源代码中也保留了原函数。在程序主体部分，由于实参的静态类型是附加参数类型的子类型，因此会调用优化函数。下文将此技术称为快慢模式。

其次，我们扩展了 Reticulated Python，使其支持局部变量的类型注解。这样，我们就能像在 SimpliPy 中所述，为局部变量选择附加类型注解。原版 Reticulated Python 不支持局部变量的显式类型注解。

我们为实验构建了一个数据集。我们在制品中将其公开。数据集中的程序没有由程序员为函数参数、函数返回值或局部变量添加显式类型注解。请注意，尽管我们在实验中将这些未注解程序用作基线，这些程序并非完全无类型。这些程序是部分类型化的，因为底层语言 Reticulated Python 总会为若干表达式（例如数字和数组字面量）赋予静态类型。这种部分类型化引入了涉及运行时类型转换的非平凡场景，其中存在追加或丢弃附加类型注解以提升执行性能的机会。因此，尽管这些程序

并未由程序员以理想方式添加显式类型注解，我们的数据集仍有资格作为试验台，用于检验该方案能否提升执行性能。

该数据集包含从多个公开来源收集的 50 个 Python 程序。其中 31 个程序收集自多项学术研究 [6, 26, 28]。这些程序包括 Python 性能基准测试套件的一个子集，[2] 以及现有研究用于评估 Reticulated Python 的程序 [5, 7, 21, 43]。下文将这些程序称为 *MicroBench*。另有 6 个程序收集自东京大学用于本科生教学的 Python 教科书。[3] 尽管这些程序并非为渐进式类型设计或使用，但我们假设它们体现了教育场景中真实的 Python 编码模式，并可作为具有代表性的真实程序。

另有 9 个程序源自《计算机程序的构造和解释》（SICP）一书 [1]。这些程序在类型推断的编译时间方面预计颇具挑战性，在静态构建和分析数据流图方面也是如此，因为它们包含数量较多的函数定义和函数应用（见表 1），其中还包括函数指针的使用。由于原始程序是用 Scheme 编写的，[4]我们使用大语言模型（Gemini2.5 Flash）协助将其转换为 Python。我们在提示词中附上原始 Scheme 程序，并使用以下提示词进行转换：

```
1 将此程序转换为 Python。此程序改编自《2 计算机程序的构造和解释》。
请
的
```

由于 Reticulated Python 和 Herder 的限制，翻译过程中会排除类等若干 Python 特性，因为生成的程序中不会用到这些特性。

另外 4 个程序是对 MicroBench 中 4 个程序的变体修改，包含嵌套函数调用。这些程序旨在测试我们的方法在包含嵌套函数调用的程序中的编译时间和性能影响。在此，嵌套函数调用是指一个函数内部的函数调用。对于给定的程序，我们会手动重构复杂或冗长的语句，使其调用辅助函数。我们为每个程序创建 3 到 6 个辅助函数。例如，某个基准测试程序中的一段代码如下所示：

```
1 def SOR_execute(omega, G, cycles):
2     w,h,d = G
3     for p in range(cycles):
4         for y in range(1, h - 1):
5             ...
```

我们将其重构为

[2]https://github.com/python/pyperformance （访问日期：2025-09-29）。[3] https://www.graco.c.u-tokyo.ac.jp/labs/morihata/textbook/python_textbook.htm （访问日期：2025-09-29）。[4]https://github.com/ivanjovanovic/sicp/tree/master （访问日期：2025-09-29）。

```
定义SOR_execute(omega,G,cycles):
w
h
h
(1
```

循环_高度是一个手动创建的辅助函数，并在函数SOR_执行中调用，因此引入了嵌套函数调用。下文将这些手动合成的程序称为*Syn*。所有代码更改均未改变原程序的功能。我们在制品中列出了完整的代码更改清单。

表 1 概述了基准程序的一些详细特征。表中每一行都按数据源对数据集进行分类。尽管我们的数据集规模不大，且可能存在一些采集偏差，但与现有研究中使用的其他数据集相比，它包含了更多来自多个公开来源的程序 [6,7, 21, 44]。

表 1 数据集概览。程序按数据源分组（括号内为包含的程序数量）。每个单元格给出程序中的最小值/平均值/最大值。Def 和 Call 分别表示词法函数（及方法）定义和函数应用的数量。Edge 表示 TypePycker 构造图中的边数。

组（程序）	LOC	Def	Call	Edge
MicroBench（31）	22/64/145	2/7/15	2025年2月11日	51/319/823
PyTextbook（6）	55/99/215	2025年7月12日	13/25/48	259/489/1011
SICP（9）	181/317/504	19/42/75	18/56/105	509/1052/1760
Syn（4）	55/85/139	9/11/12	22/30/38	390/500/804

实验在一台搭载 AMD Ryzen 9 9950X 16 核处理器的 Ubuntu 24.04.2 LTS 台式机上进行。该台式机安装了 OpenJDK 17.0.5 和 Python 3.9.7，用于我们的实现。程序使用 Reticulated Python 的防护语义执行 [43, 44]。

4.2 验证选择的有效性

我们比较基准程序的执行时间，以检验所提出的附加类型注解选择能否提升渐进类型程序的性能。给定一个基准测试程序，我们比较其三种变体的执行时间：*Given* 变体等同于原始基准测试程序；*Infer* 变体包含由类型推断得到的全部附加类型注解；*Chosen* 变体仅包含由 TypePycker 选出的附加类型注解。我们将每种程序变体各执行 10 次，并测量所耗时间的算术平均值。下文我们使用 Given、Infer 和 Chosen 这三个术语来指代这三类程序。

我们的方法在 50 个基准程序中的 9 个上选择了由类型推断得到的全部附加类型注解。这是我们方法的预期行为。直观而言，如果一个渐进类型程序能够借助类型推断引擎增强为完全类型化，或者如果所有附加类型注解都预期能够消除运行时类型转换，那么就应当使用这些附加类型注解以获得最优性能，而不是将其丢弃。例如，来自 MicroBench 的一个基准程序曼德勃罗计算经典分形，主要涉及浮点数上的二元运算。它没有使用任何在类型方面未知的库函数。我们的类型推断引擎推断出了该程序中的大部分类型。下文我们在本小节的进一步比较中排除这 9 个程序的结果。这些程序的 Infer 与 Chosen 在实验中也给出了相近的性能。

实验结果表明，所提出的方法对基准程序的性能有正向影响。图3展示了每个基准测试程序中 Given、Infer 和 Chosen 的执行时间。为了定性评估性能影响，我们将每个基准测试程序中 Infer 和 Chosen 的相对执行时间分为三种情况。分类依据是测得执行时间的算术平均值（平均值）和标准误（SE）：

$$\text{case} = \begin{cases} \text{win} & Mean_{Chosen} + SE_{Chosen} < Mean_{Infer} - SE_{Infer} \\ \text{loss} & Mean_{Chosen} - SE_{Chosen} > Mean_{Infer} + SE_{Infer} \\ \text{tie} & \text{otherwise} \end{cases}$$

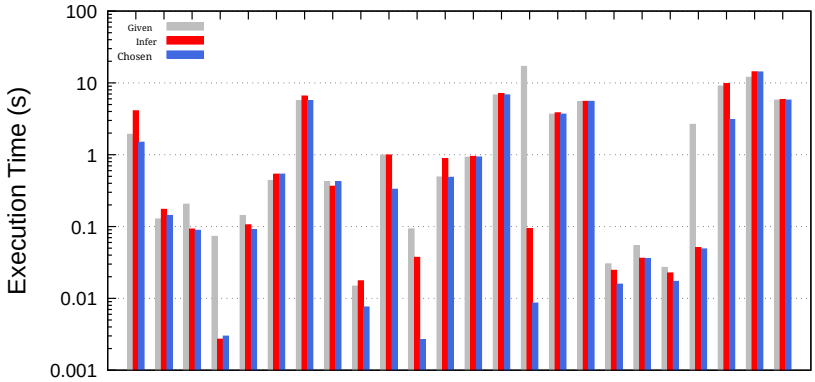
如果 Infer 和 Chosen 的误差条不重叠，则将程序视为胜出或落败 情况；否则，视为平局 情况。这表明我们的方法对性能的影响并不显著。

我们的方法在 41 个程序中的 32 个取得胜出，在 41 个程序中的 6 个出现平局。图3e展示了 Infer 与 Chosen 的加速比，计算方式为平均值$_{Infer}$/平均值$_{Chosen}$。在 22 个程序中，Chosen 的加速比超过 1.1 倍；在 4 个程序中，加速比超过 5 倍。在 10 个程序中，Infer 比 Given 慢，而 Chosen 比 Given 和 Infer 都快。这展示了这样一些情况：简单地将类型推断建议的所有类型注解都添加进去，会降低渐进类型程序的性能；但选择性地添加这些注解则能提升性能。

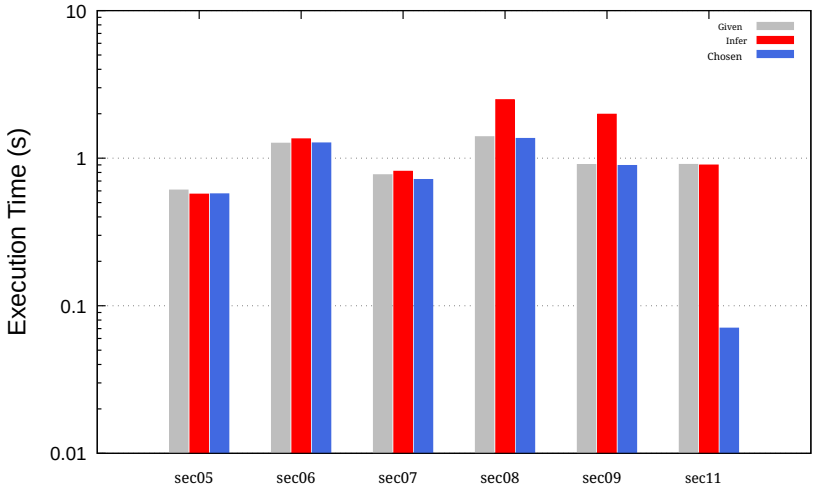
我们还观察到，我们的方法并不总能提升性能。在 3 个程序中，该方法导致了性能损失。我们手动检查了运行较慢的程序，发现性能下降主要是由静态分析的局限性导致的。在图3a 中的dijkstra_alternate里，下面展示了相关代码片段：

```
1 u  =expr2 遍历 v 于顶点：3 距离[v] = 距离[u
] .
```

变量 u 被赋值为expr ，其类型为未知类型 *；之后u 被用作数组访问的索引，预期类型为整型。推断出的u 类型为整型，该类型会添加到 Infer 中，但不会添加到 Chosen 中。这会使运行时类型转换整型⇐ * 从赋值处转移到后续的数组访问中。然而，后续的运行时类型转换发生在 for 循环中，因此导致

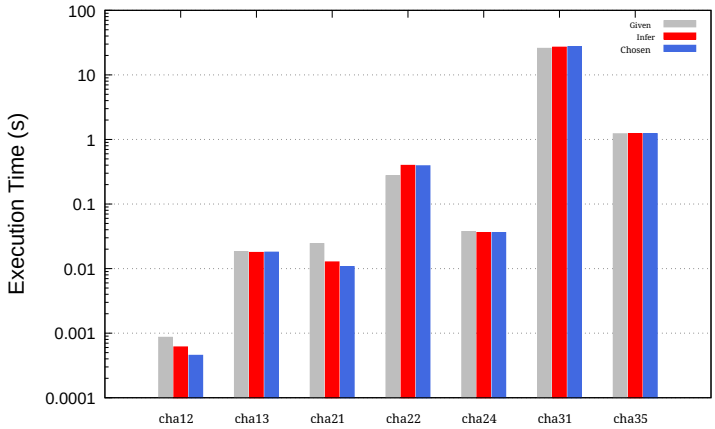


(a) MicroBench

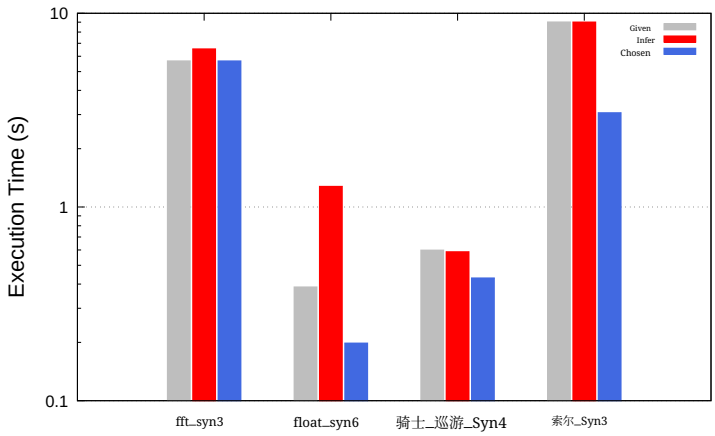


(b) PyTextbook

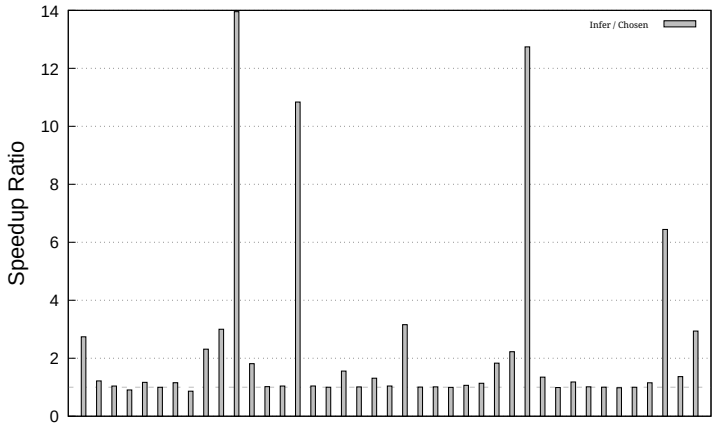
图3 执行时间结果（第一部分）。



(c) SICP



(d) Syn



(e)加速比（Infer 相对于 Chosen）

图3 执行时间结果（第二部分）。(a)-(d)：每个子图组按数据源对程序进行分组；X 轴显示程序名称；Y 轴显示以秒为单位的执行时间；每个程序从左到右分别显示 Given、Infer 和 Chosen；每个柱形表示 10 次运行的平均值；越低越好。(e)：Infer 相对于 Chosen 的加速比；X 轴显示 (a)-(d) 中的程序； > 1 越好。

Chosen 在执行期间产生了更多开销。程序 float 中的另一个较慢案例也存在类似情况。对于另一个较慢案例 cha31，Chosen 会追加不同的类型注解，并产生多个与 Infer 不同的运行时类型转换。很难将性能下降归结为一个直接的原因。

我们还进行了检查，发现当运行时类型转换会发生在不同的控制流分支中时，我们的方法可能无法带来更好的性能。例如，程序seco8 中的一段代码如下所示：

```
1 定义mergesort(xs): 2 如果 长度(xs) <= 1:
3
并
围
```

第 7 行的参数是一个类型为 Array(*) 的数组字面量，因为 randint 是一个类型不可用的库函数。在 Infer 和 Chosen 中，函数 merge 的返回类型都是Array(Int) 。mergesort的返回类型被推断为 Array(Int)。在 Infer 中，该返回类型注解会在 if 分支中产生一次运行时类型转换 Array(*)⇐Array(Int) ，但在 else 分支中不会产生运行时类型转换。在 Chosen 中，else 分支会发生一次运行时类型转换 *⇐ Array(*) 。由于其最近源顶点之一（表示第 7 行的数组字面量）包含*，TypePycker 不会为 mergesort 选择返回类型注解。然而，程序更频繁地执行 else 分支，因此在 Infer 中，else 分支的运行时开销更低。为该函数选择返回类型会降低执行性能。

4.3 展示稳定的编译时间

我们将我们的方法与现有工作 Herder 进行比较，以表明我们的方法在编译时间方面更加稳定且实用。此处，编译时间主要包括为推导附加类型注解而进行的类型推断，以及将选定的类型注解插入原始程序。Herder [6] 是一个工具，用于静态分析具有不同附加类型注解子集的渐进类型程序的运行时类型转换开销。这些附加类型注解由其类型推断推导得出。它输出一个由其分析所建议的、具有最优性能的程序，并将附加类型注解的一个子集追加到该程序中。与我们的方法的主要区别在于选择附加类型注解的方式。Herder 借助精心设计的开销分析，对附加类型注解的所有可能子集进行全面分析，而 TypePycker 采用轻量级摊销方法。另一个区别是，Herder 将类型推断与其选择集成在一起，而我们的选择独立于外部类型推断引擎。

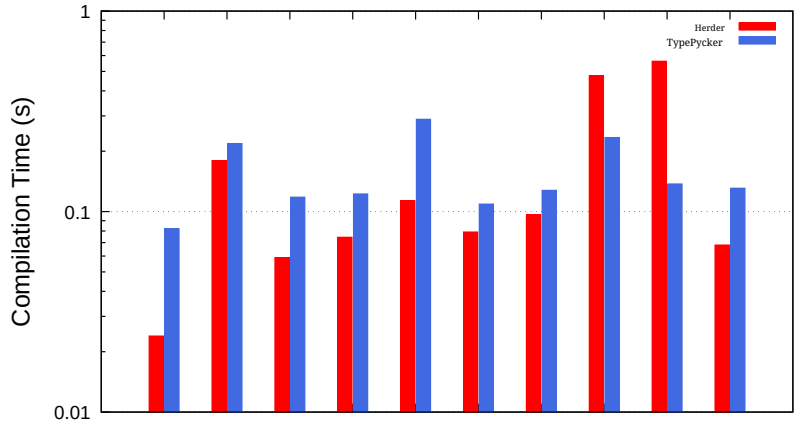
我们比较了 TypePycker 和 Herder 在各基准程序上的编译时间。我们还比较了使用 TypePycker 选出的附加类型注解与使用 Herder 选出的附加类型注解时，各基准测试程序的执行时间。

实验结果表明，我们的方法相比 Herder 能实现更稳定的编译时间。图4展示了 Herder 和 TypePycker 在各基准程序上的编译时间。请注意，由于 Herder 不支持类等若干 Python 特性，因此部分基准程序未纳入比较。尽管 Herder 在小型程序上的编译时间尚可且更快，但扩展到其他程序时表现不佳。例如，对于图4a中来自 MicroBench 的程序，Herder 和 TypePycker 的编译时间都在 1 秒以内。对于其中大多数程序，Herder 的速度都快于 TypePycker。然而，在 28 个程序中，Herder 的编译时间有 12 个超过 1 秒，7 个超过 10 秒。最糟糕的情况下，耗时超过 2000 秒；该程序来自 SICP，并非人为构造。编译缓慢的原因是 Herder 必须探索庞大的搜索空间。尽管与暴力枚举相比，Herder 采用变分类型 [9] 以提高计算效率，但在一般情况下，程序中运行时类型转换的不同出现方式的可能数量呈指数级增长。例如，如果程序包含深度嵌套的函数调用，那么在针对函数参数的不同类型注解组合下，程序中会出现大量不同的运行时类型转换。Herder 必须分析所有不同组合的开销，才能提出性能最优的建议。

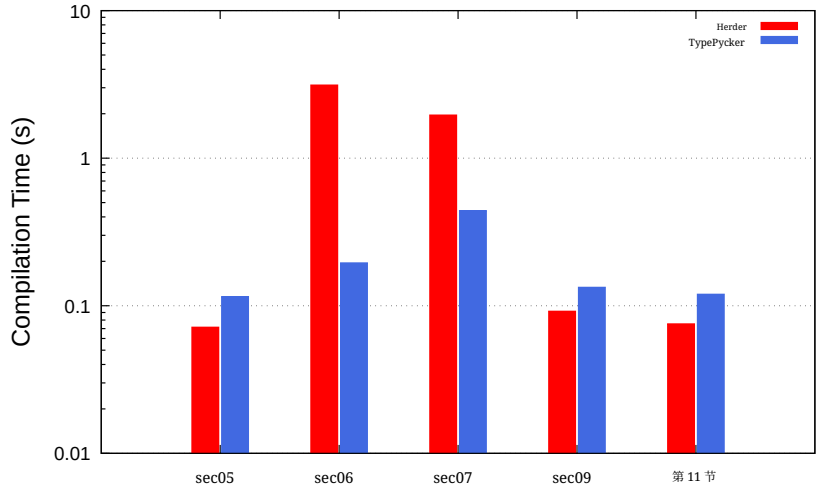
相比之下，我们的方法在所有基准程序中都表现出稳定的编译时间。在大多数情况下，编译时间都少于 1 秒，包括 Herder 耗时较长的情况。我们还分别测量了 TypePycker 编译时间中的类型推断耗时，以及插入所选类型注解的耗时。在图4所示的所有基准程序中，类型推断平均耗时 0.23 秒；后续过程平均耗时 0.026 秒。类型推断成本较高，是因为它会调用 SMT 求解器，使用工具包 InferType 进行约束求解，如第 4.1 节所述。调用 SMT 求解器的相对较高成本，是 TypePycker 在 MicroBench 上比 Herder 更慢的原因之一。在 Herder 中，由于其类型推断和选择过程是集成在一起的，我们无法将类型推断耗时与选择耗时分别测量。

实验结果还表明，对于包含嵌套函数调用的程序，所提出的方法有望实现更快且稳定的编译时间。对于图4d中 Syn 的所有程序，TypePycker 的编译时间都显著短于 Herder。TypePycker 编译这四个程序的时间不到 1 秒，而 Herder 则需要 3 到 627 秒。与此同时，对于图3d中的这些程序，Chosen 的速度快于 Infer，加速比最高可达 6 倍。尽管这些程序是基于现有基准测试修改而来的变体，但其结果证明，确实存在一些程序能够从我们的方法中明显获益。TypePycker 展现了一个胜例（Chosen 优于 Infer），并且在 28 个程序中的 7 个上，其编译时间短于 Herder

我们证明，所提出的方法能够实现与 Herder 相当的执行性能。图 5 展示了基准程序在附加由 Herder 和 TypePycker 选出的附加类型注解后的执行时间。Herder 只会为函数参数和函数返回值提供类型注解。为进行公平比较，此处我们将附加的类型注解限制为仅

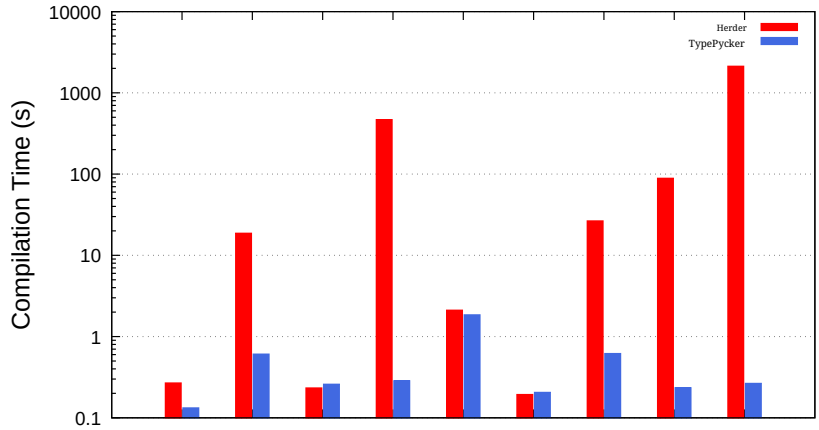


(a) MicroBench

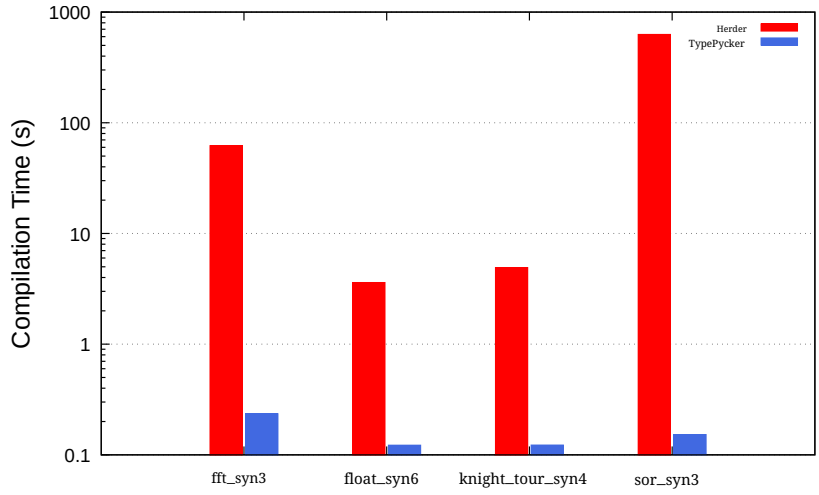


(b) PyTextbook

图4 Herder 和 TypePycker 的编译时间结果（第一部分）。



(c) SICP

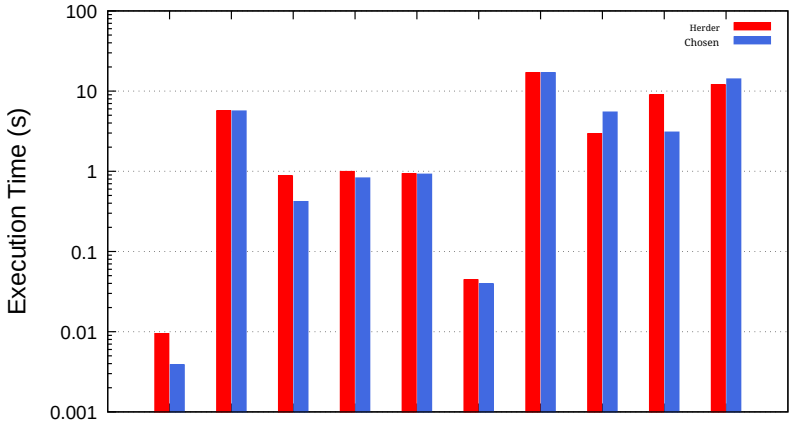


(d) Syn

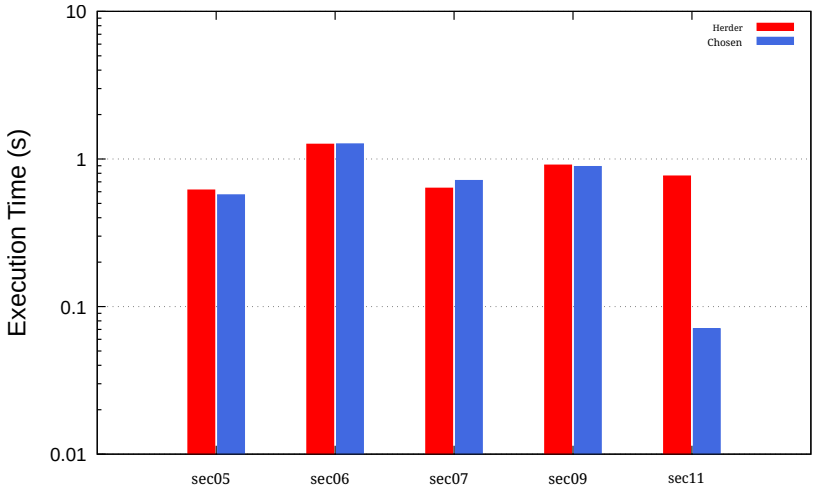
图 4 Herder 和 TypePycker 的编译时间结果（第二部分）。每个子图组按数据源对程序进行分组。X 轴显示程序名称。Y 轴显示以秒为单位的编译时间。

函数参数和函数返回值，不包括局部变量的类型。使用我们修改后的 Reticulated Python 执行这些程序 10 次。此处比较中唯一的差异是为函数参数和函数返回值添加了类型注解。

在 28 个程序中，Chosen 有 17 个比 Herder 更快，其余 11 个则比 Herder 慢。执行时间比（Herder 相对于 Chosen）介于 0.5 倍至 10 倍之间。在若干程序中，Chosen 的性能优于 Herder，这主要是因为我们的类型推断引擎能够推断出更多函数参数类型，从而带来更多优化机会。

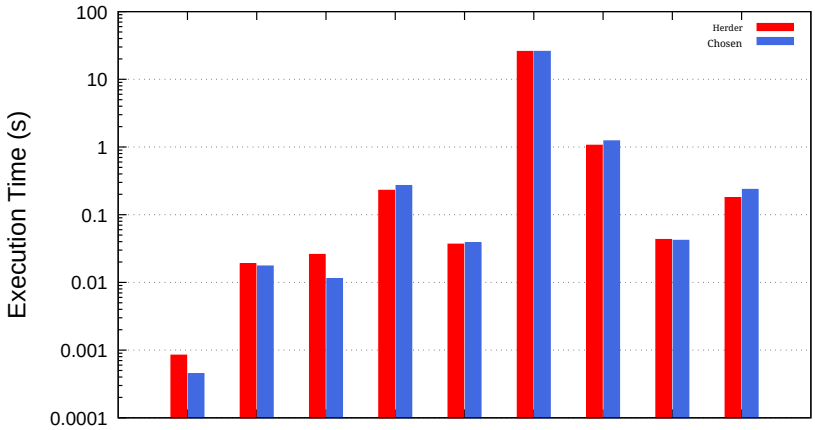


(a) MicroBench

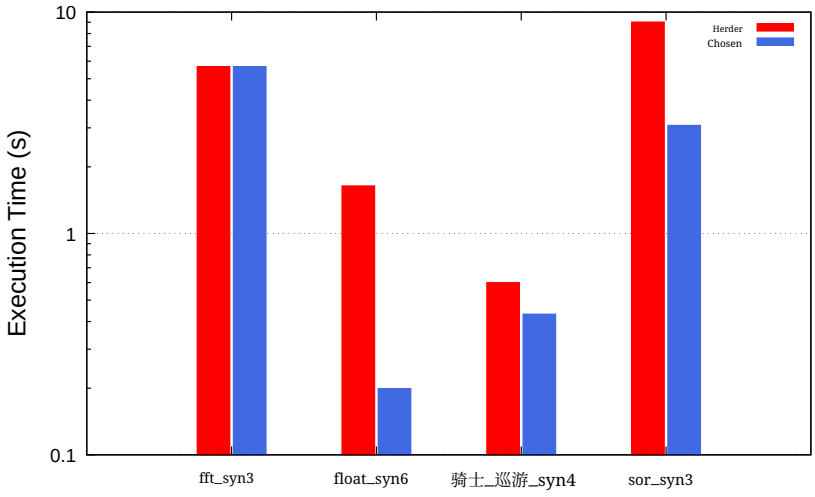


(b) PyTextbook

图 5 Herder 和 TypePycker 的执行时间结果（第一部分）。



(c) SICP



(d) Syn

图 5 Herder 和 TypePycker（第二部分）的执行时间结果。每个子图组按数据源对程序进行分组。X 轴显示程序名称。Y 轴显示以秒为单位的执行时间。每个柱形代表 10 次运行的平均值。

4.4 评估不同语言平台上的性能影响

我们在基于 Reticulated Python 的四种不同语言平台上运行基准程序，以展示所提出的方法如何影响执行性能。这里，语言平台指的是底层语言 Reticulated Python 的一种具体实现变体。由于表达式可能被赋予不同的静态类型，对于同一个程序，每种实现的语言变体都会产生不同的运行时类型转换。我们通过在原始 Reticulated Python 的基础上应用两项主要改动的不同组合，实现了这四种语言平台：快慢技术，以及对局部变量的类型注解支持。请注意，若省略快慢技术，将无法保持程序行为，因为附加类型注解会直接添加到原始函数中。本小节旨在表明，性能提升并非仅仅源于我们对原始 Reticulated Python 所做的改动；我们的方案对附加类型注解的选择也促成了观察到的性能提升。

我们将通过本方法仅追加选定类型注解的执行时间，与追加由类型推断得出的所有附加类型注解的执行时间进行比较。在每个语言平台上，每个程序均执行 10 次，并测量经过的时间。与第 4.2 节中的比较相同，我们根据 Mean 和标准误，将 Infer 和 Chosen 的相对执行时间分为胜、平、负三种情况。由于 Chosen 追加的附加类型注解与 Infer 相同，下面的比较排除了 9 个基准程序的结果。

结果表明，所提出的方法能够在四种不同的语言平台上带来积极的性能影响。我们将支持快慢技术的语言平台记为*F*，支持局部变量类型注解的语言平台记为*L*，而不支持这两者的语言平台记为*n*。例如，支持局部变量类型注解但不支持快慢技术的语言平台记为*nL*。在四种语言平台*nn*、*nL*、*Fn*和*FL*上，在 41 个程序中，Chosen 相较于 Infer 分别在 29、27、40 和 38 个程序中表现更优（胜）或取得相当的性能（平）。四种语言平台的结果表明，性能提升并不仅仅来自我们对 Reticulated Python 所做的修改。我们的方法对类型注解的选择有助于实现性能提升。

表2展示了若干选定基准程序在不同语言平台上的分类结果和性能比率。这些程序的选取旨在展示三种情形：给定一个程序，我们的方法在所有语言平台上呈现出以下情况之一：a) 全部为胜、平或负；b) 胜/负与平局并存；c) 在一个平台上获胜，但在另一个平台上落败。

表2 四种语言平台上所选程序的结果。水平线按分类情景对程序进行分组。第二列显示四个平台上的分类结果。W、T、L 分别表示胜、平、负。后续列显示各平台上的加速比（Infer 相对于 Chosen）。 > 1 表示更优。

程序	分类方式	*nn*	*nL*	*Fn*	*FL*
不相交_集合	WWWW	1.19	1.18	1.20	1.17
list	WWWW	2.45	2.44	1.80	1.81
sec07	WWWW	1.20	1.14	1.06	1.14
第09节	WWWW	1.34	1.35	2.25	2.22
fft	WWTW	1.19	1.15	1.00	1.16
筛法	TWTW	0.99	1.03	0.99	1.04
cha13	TWWT	1.02	1.11	1.13	0.99
cha24	LLTT	0.98	0.92	0.99	1.00
dijkstra_替代方案	LLWL	0.87	0.91	1.08	0.91
骑士_巡游	LLWW	0.24	0.56	1.20	3.00
cha31	WLTL	1.02	0.94	1.00	0.98
sor_syn3	LLWW	0.06	0.06	2.94	2.94

4.5 使用即时编译测试行为

我们在搭载 PyPy[5] 的 Reticulated Python 上测试了若干基准程序。PyPy 是一种支持即时编译的 Python 变体。此项测试旨在展示 TypePycker 能否提升支持即时编译的渐进类型语言的执行性能。图6展示了测试程序在添加了类型推断得出的所有附加类型注解，以及仅添加 TypePycker 选出的类型注解后的执行时间。我们发现，TypePycker 能够提升某些基准程序的性能。我们还观察到，与 CPython 相比，TypePycker 在 PyPy 上造成的性能影响较差或有所不同。例如，在 CPython 上，TypePycker 的表现优于使用类型推断得出的所有附加类型注解fft 和 so46582238（图3）。然而，在 PyPy 上，TypePycker 会降低这两个程序的执行速度。由于比较范围有限，且测试的基准程序数量也相对较少，因此还需要更全面的评估，才能可靠地判断 TypePycker 是否能使基于即时编译或其他动态分析技术的渐进类型语言受益。我们将此视为未来工作方向之一。

[5]其他大多数基准测试在使用 PyPy 的 Reticulated Python 上会遇到错误，现有工作也报告了这一点 [7]。

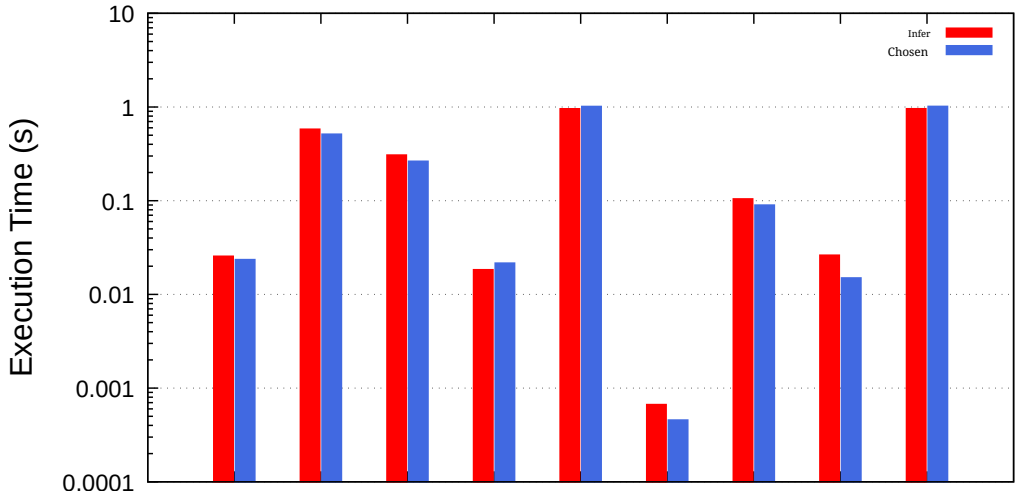


图6 在 Reticulated Python 上使用 PyPy 的执行时间结果。横轴表示程序名称。纵轴表示以秒为单位的执行时间。每个柱形表示 10 次运行的平均值。

5 有效性威胁

本研究可能受到若干潜在因素的影响，这些因素可能会影响研究结果的有效性。

本工作未正式讨论其优化在保持程序行为方面的正确性。其正确性依赖于底层的类型推断引擎。为确保正确性，必须使用只推导可靠类型且不会改变程序行为的类型推断引擎。探索此类引擎不在本文的研究范围之内。我们通过两种有限的方式来缓解这一风险。首先，我们手动验证了实验中基准程序的推断类型是否正确，并确认它们不会改变基准程序的行为。其次，我们使用近期一篇研究论文 [7] 中介绍的几个具有挑战性的程序测试了我们的实现。这些程序专门用于测试正确性。我们的引擎成功保留了这些程序的行为。我们将这些程序收录在制品中。

实验中观察到的性能提升可能比实际场景中所能达到的更为显著。这一局限主要源于我们将收集到的未注解程序用作基线。尽管在实验中由 Reticulated Python 执行时，这些基准程序被视为部分类型化，但在渐进类型语言中，真实程序可能已经由程序员为函数参数、函数返回值或局部变量添加了一定程度的具体类型注解。我们未能收集到包含真实注解程序的合适数据集。

6 相关工作

随着各类渐进式类型语言的发展，理解渐进式类型中的性能变化已成为一个重要的研究领域。现有的渐进式类型语言主要分为两大类：采用擦除语义和非擦除语义的语言 [10, 13]。采用擦除语义的语言会在运行时擦除类型，从而允许意外类型的值从无类型代码流入有类型代码。TypeScript、Flow 和 mypy 就是这类语言的例子。相比之下，采用非擦除语义的语言会通过在无类型代码和有类型代码的边界处执行运行时类型转换，在运行时强制执行静态类型。此类语言又分为深度语义（如 Gradualtalk [2], 和 Typed Racket [42] ）与一阶语义（如 Reticulated Python [43], 和 Nom [34]）。多项研究共同发现，采用非擦除语义的渐进式类型语言会产生显著的运行时开销，例如 Typed Racket [14, 41] 和 Reticulated Python [6, 15]。本文不考虑 TypeScript 等语言，因为采用擦除语义编写的程序不会因静态类型注解而产生显著的运行时开销。本研究旨在开发一种编译时技术，以减轻采用非擦除语义的渐进式类型语言中的性能损失。实验中研究的语言 Reticulated Python 就是此类语言的一个实例。探索该方案在其他采用非擦除语义的渐进式类型语言 [22, 31, 42]中的有效性，是我们未来的工作。

渐进式类型中出现性能下降的报告之后，涌现出许多相关研究。其中一个主要方向是增强静态信息以提升性能。Rastogi 等人 [37] 首先使用类型推断来优化 ActionScript [33] 中的渐进类型程序，并以保持程序行为作为正确性目标。Vitousek 等人 [44] 通过移除经类型推断验证的运行时类型转换，优化了 Reticulated Python 的瞬态语义。遵循保持程序行为这一原则，Campora 等人 [7] 提出了判别式类型化：基于类型推断，同时保留函数的优化版本和未优化的原始版本，从而安全地提升性能。我们的实现采用了 Campora 等人同时维护优化函数与未优化函数的思路。然而，与他们的方法不同，由于我们使用了不同的类型系统，我们的方法无法正式保证运行时错误会在相同位置被归咎。我们的方法通过有选择地添加由类型推断得出的类型注解来提升性能。

已有研究通过静态推理类型如何影响性能，来应对渐进式类型中的性能陷阱。Herder [6] 是一种静态代价分析工具，用于估计给定程序中不同类型化参数组合的执行时间。在渐进式类型的语境中，这些组合通常被称为配置 。该工具会根据其开销分析输出预期可获得最优执行时间的配置。与 Herder 相比，我们的创新在于一种选择附加类型注解的高效方法。TypePycker 无需分析所有可能的配置。实验结果表明，TypePycker 能够取得相当的执行性能，同时具有更稳定的编译时间。Khan 等人 [21] 开发了一种基于机器学习的方法*LearnPerf* 来

预测给定程序中可能配置的性能。与 TypePycker 类似，LearnPerf 使用外部类型推断引擎来推导附加类型注解。除第 2 节所述差异外，LearnPerf 与 TypePycker 还有两点不同。第一，LearnPerf 不考虑保持程序行为。第二，LearnPerf 为每个输入程序训练单独的模型，这限制了其在实际使用中的可移植性。

理解并优化渐进式类型性能的另一个方向是采用运行时技术。Bauman 等人 [3]在 Takikawa 等人报告了显著的运行时开销之后，展示了通过使用追踪式即时编译所实现的实际性能提升。Richards 等人 [38] 弥合了利用即时编译的虚拟机与渐进式类型语言之间的差距，以消除冗余的运行时类型转换。Roberts 等人 [39] 采用标准即时编译来降低瞬时运行时类型转换的开销。本文主要在 Reticulated Python 上检验所提出的方法，且不使用即时编译等额外运行时技术，因为我们旨在研究该方案作为编译时技术的真实效果。尽管第 4.5 节中使用 PyPy 进行的实验相当有限，但我们相信所提出的方法可以与其他离线编译和运行时技术相结合，并可能带来性能提升。Greenman 等人 [14] 和 Hejduk 等人 [17] 通过基于理性程序员 [23, 24]方法的大规模实证研究，探讨了如何利用性能分析数据来引导类型迁移决策。我们的工作则针对不同的切入点，即提供一种编译时技术，以自动提升渐进类型程序的性能。

静态类型推断已成为适用于各种编程范式的成熟技术。类型推断有助于实现早期错误检测 [19, 29] 和性能提升 [8]等基础应用。在 Python 中，已有许多类型推断引擎采用了各种技术，例如基于规则的系统 [12], 抽象解释 [32], SMT 求解器 [16, 45], 以及基于机器学习的方法 [18, 30, 35]。在这项工作中，我们使用编译器工具包 InferType [26]实现了一个自定义类型推断引擎。尽管我们尚未通过实证探索与现有引擎的集成，但我们预计所提出的方法能够顺利扩展，因为它独立于类型推断过程。

类型推断领域已应用多种数据流分析。Maggi 等人 [27] 提出了一种利用数据流分析推断与 Java 字节码相关类型的方法。Zhou 等人 [46] 提出了一种基于数据流分析的反编译类型推断方法。Bhanuka 等人 [4] 利用子类型约束捕获数据流，以便在基于约束的类型推断系统中生成易于理解的类型错误消息。我们的方案与这些用于构建图以模拟值传播的数据流技术有相似之处。我们的贡献则有所不同：通过有选择地添加类型注解，提升渐进类型程序的性能。

7 结论

我们提出了 TypePycker，这是一种新的轻量级方法，可高效选择类型推断得出的类型注解，从而提升渐进类型程序的执行性能。由于值在无类型代码与有类型代码之间反复跨越边界时，通常会产生高昂的运行时类型转换成本，因此该方案会选择沿数据流放置的类型注解。它采用轻量级的摊销方法，快速检查数据流中的所有无类型变量是否都需要添加类型推断得出的类型注解。

我们展示了 TypePycker 的用法，并对其提升 Reticulated Python 执行性能的效果进行了实验。我们发现，在基准程序中，与简单地使用类型推断得出的所有推断类型相比，TypePycker 能实现更快的执行速度，最高加速比超过 5 倍。与现有方法相比，TypePycker 的执行性能相当，同时编译时间更加稳定。我们还观察到，在现有方法导致编译时间显著过长（例如超过 10 分钟）的真实程序中，TypePycker 的编译时间快得多。我们认为，该方案有望缓解实际场景中渐进式类型的性能瓶颈。

数据可用性声明

本文相关的制品可在 Zenodo 上获取 [25]。

致谢 本研究部分由日本学术振兴会科研费 JP24H00688 资助 .

参考文献

[1] Harold Abelson 和 GeraldJ. Sussman。《计算机程序的构造和解释》。第 2 版。美国马萨诸塞州剑桥：麻省理工学院出版社，1996 年。isbn：0262011530。[2] Esteban Allende、Oscar Callaú、Johan Fabry、Éric Tanter 和 Marcus Denker。“Smalltalk 的渐进式类型”。载于：*Science of Computer Programming* 96（2014 年）。Smalltalk 系统进展特刊，第 52–69 页。issn：0167-6423。doi：10.1016/j.scico.2013.06.006。[3] Spenser Bauman、Carl Friedrich Bolz-Tereick、Jeremy Siek 和 Sam Tobin- Hochstadt。“健全的渐进式类型：虽未完全消亡”。载于：*ACM* 编程语言会议录 1.OOPSLA（2017 年 10 月）。doi：10.1145/3133878。[4] Ishan Bhanuka、Lionel Parreaux、David Binder 和 Jonathan Immanuel Brachthäuser。“理解 Flow：改进基于约束的类型推断中的类型错误消息”。载于：*ACM* 编程语言会议录 7.OOPSLA2（2023 年 10 月）。doi：10.1145/3622812。

[5] John Peter Campora 和 Sheng Chen。“驯服渐进式类型中的类型注解”。载于：*ACM* 编程语言会议录 4.OOPSLA（2020 年 11 月）。doi：10.1145/3428259。[6] John Peter Campora、Sheng Chen 和 Eric Walkingshaw。“类型转换与代价：协调渐进式类型中的安全性与性能”。载于：*ACM* 编程语言会议录 2.ICFP（2018 年 7 月）。doi：10.1145/3236793。[7] John Peter Campora、Mohammad Wahiduzzaman Khan 和 Sheng Chen。“基于类型的渐进式类型性能优化”。载于：*ACM* 编程语言会议录 8.POPL（2024 年 1 月）。doi：10.1145/3632931。[8] Robert Cartwright 和 Mike Fagan。“软类型”。载于：*SIGPLAN* 通讯 26.6（1991 年 5 月），第 278–292 页。issn：0362-1340。doi：10.1145/113446.113469。[9] Sheng Chen、Martin Erwig 和 Eric Walkingshaw。“将类型推断扩展到变体程序”。载于：*ACMTrans. Program. Lang. Syst.* 36.1（2014 年 3 月）。issn：0164-0925。doi：10.1145/2518190。[10] Benjamin Chung、Paley Li、Francesco Zappa Nardelli 和 Jan Vitek。“KafKa：面向对象的渐进式类型”。载于：第 *32* 届欧洲面向对象编程会议（*ECOOP 2018*）。Todd Millstein 编。第 109 卷。莱布尼茨国际信息学会议录（LIPIcs）。德国达格施图尔：达格施图尔城堡—莱布尼茨信息学中心，2018 年，12:1–12:25。isbn：978-3- 95977-079-8。doi：10.4230/LIPIcs.ECOOP.2018.12。[11] Damien P. George。*MicroPython*。https://micropython.org。2014 年。（访问于 2025-09-29）。[12] google。*pytype*。2021 年。url：https://google.github.io/pytype/ （访问于 2025- 09-29）。[13] Ben Greenman 和 Matthias Felleisen。“ 类型健全性与性能的谱系 ”。载于：*ACM* 编程语言会议录 2.ICFP（2018 年 7 月）。doi：10.1145/3236766。[14] Ben Greenman、Matthias Felleisen 和 Christos Dimoulas。“性能分析器如何助力类型迁移”。载于：*ACM* 编程语言会议录7.OOPSLA2（2023 年 10 月）。doi：10.1145/3622817。[15] Ben Greenman 和 Zeina Migeed。“论类型标签健全性的代价”。载于：*ACM SIGPLAN* 部分求值与程序操作研讨会。PEPM ’18。美国加利福尼亚州洛杉矶：计算机协会，2017 年，第 30–39 页。isbn：978-1-4503-5587-2。doi：10.1145/3162066。[16] Mostafa Hassan、Caterina Urban、Marco Eilers 和 Peter Müller。“基于 MaxSMT 的 Python 3 类型推断”。载于：计算机辅助验证。Hana Chockler 和 Georg Weissenbacher 编。Cham：Springer 国际出版公司，2018 年，第 12–19 页。isbn：978-3-319-96142-2。doi：10.1007/978-3-319-96142-2_2。[17] Nathaniel Hejduk、Ben Greenman、Matthias Felleisen 和 Christos Dimoulas。“借助性能分析器探索混合类型迁移”。载于：*ACM Trans. Program.Lang. Syst.* （2025 年 12 月）。已接受。issn：0164-0925。doi：10.1145/3785004。

[18] 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* 欧洲软件工程大会与软件工程基础研讨会联合会议录。ESEC/FSE 2018。美国佛罗里达州布埃纳维斯塔：计算机协会，2018 年，第 152–162 页。isbn：978-1- 4503-5573-5。doi：10.1145/3236024.3236051。[19] J. Roger Hindley。《组合逻辑中对象的主类型方案》。载于：《美国数学会汇刊》146（1969），第 29–60 页。issn：0002-9947。doi：10.2307/1995158。[20] Milod Kazerounian、Brianna M. Ren 和 Jeffrey S. Foster。《Ruby 的可靠启发式类型注解推断》。载于：第 *16* 届 *ACM SIGPLAN* 动态语言国际研讨会会议录。DLS 2020。美国：线上会议，计算机协会，2020 年，第 112–125 页。isbn：978-1-4503-8175-8。doi：10.1145/3426422.3426985。[21] Mohammad Wahiduzzaman Khan、Sheng Chen 和 Yi He。《学习渐进式类型的性能》。载于：第 *38* 届欧洲面向对象编程会议（*ECOOP 2024*）。Jonathan Aldrich 和 Guido Salvaneschi 编。第 313 卷。莱布尼茨国际信息学会议录（LIPIcs）。德国达格施图尔：达格施图尔城堡—莱布尼茨信息学中心，2024 年，21:1–21:27。isbn：978-3-95977-341-6。doi：10.4230/LIPIcs.ECOOP.2024.21。[22] Andre Kuhlenschmidt、Deyaaeldeen Almahallawi 和 Jeremy G. Siek。《通过强制转换实现结构类型的高效渐进式类型》。载于：第 *40* 届 *ACM SIGPLAN* 编程语言设计与实现会议录。PLDI 2019。美国亚利桑那州菲尼克斯：计算机协会，2019 年，第 517–532 页。isbn：978-1-4503-6712-7。doi：10.1145/3314221.3314627。[23] Lukas Lazarek、Ben Greenman、Matthias Felleisen 和 Christos Dimoulas。《如何评估渐进式类型中的归咎》。载于：*ACM* 编程语言会议录5.ICFP（2021 年 8 月）。doi：10.1145/3473573。[24] Lukas Lazarek、Ben Greenman、Matthias Felleisen 和 Christos Dimoulas。《如何评估渐进式类型中的归咎（第二部分）》。载于：*ACM* 编程语言会议录 7.ICFP（2023 年 8 月）。doi：10.1145/3607836。[25] Senxi Li、Feng Dai、Tetsuro Yamazaki 和 Shigeru Chiba。《为提升渐进式类型性能而高效选择类型注解》的已接收的工件。2026 年 1 月。doi：10.5281/zenodo.18415892。[26] Senxi Li、Tetsuro Yamazaki 和 Shigeru Chiba。《InferType：用于实现高效基于约束的类型推断的编译器工具包》。载于：第 *38* 届欧洲面向对象编程会议（*ECOOP 2024*）。Jonathan Aldrich 和 Guido Salvaneschi 编。第 313 卷。莱布尼茨国际信息学会议录（LIPIcs）。德国达格施图尔：达格施图尔城堡—莱布尼茨信息学中心，2024 年，23:1–23:28。isbn：978-3-95977-341-6。doi：10.4230/LIPIcs.ECOOP.2024.23。

[27] Paolo Maggi 和 Riccardo Sisto。《使用数据流分析推断 Java 字节码中的类型信息》。载于：首届 *IEEE* 源代码分析与处理国际研讨会会议录 （2001），第 211–222 页。doi：10.1109/SCAM.2001.972683。[28] Stefan Marr、Benoit Daloze 和 Hanspeter Mössenböck。《跨语言编译器基准测试：我们现在够快了吗？》。载于：第 *12* 届动态语言研讨会会议录。DLS 2016。荷兰阿姆斯特丹：计算机协会，2016 年，第 120–131 页。isbn：978-1-4503-4445-6。doi：10.1145/2989225.2989232。[29] Robin Milner。《编程中的类型多态理论》。载于：《计算机与系统科学杂志》17.3（1978），第 348–375 页。issn：0022-0000。doi：10.1016/0022-0000(78)90014-4。[30] Amir M. Mir、Evaldas Latoškinas、Sebastian Proksch 和 Georgios Gousios。《Type4Py：面向 Python 的实用深度相似性学习类型推断》。载于：第 *44* 届国际软件工程会议录。ICSE '22。宾夕法尼亚州匹兹堡：计算机协会，2022 年，第 2241–2252 页。isbn：978-1-4503-9221-1。doi：10.1145/3510003.3510124。[31] Fumika Mochizuki、Tetsuro Yamazaki 和 Shigeru Chiba。《通过卸载动态增量编译实现微控制器交互式编程》。载于：第 *21* 届 *ACM SIGPLAN* 托管编程语言与运行时国际会议录。MPLR 2024。奥地利维也纳：计算机协会，2024 年，第 28–40 页。isbn：979-8-4007-1118-3。doi：10.1145/3679007.3685062。[32] Raphaël Monat、Abdelraouf Ouadjaout 和 Antoine Miné。《通过对 Python 程序进行抽象解释的静态类型分析》。载于：第 *34* 届欧洲面向对象编程会议（*ECOOP 2020*）。Robert Hirschfeld 和 Tobias Pape 编。第 166 卷。莱布尼茨国际信息学会议录（LIPIcs）。德国达格施图尔：达格施图尔城堡—莱布尼茨信息学中心，2020 年，17:1–17:29。isbn：978-3-95977-154-2。doi：10.4230/LIPIcs.ECOOP.2020.17。[33] Colin Moock。《*Essential ActionScript 2.0*》。O'Reilly Media, Inc.，2004 年。isbn：0-596- 00652-7。[34] Fabian Muehlboeck 和 Ross Tate。《健全的渐进式类型在名义类型中依然生机勃勃且运行良好》。载于：*ACM* 编程语言会议录 1.OOPSLA（2017 年 10 月）。doi：10.1145/3133880。[35] Yun Peng、Cuiyun Gao、Zongjie Li、Bowei Gao、David Lo、Qirun Zhang 和 Michael Lyu。《静态推断结合深度学习：一种面向 Python 的混合式类型推断方法》。载于：第 *44* 届国际软件工程会议录。ICSE '22。宾夕法尼亚州匹兹堡：计算机协会，2022 年，第 2019–2030 页。isbn：978-1-4503-9221-1。doi：10.1145/3510003.3510038。[36] Benjamin C. Pierce。《类型与程序设计语言》。第 1 版。麻省理工学院出版社，2002 年。isbn：978-0-262-16209-8。

[37] Aseem Rastogi、Avik Chaudhuri 和 Basil Hosmer。“渐进式类型推断的方方面面”。载于：第*39*届 *ACM SIGPLAN-SIGACT* 编程语言原理年会论文集。POPL ’12。美国宾夕法尼亚州费城：计算机协会，2012年，第481–494页。ISBN：978-1-4503-1083-3。DOI：10.1145/2103656.2103714。[38] Gregor Richards、Ellen Arteca 和 Alexi Turcotte。“虚拟机早已知晓：利用编译时知识优化渐进式类型”。载于：*ACM* 编程语言会议录 1.OOPSLA（2017年10月）。DOI：10.1145/3133879。[39] Richard Roberts、Stefan Marr、Michael Homer 和 James Noble。“瞬态类型检查（几乎）无需成本”。载于：第*33*届欧洲面向对象编程会议（*ECOOP 2019*）。Alastair F. Donaldson 编。第134卷。莱布尼茨国际信息学会议录（LIPIcs）。德国达格施图尔：达格施图尔城堡—莱布尼茨信息学中心，2019年，5:1–5:28。ISBN：978-3-95977-111-5。DOI：10.4230/LIPIcs.ECOOP.2019.5。[40] Jeremy G. Siek、Michael M. Vitousek、Matteo Cimini 和 John Tang Boyland。“渐进式类型的改进判据”。载于：第*1*届编程语言进展峰会（*SNAPL 2015*）。Thomas Ball、Rastislav Bodík、Shriram Krishnamurthi、Benjamin S. Lerner 和 Greg Morriset 编。第32卷。莱布尼茨国际信息学会议录（LIPIcs）。德国达格施图尔：达格施图尔城堡—莱布尼茨信息学中心，2015年，第274–293页。ISBN：978-3-939897-80-4。DOI：10.4230/LIPIcs.SNAPL.2015.274。[41] Asumu Takikawa、Daniel Feltey、Ben Greenman、Max S. New、Jan Vitek 和 Matthias Felleisen。“健全的渐进式类型已死？”载于：*SIGPLAN* 通讯 51.1（2016年1月），第456–468页。ISSN：0362-1340。DOI：10.1145/2914770.2837630。[42] Sam Tobin-Hochstadt 和 Matthias Felleisen。“Typed Scheme 的设计与实现”。载于：第*35*届 *ACM SIGPLAN-SIGACT* 编程语言原理年会论文集。POPL ’08。美国加利福尼亚州旧金山：计算机协会，2008年，第395–406页。ISBN：978-1-59593-689-9。DOI：10.1145/1328438.1328486。[43] Michael M. Vitousek、Andrew M. Kent、Jeremy G. Siek 和 Jim Baker。“Python 渐进式类型的设计与评估”。载于：第*10*届 *ACM* 动态语言研讨会论文集。DLS ’14。美国俄勒冈州波特兰：计算机协会，2014年，第45–56页。ISBN：978-1-4503-3211-8。DOI：10.1145/2661088.2661101。[44] Michael M. Vitousek、Jeremy G. Siek 和 Avik Chaudhuri。“优化与评估瞬态渐进式类型”。载于：第*15*届 *ACM SIGPLAN* 动态语言国际研讨会论文集。DLS 2019。希腊雅典：计算机协会，2019年，第28–41页。ISBN：978-1-4503-6996-1。DOI：10.1145/3359619.3359742。[45] Fangke Ye、Jisheng Zhao、Jun Shirako 和 Vivek Sarkar。“利用机器学习和 SMT 求解进行代码优化的具体类型推断”。载于：*ACM* 编程语言会议录 7.OOPSLA2（2023年10月）。DOI：10.1145/3622825。

[46] Lina Zhou、Yin Qing、Liehui Jiang、Wenjian Yin 和 Tieming Liu。“一种基于数据流分析的反编译类型推断方法”。载于：*2009*年计算智能与软件工程国际会议。2009年，第1–4页。DOI：10.1109/CISE.2009.5362985。

关于作者

Senxi Li 联系方式：lisenxi@csg.ci.i.u-tokyo.ac.jp。
https://orcid.org/0009-0008-2644-7763

Feng Dai 联系方式：daifeng@csg.ci.i.u-tokyo.ac.jp。
https://orcid.org/0009-0006-0995-2536

Tetsuro Yamazaki 联系方式：yamazaki@csg.ci.i.u-tokyo.ac.jp。
https://orcid.org/0000-0002-2065-5608

Shigeru Chiba 联系方式：chiba@csg.ci.i.u-tokyo.ac.jp。
https://orcid.org/0000-0002-1058-5941