arXiv:2512.17616v1 [cs.PL] 19 Dec 2025

通过 L 系统进行多语言基准测试生成 S

Vinicius Francisco daSilva Heitor Leite UFMG，巴西
UFMG，巴西
silva.vinicius@dcc.ufmg.br heitor.leite@dcc.ufmg.br
Fernando MagnoQuintão Pereira UFMG，巴西
fernando@dcc.ufmg.br

摘要

L 系统是生物学家 Aristid Lindenmayer 提出的一种数学形式体系，旨在模拟树木、雪花、花朵及其他分枝现象等有机结构。它们以形式语言的形式实现，用于定义如何迭代地改写模式。本文介绍了如何利用这种形式体系创建以 C、C语言、Julia 和 Go 等编程语言编写的人工生成的程序。这些程序规模庞大且复杂，可用于测试编译器、操作系统和计算机体系结构的性能。本文通过多个案例研究展示了这些基准测试的实用性。这些案例研究包括：比较 `clang`和 `gcc`；比较 C、C语言、Julia 和 Go；从代码质量角度研究`gcc` 的历史演变；考察配置文件引导优化对 `gcc`的影响；分析 `clang`编译流水线不同阶段的渐近行为；以及比较 Gnome 库（GLib）中提供的多种数据结构。这些案例研究表明，与 CSmith 等模糊测试器相比，L 系统方法在创建基准测试方面具有优势；CSmith 等模糊测试器旨在发现编译器中的缺陷，而非评估其性能。

关键词： L 系统、基准测试、合成

1 引言

语言处理系统（如编译器、解释器和静态分析器）是复杂的工具，其验证依赖于用目标语言编写的程序。然而，针对任何给定编译器，可用的基准测试数量通常有限 [43]。为解决这一限制，人们开发了多种工具来自动生成测试程序 [45]。这一过程称为模糊测试 [28], ，广泛用于发现崩溃和内存泄漏等缺陷。在编译器领域，CSmith [45] 和 YARPGen [27] 等模糊测试器会生成随机 C 程序，以对编译器进行压力测试并支持静态分析。近年来，Fuzz4All [44]等模糊测试器利用大语言模型（LLM）生成测试程序，其应用范围不仅包括编译器，还包括约束求解器、解释器以及其他提供可访问应用程序接口的软件系统。

尽管程序生成器种类繁多， [45], 现有工具仍存在一些重要局限。一个关键问题是无法控制生成程序的大小。例如，使用最广泛的 C 语言编译器模糊测试工具 CSmith 不允许用户调整输出大小，而是生成大小符合正态分布的程序。使用 `clang` v9.0.1 在`-O0`编译时，这些程序平均包含 20,190 条 LLVM 指令，标准差为

3,650 条，中位数为 19,161 条指令 [13]。其他模糊测试器，例如 YARPGen、 [27], LDRGen [2],和 Orange，3 [25], 也表现出类似的情况。因此，这些工具并不适合评估编译器组件的性能，例如语法解析、语义分析和代码生成。

基于 L 系统的程序。为克服这一局限，本文提出一种用于压力测试编译器的方法，重点关注性能，而不仅仅是正确性。我们的方法能够受控地生成程序，并由用户精确调整其大小。因此，可以构造几乎任意大小的合成代码，唯一的限制是生成时间和存储空间等实际资源。

如第 2 节所述，程序通常具有递归的自相似结构。例如，控制结构（如 `if-then-else`）的分支本身也是程序，其中可能递归地包含更多控制结构。为了利用这一特性，我们引入一种基于*L* 系统 [26] 的生成技术。L 系统最初由 Aristid Lindenmayer 提出，用于模拟生物体的生长；它提供了一个基于语法的框架，我们将其扩展用于代码生成。本文展示了如何将一系列自相似程序编码为 L 文法。随后通过迭代重写生成程序，其中每个生成的字符串都描述了一个程序的蓝图，该程序围绕数组或链表等核心数据结构组织。如第 3 节所述，这项技术支持多种编程语言，能够生成复杂的控制流图，避免未定义行为，并支持对不同数据结构进行操作。

贡献总结。为了展示使用 L 系统表示程序增长的优势，我们推出了 BenchGen，这是一款多语言基准测试生成器。第 4 节通过七个案例研究对 BenchGen 进行评估，包括： `gcc`与 `clang`的比较；Go、Julia、C语言和 C++的比较；对 `clang`不同组件的渐近分析；对 `gcc`演变过程的纵向分析；对`clang`中配置文件引导优化的评估；对 GLib 数据结构的比较；以及对 BenchGen 和 Csmith 生成的程序进行比较。BenchGen 根据 GPL 3.0 许可证公开提供，且已用于为本文讨论以外的多种语言生成基准测试，包括 Rust 和 Vale。

2 L 系统

L 系统（或林登迈尔系统）是一种基于重写规则的形式模型，最初用于描述植物及其他类分形结构的生长模式。它由符号字母表、一组定义符号如何转换的产生式规则，以及作为起点的初始字符串（即公理）组成。在每次迭代中，规则都会递归地应用于当前字符串，从而生成日益复杂的序列。示例 1展示了这一形式体系的运作方式。

示例 1. 图 *1* 展示了一个 L 系统的示例。该系统使用的规则通过字符串重写生成几何图案。从公理 A 出发，产生式规定了符号在每一步如何演变： $A \to B - A - B$ 和 $B \to A + B + A$。其中，符号 $-$和 $+$ 分别表示旋转 *60* 度和 *300* 度。反复应用这些规则会生成一系列序列，将其以图形方式进行解释后，便可得到复杂的分形曲线，例如著名的谢尔宾斯基三角形。

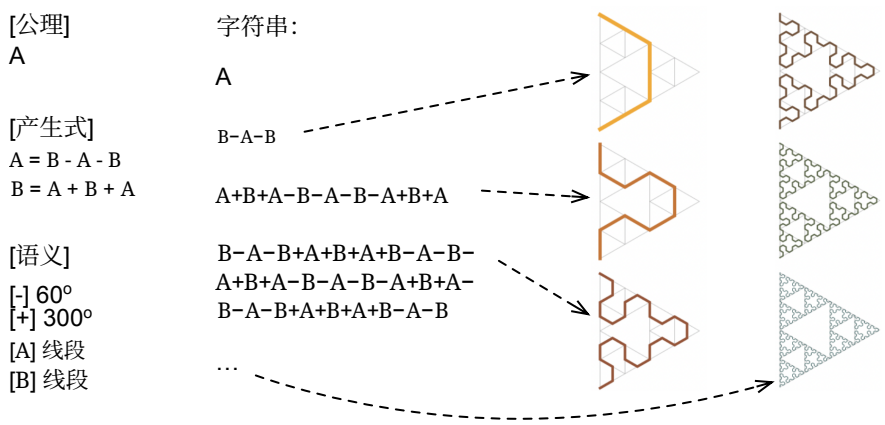

图 1：描述谢尔宾斯基三角形的 L 系统。

2.1 程序作为自相似结构

L 系统具有一种称为 自相似性 的性质，这意味着结构在不同尺度下都包含自身的较小副本。在 L 系统中，这一特性自然源于重写规则的递归应用，由此生成的图案在逐渐精细的层次上保持相同的形状。这种行为在示例 1 中的谢尔宾斯基三角形等分形中显而易见，其中每个组成部分都是整体按比例缩小后的副本。这种层级式重复是对植物生长、海岸线和树木分枝等现象进行建模的关键。

自相似性也存在于计算机程序中，程序通常具有递归和层次结构。许多程序由较小的函数组成，这些函数可以调用自身，也可以嵌套在彼此之中，如 `if-then-else` 代码块或循环。模块化和代码复用进一步强化了这一模式：通用例程可以在不同抽象层级上反复实例化，正如示例 2 所述。

示例 2. 图 *2* 展示了如何通过嵌套的 *if-then-else*代码块来体现代码中的自相似性。起初，函数 *g(x)*包含一个条件语句。然而，它可以递归扩展为 *g(x) = if g(x) then g(x) else g(x)*，从而形成自指结构。这类递归定义自然会产生自相似性，并常见于编码程序控制流的语法结构中。

构想概要本文利用自相似性原理，生成定义良好且复杂度任意的 C 程序。生成模型基于一种 L 系统语法，类似于示例 1 所示的语法，但它合成的不是几何图案，而是具有可执行语义的 C 代码结构。图3展示了 BenchGen 为 C语言和 Julia 生成的同一程序的两个版本示例。该文件是一个包含 52 个文件的基准测试的一部分，我们在第 4.2 节中使用该基准测试比较不同语言处理系统的性能。

3 基于 L 系统的代码生成

本文开发的工具 BenchGen 会根据图4所示的模式生成操作数据结构的程序。第3.1节介绍程序构建中使用的核心构建模块，而第3.2节描述这些构建模块的语义。

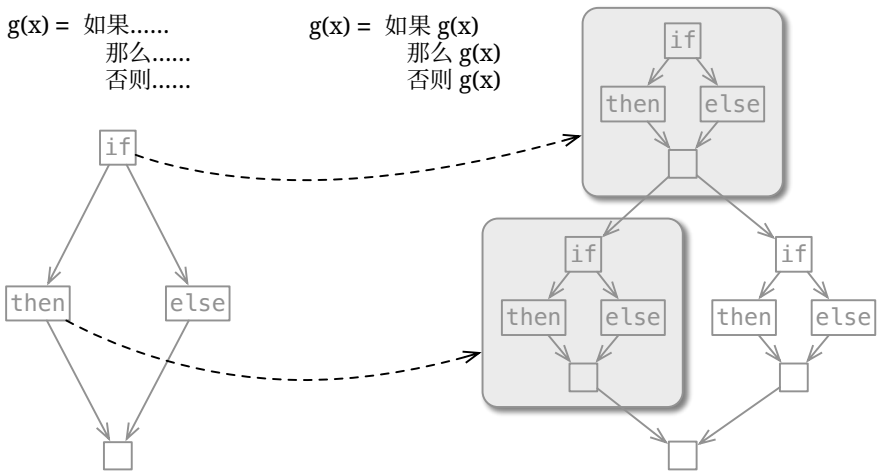

图 2：计算机代码的自相似特性。

3.1 语法构建块

本文所述的 L 系统由两类构造组成 :

- **结构：** 定义程序控制流的元素，包括 `IF`、 `LOOP` 和 `CALL`。
- **行为：** 指定如何操作数据的操作，包括 `new`、 `insert`、 `remove` 和 `contains`。

3.1.1 结构块

BenchGen 生成的程序中的控制流源于组合四种类型的代码块，如下方语法所示：

$$
\begin{array}{rcll}
b & ::= & \mathrm{IF}(b_{\mathrm{cond}}, b_{\mathrm{then}}) & ;;\mathit{if_then} \\
& | & \mathrm{IF}(b_{\mathrm{cond}}, b_{\mathrm{then}}, b_{\mathrm{else}}) & ;;\mathit{if_then_else} \\
& | & \mathrm{LOOP}(b_{\mathrm{cond}}, b_{\mathrm{body}}) & ;;\mathit{while} \\
& | & \mathrm{CALL}(b) & ;;\mathit{function_call}
\end{array}
$$

这些结构分别对应于熟悉的编程结构：条件分支（`if-then`、 `if-then-else`）、循环（`while`）和函数调用。示例 3 对此进行了说明。

示例 3. 图 *5* 展示了一种用于综合程序的语法。右侧说明了该 L-文法的两个推导步骤，以及根据第二个推导生成的相应（简化）*C* 程序。

3.1.2 行为块

BenchGen 程序的动态行为源于与数据结构的交互。程序中的所有数据结构必须具有相同的类型。目前，BenchGen 支持整数数组和有序整数链表。此外，C 代码生成器还支持 GNOME 库中提供的十二种数据结构中的任意一种。要支持新建容器，需要扩展四个 C++ 类。这些类定义了 L-文法中四种构造的行为：

C 基准测试（52 个 .c 文件）中的一个函数

```
#include "program_c.h"
数组_t* 函数42
(数组_t_参数* vars, const 无符号长整型 PATH0, int loopsFactor) {
  大小_t p计数器 = vars->大小;
  if(PATH0 & 1) {
    数组_t_参数 参数0;
    params0.大小 = 0;
    params0.数据 = (数组_t**)内存分配(params0.大小*sizeof(数组_t*));
    数组_t* 数组29 = 函数43(&params0, loopsFactor);
    DEBUG_RETURN(array29->标识符);
    free(params0.数据);
    array29->refC--;
    if(array29->refC == 0) {
      free(array29->数据);
      free(数组29);
      DEBUG_FREE(array29->标识符);
    }
  }
  else {
    数组_t_参数 参数0;
    params0.大小 = 0;
    params0.数据 = (数组_t**)内存分配(params0.大小*sizeof(数组_t*));
    数组_t* 数组31 = 函数44(&params0, 获取_路径(), loopsFactor);
    调试_返回(数组31->标识符);
    释放(params0.数据);
    数组31->refC--;
    if(数组31->refC == 0) {
      free(array31->数据);
      free(array31);
      DEBUG_FREE(array31->标识符);
    }
  }
  数组_t_参数 params0;
  params0.大小 = 0;
  params0.数据 = (数组_t**)内存分配(params0.大小*sizeof(数组_t*));
  数组_t* 数组35 = func45(&params0, loopsFactor);
  DEBUG_RETURN(数组35->标识符);
  free(params0.数据);
  return 数组35;
}
```

控制流 (第3.2节)

函数调用 (第3.3节)

内存管理 (第3.4节)

类似 Julia 基准测试中的一个函数

```
include("程序_julia_head.jl")
函数 func42
(vars::JArray_参数, PATH0::UInt64, loopsFactor::Int)::JArray
  pCounter = 长度(vars.数据)
  如果 (PATH0 & 1) != 0
    参数0 = JArray_参数{Int}()
    params0.大小 = 0
    数组29 = 函数43(参数0, loopsFactor);
    数组29.refC -= 1
    如果 数组29.refC == 0
      empty!(数组29.数据)
      数组29 = 无
    end
  else
    参数0 = JArray_参数{Int}()
    参数0.大小 = 0
    数组31 = 函数44(参数0, 获取_路径(), loopsFactor);
    数组31.refC -= 1
    如果 数组31.refC == 0
      空！(数组31.数据)
      数组31 = 无
    end
  end
  参数0 = JArray_参数{Int}()
  参数0.大小 = 0
  数组35 = func45(参数0, loopsFactor);
  返回 数组35;
end
```

结构与行为 (第3.1节)

多语言支持 (第 3.5 节)

图3：BenchGen 为操作数组的 C语言和 Julia 生成的代码示例。

- **新建：** 创建并初始化一个数据结构。
- **插入：** 将一个元素添加到作用域中的数据结构。
- **移除：** 从作用域中的数据结构移除一个元素。
- **包含：** 检查作用域中的数据结构是否包含某个元素。

请注意，这四个行为块可由 BenchGen 用户以任何方式自定义。例如，BenchGen 提供了“标量模式”，用于生成操作整数变量的程序。在此情况下，我们将 `insert` 配置为递增变量； `remove` 配置为递减变量；`new` 配置为声明变量并将其初始化为零； `contains` 配置为测试变量是否为零。下面以更具体的例子说明如何为处理数组的程序定义这类操作。

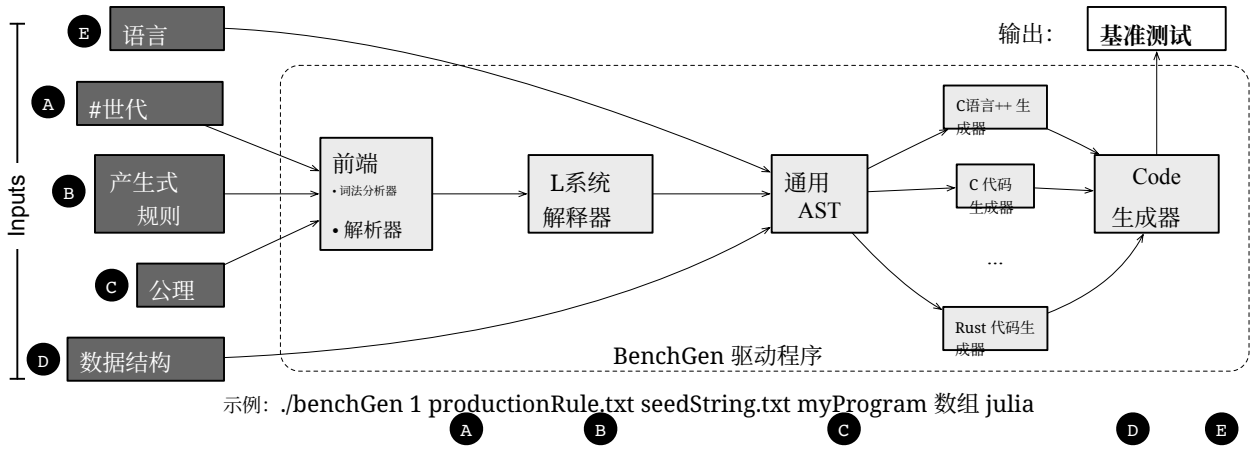

图4：BenchGen 是一个可生成不同编程语言程序的工具。它接收一个 L 系统的描述作为输入，该描述由一组产生式规则和一个起始符号（即公理）组成。它会解释这些规则，生成字符串（即 L 字符串），用于指导通用抽象语法树的构建。这些树可通过不同的容器转换为不同编程语言的程序。随后，代码生成器会将特定于语言的树转换为程序。

示例 4. 图*6*展示了用于操作数组的 *C*代码。尽管程序可以处理多个数组，此例仅聚焦于特定变量 *array156*。用户必须自定义操作 *new()*、*insert()*和 *contains()*，以定义生成代码的行为。

3.2 执行流程

BenchGen 生成的程序均可执行。其控制流由一个名为 `PATH`、类型为 `unsigned long`的变量控制，该变量决定条件分支的结果。图 1 中的 BitPath 算法将分支结果与此 `PATH`变量相关联。该算法使用计数器栈，确保每条路径都能通过 `PATH` 变量被唯一标识。它通过以下方式为每个分支条件分配唯一的位掩码：

1.**跟踪嵌套深度**：使用栈管理每个嵌套分支层级的计数器。2. **处理合并点**：在分支合并时向上传播“最大计数器”，确保位掩码不会发生冲突。

算法 1 确定控制流走向的方式，使我们能够使用 BenchGen 测试配置文件引导优化的效果，正如第4.5节将展示的那样。示例 5 对这一机制作了说明。

示例 5。 图 *7* 展示了算法 *1* 创建的掩码与包含 *PATH*变量的程序的控制流图。请注意，用作位掩码的计数器会因嵌套或顺序执行而更新。嵌套会使计数器栈顶端的计数器递增。每当我们分析一个分支时，其计数器值将是所有到达该分支的可能控制流中的最大值。

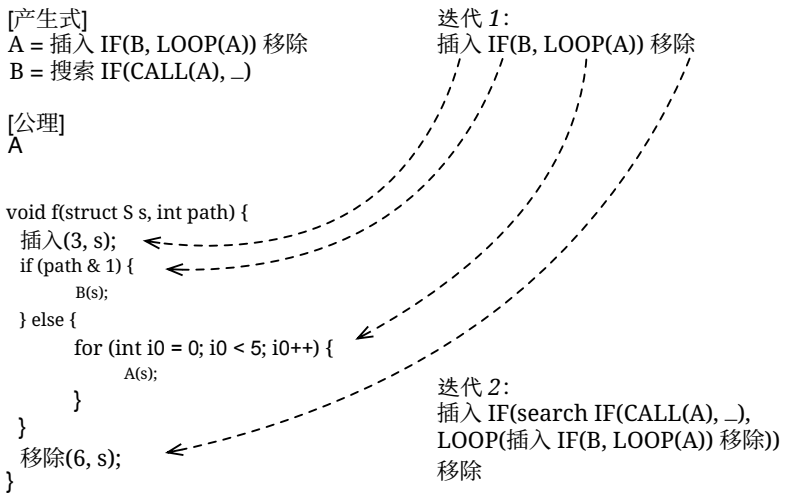

图 5：用于定义程序的 L-文法示例。图中展示了一个简化版的 C 程序。图3提供了更具体的示意。

3.3 函数调用

BenchGen 通过 `CALL`子句支持函数调用。当 L-字符串包含形式为 `CALL`(e) 的构造时，会提取整个子字符串 e ，并将其定义为单独的函数。根据 BenchGen 中的配置参数，由给定 L-规范生成的函数可以归入单个文件，也可以分散到多个文件中。示例 6 展示了 BenchGen 如何生成过程间代码。

示例 6. 图 *8* 描绘了由 a *CALL* 代码块生成的控制流。括起来的字符串会生成一个新函数，并成为合成程序的一部分。该新函数会在 L-字符串中出现 *CALL* 子句的位置被调用。

所有具有相同字符串的 `CALL`(b)出现形式 b 都会调用同一个函数。为避免冗余，BenchGen 会维护一个将字符串映射到函数的表，确保相同的字符串指向同一个函数实例。示例 7 展示了如何多次调用同一个函数。请注意，BenchGen 不支持创建递归函数调用。如果一个函数 `f0` 调用另一个函数 `f1`，那么 `f0` 由一个严格大于生成 `f1`的字符串所生成。

示例 7. 图 *9* 展示了一个生成含有两个函数的程序的 L 系统， *f0* 和*f1*。后者被调用两次，因为它由一个在扩展过程的某次迭代中出现两次的字符串生成。

参数传递。 BenchGen 生成的函数接收两个参数：

- **数据：** 一个数据结构数组，用于在调用方和被调用函数之间共享数据。
- **路径：** 一个控制变量，用于控制执行流程，如示例 5 所述。

该**数据** 数组通过到达定义分析进行填充，该分析可确定调用函数中哪些变量可在调用点作为实参传递。示例 8 展示了这一机制。

```
new
    数组_t* array156;
    数组156 = (数组_t*)内存分配(sizeof(数组_t));
    array156->大小 = 42;
    array156->refC = 1;
    array156->标识符 = 156;
    array156->数据 = (unsigned int*)内存分配(array156->大小*sizeof(unsigned int));
    memset(array156->数据, 0, array156->大小*sizeof(unsigned int));
    DEBUG_新建(array157->标识符);
插入
    unsigned int loop46 = 0;
    unsigned int loopLimit46 = (rand()%loopsFactor)/2 + 1;
    for(; loop46 < loopLimit46; loop46++) {
      for (int i = 0; i < array156->大小; i++) {
        array156->数据[i]--;
      }
    }
包含
    无符号整数 loop46 = 0;
    无符号整数 loopLimit46 = (rand()%loopsFactor)/2 + 1;
    对于 (整数 i = 0; i < array156->大小; i++) {
      如果 (array156->数据[i] == 61) {
        中断;
      }
    }
```

图6： `new`、 `insert` 和 `contains` 的块定义示例。

示例 8。 图 *10* 展示了 *BenchGen* 生成的程序中的参数传递。在函数调用点，到达定义分析识别出三个可用变量： *array1*、*array156* 和 *array157*。这些变量的指针被插入到 *param.data* 结构中，并传递给函数 *func0*。在被调用方内部，传入的变量通过 *new*结构复制到新变量中。每个变量恰好复制一次。如果没有其他可供复制的参数，后续的 *new* 操作将创建新的变量，如示例 *5* 所示。

3.4 内存管理

BenchGen 生成的程序不会发生内存泄漏，尽管它们经常依赖堆分配。为防止泄漏，BenchGen 使用引用计数垃圾回收器。这种方法会跟踪每个动态分配对象所具有的引用（指针）数量。创建引用时，计数递增；移除引用时，计数递减。一旦计数变为零，该对象便不再可达，可以安全地释放。为支持这一机制，BenchGen 创建的每个结构都包含一个额外字段 `refC`，用于存储当前引用计数。示例 9 展示了 BenchGen 如何使用引用计数来防止内存泄漏。

示例 9. 图*11* 展示了 *BenchGen* 如何使用引用计数来管理内存。在此示例中，两个变量 *x*和 *y*最初分别指向两个独立的堆分配结构。每个结构都包含一个 *refC* 字段，用于保存指向该对象的活动引用数量。发生赋值 *x := y* 时，原先由*x* 指向的结构的引用计数会递减，因为 *x* 不再引用它。如果这次递减使 *refC* 的值降为零，该结构就会自动释放。与此同时，被指向的结构的引用计数

算法 1： 为分支条件分配唯一位掩码的 BitPath 赋值

输入： 一个函数 f

输出： 将 f 中的每个分支条件映射到唯一位掩码

初始化栈：压入一个计数器 $Counter_0 = 1$，表示最低有效位（LSB）；

为分支分配掩码:;

相同嵌套层级：使用栈顶的当前计数器生成位掩码，例如，`PATH & (1 « (counter - 1))`;

嵌套分支：压入一个新建计数器，其值比父级计数器递增 1，确保更深层嵌套使用新的位；

处理汇合点（在 `if-then-else` 之后）：

从栈中弹出当前计数器；

将父级计数器（此时位于栈顶）更新为以下值中的最大值：；

(a) 父级原始值；

(b) 弹出的（子级）计数器；

这可确保父级分支能够计入其下方的最深嵌套层级；

独立分支：合并点解析后，同一层级的后续分支会重复使用更新后的父级计数器；

由 y 指向的结构的引用计数会递增，以计入来自 x的新引用。赋值后， x 和 y都指向同一个对象，其 $refC$ 现在反映了两个活动引用。此机制确保堆分配内存在不再可达时立即得到回收，从而防止 *BenchGen* 生成的程序出现内存泄漏。

3.5 多语言支持

BenchGen 目前提供测试版，支持为多种编程语言创建基准测试。其中一些语言将在第 4.2 节中评估。基准测试生成器本身使用 C语言++ 实现。要支持一种新语言，用户必须实现新的 C语言++ 类，以定义第3.1节所述结构块和行为块的语义。此过程还需要修改用于为结构块生成代码的模板（`IF`、 `LOOP` 和 `CALL`），以及用于为行为块生成代码的模板（`insert`、 `remove`、 `contains` 和 `new`）。

因此，扩展 BenchGen 需要重新编译，目前尚未实现完全自动化。除了实现新的 C语言++ 类，用户还必须在 BenchGen 主驱动程序中注册语言模块，这需要修改驱动程序本身中的一行代码。未来，我们计划将这一过程完全自动化，并使该工具与其支持的特定语言解耦。为此，我们正在考虑为 BenchGen 扩展一种领域专用语言，让用户能够以声明式方式指定新的代码生成指令。

4 实验评估

本节旨在展示如何在实践中使用 BenchGen。为此，我们将在以下案例研究中探讨其用法：

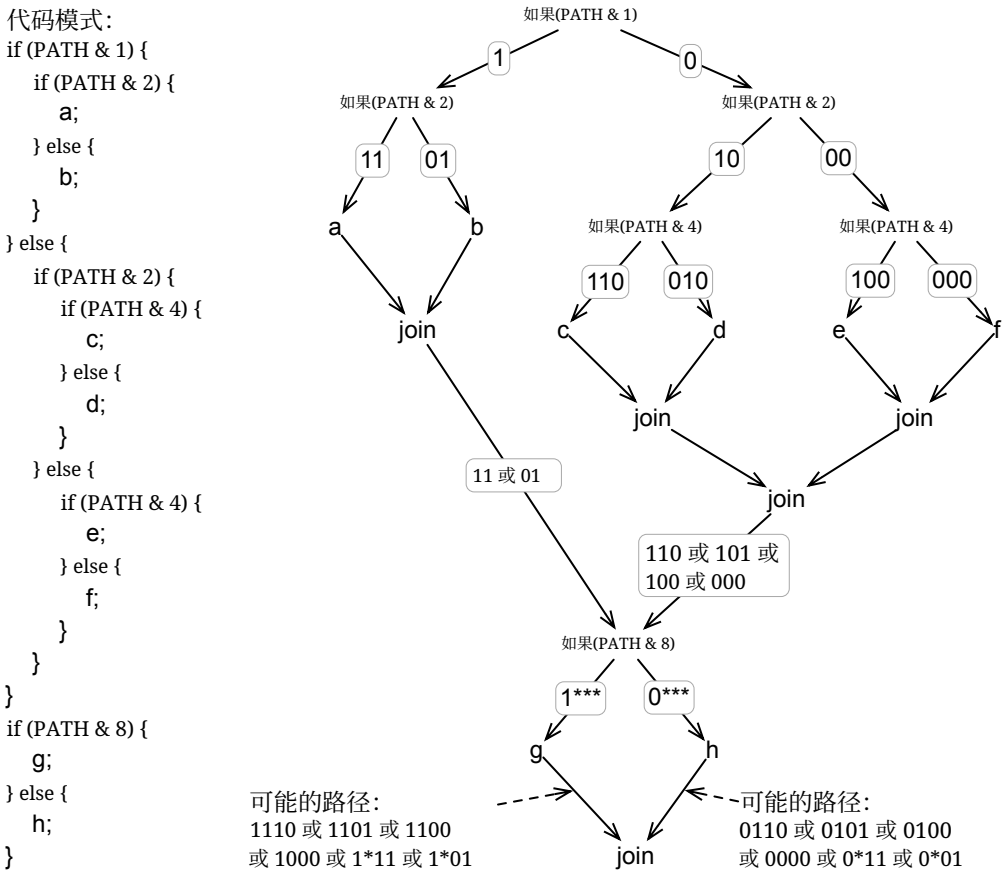

图 7 : 合成程序的控制流由 path 参数决定 .

- **CS1**：比较 gcc和 clang在执行速度、二进制文件大小和编译时间方面的表现。
- **CS2**：Gnu Compiler Collection 中 C语言和 C++ 编译器后端的比较。
- **CS3**：分析 clang 和 gcc中不同优化级别的渐近行为。
- **CS4**：考察 gcc 从版本 5 到 14 的演变，重点关注执行速度、二进制文件大小和编译时间。
- **CS5**：评估 clang中配置文件引导优化的影响。
- **CS6**：比较 Gnome 库（GLib）中不同数据结构的插入、搜索和删除时间。
- **CS7**：比较 BenchGen 与编译器模糊测试工具 CSmith。

实验设置：所有实验均在运行频率为 2.80 GHz 的 Intel(R) Xeon(R) CPU E5-2680 v2 上进行，操作系统为 Linux Ubuntu 5.15.0-139-generic。所用的具体编译器和库版本在各个案例研究中均有详细说明。

检查未定义行为： 我们使用四种能够检测未定义行为的工具，分析了第 4.1 节中评估的 C 程序：来自 clang [40] 的 UBSan 和 ASan、

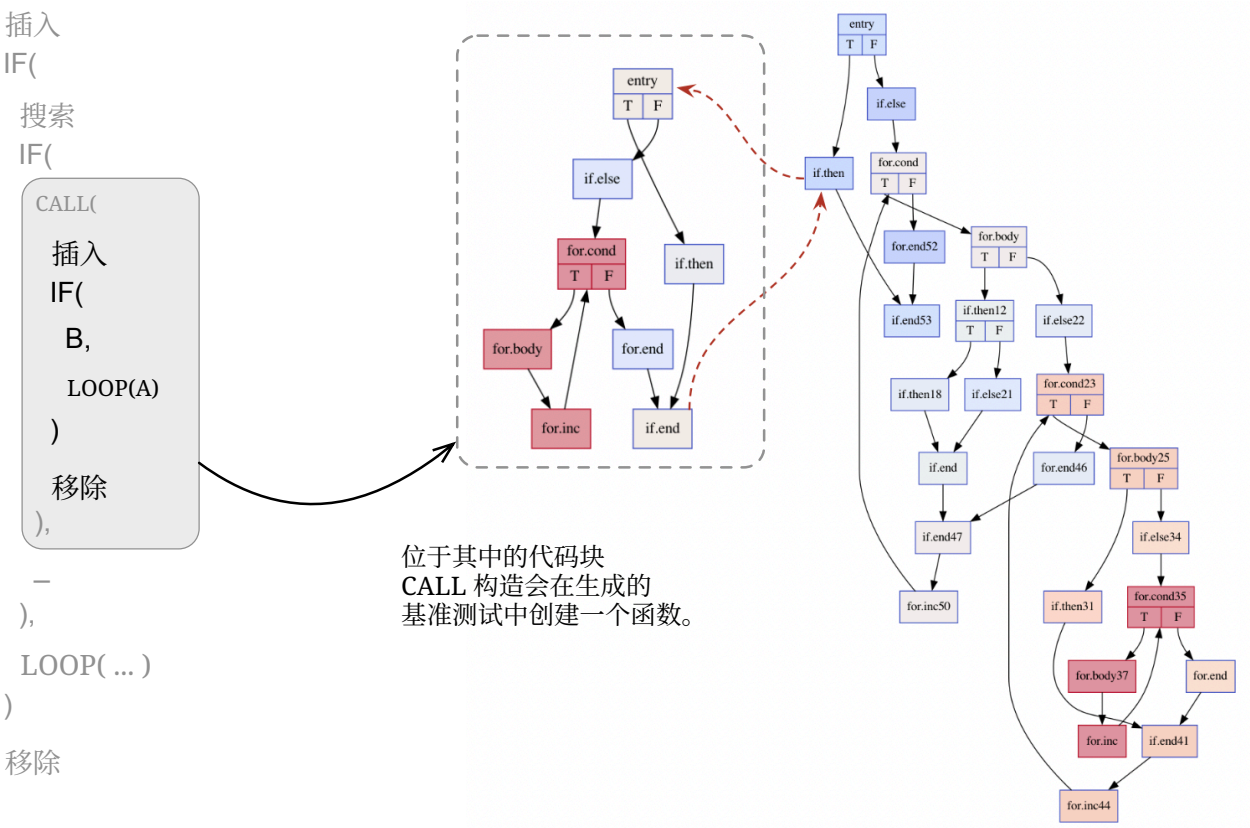

图 8：包含 CALL代码块的程序的控制流。

Valgrind [33], 、Frama-C 的 EVA 插件 [4], 以及 KCC [14]。此外，我们验证了在第 4.2 节中评估的每个程序在调试模式下执行时，都会产生相同的轨迹[1]。虽然这些措施无法从形式上证明 BenchGen 始终生成可靠的基准测试，但它们有力地表明这一点极有可能成立。

4.1 CS1： gcc 与 clang

GNU C 编译器（gcc）和 LLVM 的 clang 是使用最广泛的两种 C 编译器，文献中常见对它们进行比较。本案例研究通过比较这两种编译器在以下三项性能指标上的表现，为这场持续的讨论补充了一个数据点：

- 编译后程序的执行时间。
- 构建程序所需的编译时间。
- 编译后程序的二进制文件大小。

比较涵盖了 gcc 14.2 的六个优化级别： -O0、 -O1、 -O2、 -O3、 -Os 和 -Ofast；以及 clang 21.0 的七个优化级别： -O0、 -O1、 -O2、 -O3、 -Oz、 -Os 和 -Ofast。二进制文件大小通过 Linux size 命令收集，仅考虑 text 节；该节表示可执行代码的字节大小。编译和执行时间使用 hyperfine 测量，

[1]BenchGen 提供一种调试模式，在该模式下，每个程序都会记录第 3.1.2 节中定义的每项行为操作。

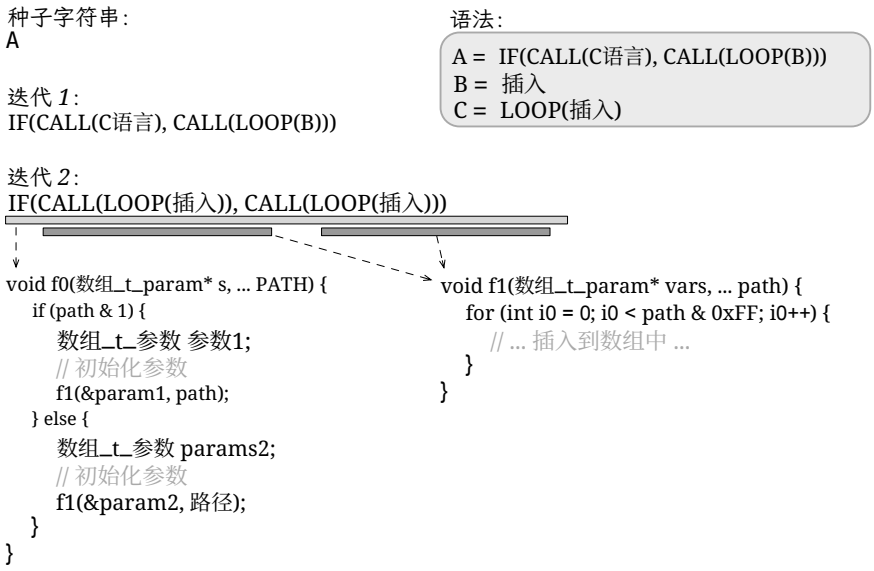

图 9：一个能够生成对同一函数的多次调用的 L 系统。

配置了三次预热运行和至少十次基准测试运行。为了比较编译器，我们使用四种不同的 L 系统生成了四个程序。我们选择了能够确保执行时间各不相同的世代；因此，这个定制基准测试集包含编译后使用 `gcc -O3` 运行时间分别为 10、50 和 200 秒的程序。

讨论。 图 12 和图 13 总结了这项比较的结果，并在每张图下方标明了产生这些数据的 L 系统及其生成。散点图展示了编译时间与执行时间的关系，点的大小与二进制文件大小成正比。我们观察到，在最低和最高优化级别下， `gcc` 和 `clang` 生成的程序往往具有相近的执行时间；不过， `gcc` 的编译时间通常更短。

`gcc`呈现出明显的趋势：它往往在编译时间与执行时间之间形成平滑的权衡，构成一条帕累托式曲线，通常编译时间越长，执行速度越快。 `clang`的这种模式则不那么明显。相反， `clang` 在最低优化级别（`-O0`、 `-O1`）和更高级别之间呈现出明显差异，而最激进的优化（`-O2`、 `-O3`和 `-Ofast`）结果非常相似。这一观察与 Curtsinger 和 Berger 在关于 Coz 的研究 [12]，中的发现相呼应：他们报告称，在 `clang`[2] 中， `-O2` 和 `-O3` 之间不存在统计学上的显著差异。就二进制文件大小而言，两种编译器在 `-Os`下生成的可执行文件都更小，其中 `clang`在 `-Oz` 级别略占优势（2–3%）。不过，两者从 `-O3` 切换到 `-Os`时，都能将代码大小减少近 50%。

4.2 CS2：不同编程语言之间的比较

如第 3.5 节所述，BenchGen 可以自定义为合成多种编程语言的基准测试。本节利用其多语言支持，对 C、C++、Julia、Go、Zig、V 和 Odin 进行比较。目前，BenchGen 的官方代码库已支持这些语言。为进行这项比较，我们使用了图12所示的第一个语法

[2]查看 `https://youtu.be/r-TLSBdHe1A?t=1438` 于 23:58

实现 **CALL** 构造的调用函数代码

```
数组_t_param params1;
params1.大小 = 3;
params1.数据 = (数组_t**)
    内存分配(params1.大小*sizeof(array_t*));
params1.数据[0] = 数组0;
params1.数据[1] = array156;
params1.数据[2] = array157;
数组_t* 数组158 = 函数6(&params1, 路径);
```

可达定义分析
确定有三个变量
在调用点可用。这些
变量作为
参数传递给被调用的函数。

被调用函数的代码，用于实现 **新建** 构造

```
数组_t* 函数6(数组_t_param* vars, const 无符号长整型 PATH0) {
  大小_t p计数器 = vars->大小;
  数组_t* 数组1;
  if (p计数器 > 0) {
    数组1 = vars->数据[--p计数器];
    数组1->refC++;
  } 否则 {
    数组1 = (数组_t*)内存分配(sizeof(数组_t));
    数组1->大小 = 386;
    数组1->refC = 1;
    数组1->标识符 = 1;
    数组1->数据 = (unsigned int*)内存分配(数组1->大小*sizeof(unsigned int));
    memset(array1->数据, 0, array1->大小*sizeof(unsigned int));
  } ... }
```

如果仍有参数
可用，则 NEW 构造
会复制其中一个参数
到新建的变量。

否则，会新建一个数据结构
如图4所示。

图 10：BenchGen 创建的程序中的参数传递。

来生成程序的十一个世代，并报告第 11 世代的五次运行结果。各语言的程序使用以下工具执行：

C语言： `gcc` 13.3.0**C语言++：** `g++` 13.3.0

Go： `go` 1.25.0

Julia： `julia` 1.11.6

Odin： `odin`dev-2015年-11

V： `vlang` 0.4.12

Zig： `zig` 0.16.0

请注意，Julia 以解释执行模式运行，并采用即时编译；其他语言则是提前编译。V 有两种执行模式：原生模式和 C 后端。前者将 V 直接编译为机器码；后者则先将 V 转换为 C，再使用 `clang-22` 从这份 C 源代码生成二进制代码。在这种情况下，我们使用优化级别 `-O3`，与 C 和 C++ 中的做法相同。我们纳入了两个版本的 C++：一个使用普通数组，另一个使用`std:vector`，只是为了让读者了解图 14 所示的相对运行时间。此评估仅考察执行时间，不包括编译时间。此外，除 C 和 C++ 会多生成一个头文件外，各语言生成的源文件数量相同。例如，在第 11 代中，BenchGen 为 Julia 和 Go 生成 52 个文件，为 C 和 C++ 生成 53 个文件。

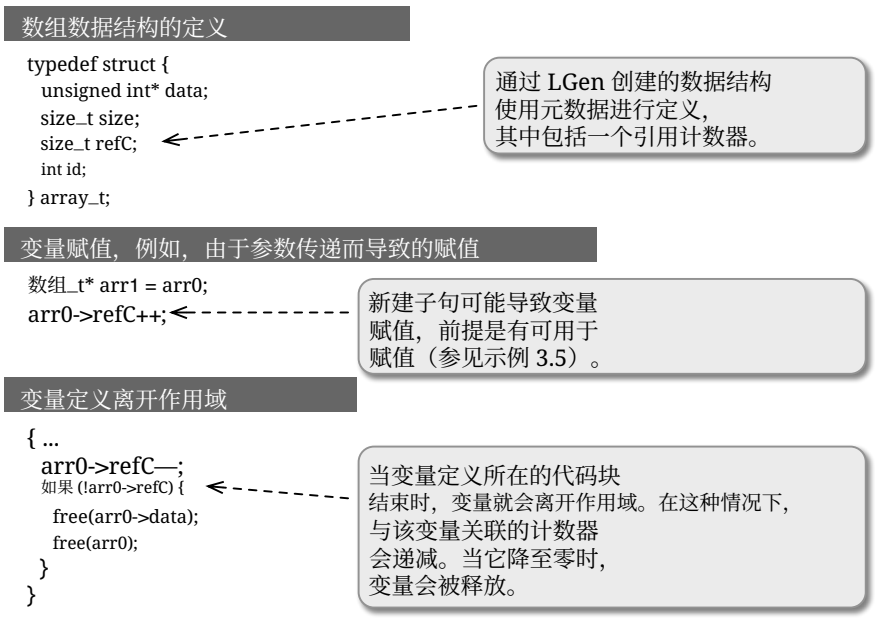

图11：引用计数器的实现。

讨论。 图 14 比较了不同语言的执行时间。结果基本符合预期：它们突显了低级静态编译语言（C 和 C++）与采用不同编译模型的高级语言（Go 和 Julia）之间的性能差异。虽然我们没有列出第 1 至第 10 代程序的运行时间，但需要强调的是，随着程序深度增加，所有语言的执行时间都会呈指数增长。这是基于分形的基准生成器固有的特性。除这一总体趋势外，我们还注意到以下几项具体发现：

- **C语言与 C语言++ 搭配数组**。性能几乎完全相同：比较同一世代的程序时，在 95% 的置信水平下，差异没有统计显著性。这是因为两者都使用相同的编译器基础设施（`gcc 13.3.0`）进行编译，生成的汇编代码高度相似。

- **Julia**。Julia 的性能出色，尤其是在处理较大的程序时——这一表现已在先前的研究中得到观察 [7]。其基于 LLVM 构建的即时编译器（JIT）可通过依据运行时类型和执行模式对代码进行特化，应用循环展开和 SIMD向量化等激进的运行时优化。随着后续世代中程序复杂度的增加，这些优化使 Julia 获得显著优势，性能超越搭配向量的 C语言++ 。

- **C语言++ 使用向量**。使用 `std::vector`的程序比使用原始数组的程序慢。这种开销源于额外的元数据（大小、容量），以及容量超出时重新分配的成本；重新分配需要将元素复制到新的内存中。在数据快速增长的基准测试中，这种重新分配成本十分显著。事实上，从第 8 代到第 9 代，Julia 的 JIT 优化使其性能超过了使用向量的 C语言++。

- **Go**。 Go 始终是运行速度最慢的语言。这并不令人意外，也符合开放论坛和技术书籍中已有的报告和讨论趋势 [41]。虽然 Go 是编译型语言，但它的

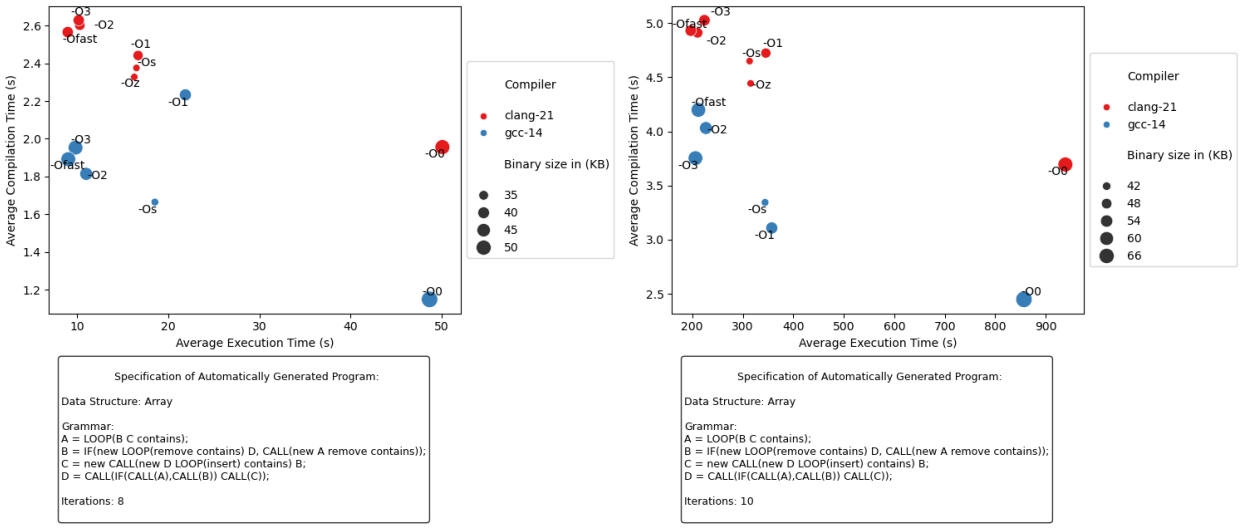

图12：在不同优化级别下对比 `gcc` 14.2 和 `clang` 21.0（第 1 部分）。

运行时间包括垃圾回收，且采用的优化较为保守，因此会产生 C、C++ 中不存在的开销。

- **V.** V 的原生模式编译器与 V 到 C 转译器之间存在显著差异。前者在第 11 代上运行程序平均需要约 3,200 秒，而后者只需 270 秒。这一差异凸显了编译器优化的影响：原生代码生成器基本没有使用优化，而 C 后端则大量使用了优化。

图 14 中观察到的趋势与知名的“Benchmark Game”网站[3]上记载的情况一致。不过，两者的实现策略存在一个关键区别。Benchmark Game 比较的是手工优化的解决方案，这些方案在语法和语义上可能各不相同；而 BenchGen 生成的代码在不同语言之间结构相似。通过消除不同算法实现所带来的变异性，这种方法能够更精确地比较编译器性能。

4.3 CS3：渐近行为

BenchGen 能够逐步改变生成程序的大小，因此适合用于对语言处理系统进行经验性渐近分析。本节将通过实证评估 `clang` 的不同阶段（前端、中端和后端）的渐近复杂度，以及 `gcc` 的完整编译流水线，展示这一能力。为此，我们采用标准 L 系统生成一个程序的八个世代（从第5个到第12个），并将这些程序输入编译器。

讨论。 图 15 展示了不同编译阶段的运行时间与目标程序每一世代所处理的二进制文件大小之间的关系。此分析得出了以下几点见解：

[3]可在此处获取 `https://benchmarksgame-team.pages.debian.net/benchmarksgame/` 截至 9 月 20 日 .

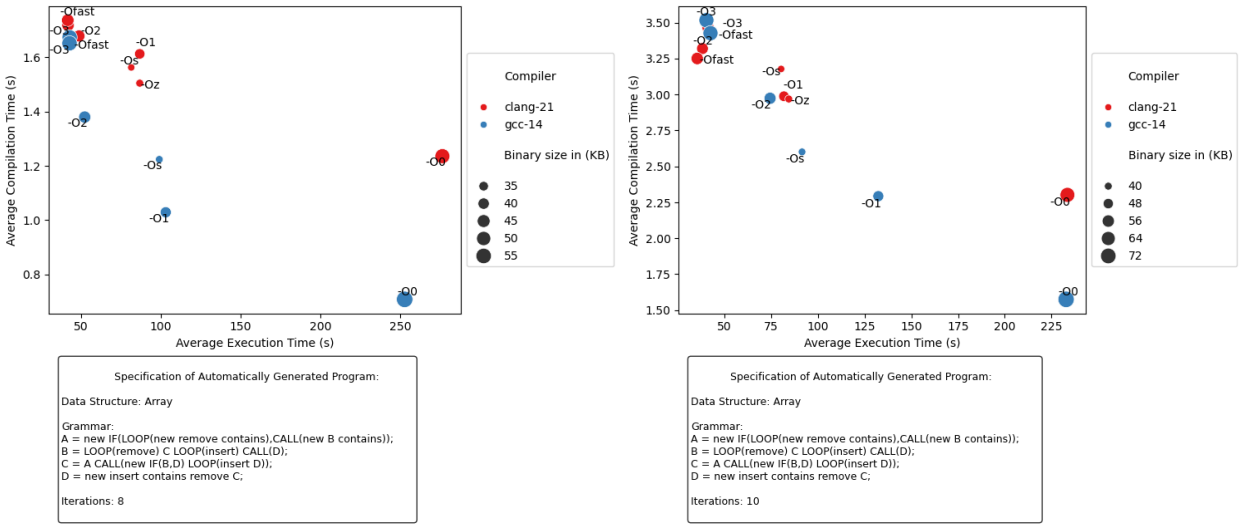

图13：在不同优化级别下对比 `gcc` 14.2 和 `clang` 21.0（第 2 部分）。

- 所有分析的工具在处理大型程序时都表现出线性行为，不同编译器和优化级别之间的相关性都非常强（高于 0.9）。这同时适用于 `clang` 和gcc。然而，对于 `clang` 的前端，由于解析器和 IR 生成器的启动开销较大，只有在后期世代（第9–12次迭代）中才显现出线性增长。

- `clang` 较高优化级别（`-O2`、 `-O3` 和 `-Ofast`）之间的统计差异很小。这一观察结果印证了第 4.1 节报告的发现。至少在本实验中，这一结论似乎适用于 `clang` 编译流水线的所有三个阶段，在 `llc`（机器码生成器）中尤为明显。

- 尽管 `gcc` 在所有优化级别下也表现出明显的线性行为，但常数因子差异显著。大小优化级别（`-Os`）的增长最为陡峭，这符合预期，因为 `gcc -Os` 会生成最小的二进制文件。

总体而言，这些结果表明，至少在编译 BenchGen 程序时， `clang` 和 `gcc` 均未表现出以往工作 [34, 19] 中报告的那类渐近性能缺陷。

4.4 CS4： `gcc` 的演进

GNU `gcc` 目前为版本 14，是历史最悠久、使用最广泛的 C 编译器之一。多年来，它经历了许多变化，包括支持其他语言以及持续提升性能。本案例研究使用三项指标，在多个优化级别下比较 `gcc` 的不同版本：

- 编译后程序的执行时间（使用 `hyperfine` 测量）。
- 构建程序所需的编译时间（同样使用 `hyperfine` 测量）。
- 可执行文件中 `.text` 段的二进制文件大小（使用 Linux `size` 实用程序收集） .

在此次比较中，我们使用了 BenchGen 和图 12 中展示的第一个 L 系统。该语法在第 10 次迭代时生成的程序，使用 `gcc` 的六个版本进行了编译。

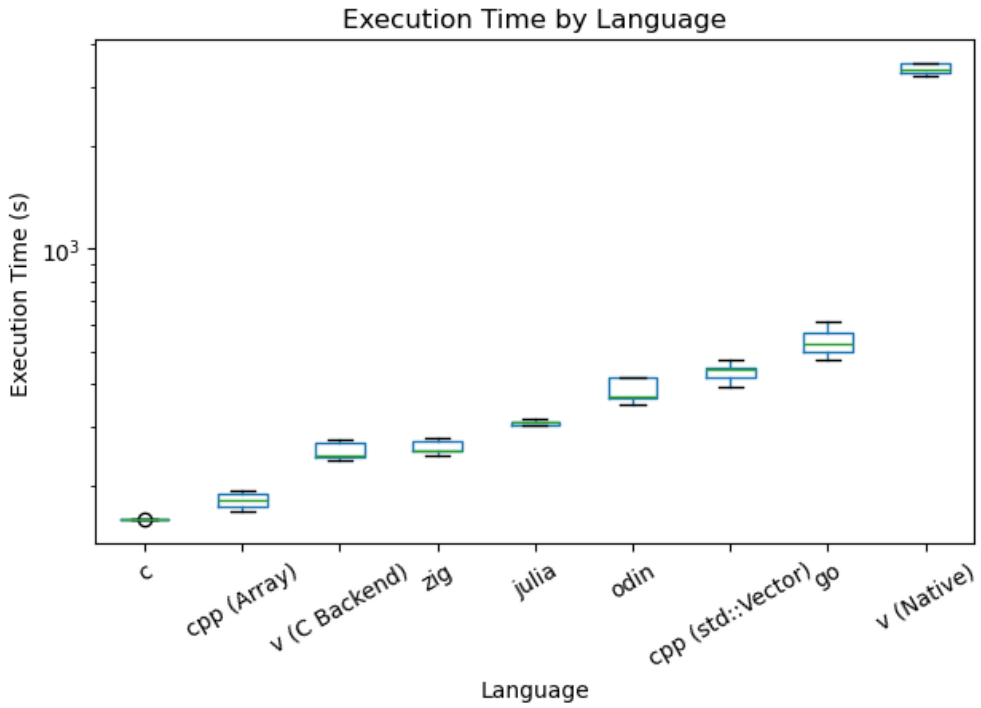

图 14：BenchGen 生成的不同编程语言基准测试的执行时间。

讨论。 图 16 总结了六个优化级别的结果： `-O0`、 `-Og`、 `-O1`、 `-Os`、 `-O2` 和 `-O3`。每组柱形图对应一个编译器版本： `gcc5`、 `gcc7`、 `gcc9`、 `gcc11`、 `gcc13` 和 `gcc14`。下面我们讨论本研究的一些见解：

编译时间：我们观察到，随着gcc的每个新版本发布，编译时间都稳步增加，尤其是在 `-O3`等较高优化级别下。比较第 5 版和第 14 版，九年间编译时间增加了约 42%，即使在 `-O0`下也增加了 31%。这并不意外：随着时间推移，新增了优化遍，虽然提升了代码质量，但也增加了编译时间。

执行时间：在较高优化级别下，性能有所提升。比较 `gcc5`和 `gcc14`，在 `-O3`下执行时间减少了约 27%，在 `-O1`下减少了 32%，在 `-Og`下最多减少了 47%。 `-O0` 下的收益较小，约为 7%。这些结果表明， `gcc`的新版本生成的代码效率更高，尤其是在中等优化级别下。

二进制文件大小：不同版本的二进制文件大小总体保持稳定。在 `-Os`下， `gcc5`至 `gcc14`之间的代码大小仅减少了 0.34%。在 `-O3`下，二进制文件大小从 `gcc5` 到 `gcc7`期间先有所增加，随后在 `gcc9`中明显下降。此后，我们再次观察到其小幅增加，直到 `gcc14`。不过，在此版本中，代码仍比 `gcc5`小近 10%。

从版本 5 演进到版本 14， `gcc` 展现出明显的权衡：以更长的编译时间换取效率更高的可执行文件。虽然二进制文件大小变化不大，但执行性能得到了显著提升，尤其是在 `-Og`和 `-O3`方面。这些结果凸显了一个重要趋势： `gcc` 社区始终将代码质量和运行时效率置于编译速度之上，接受更长的构建时间，以换取可量化的性能提升。

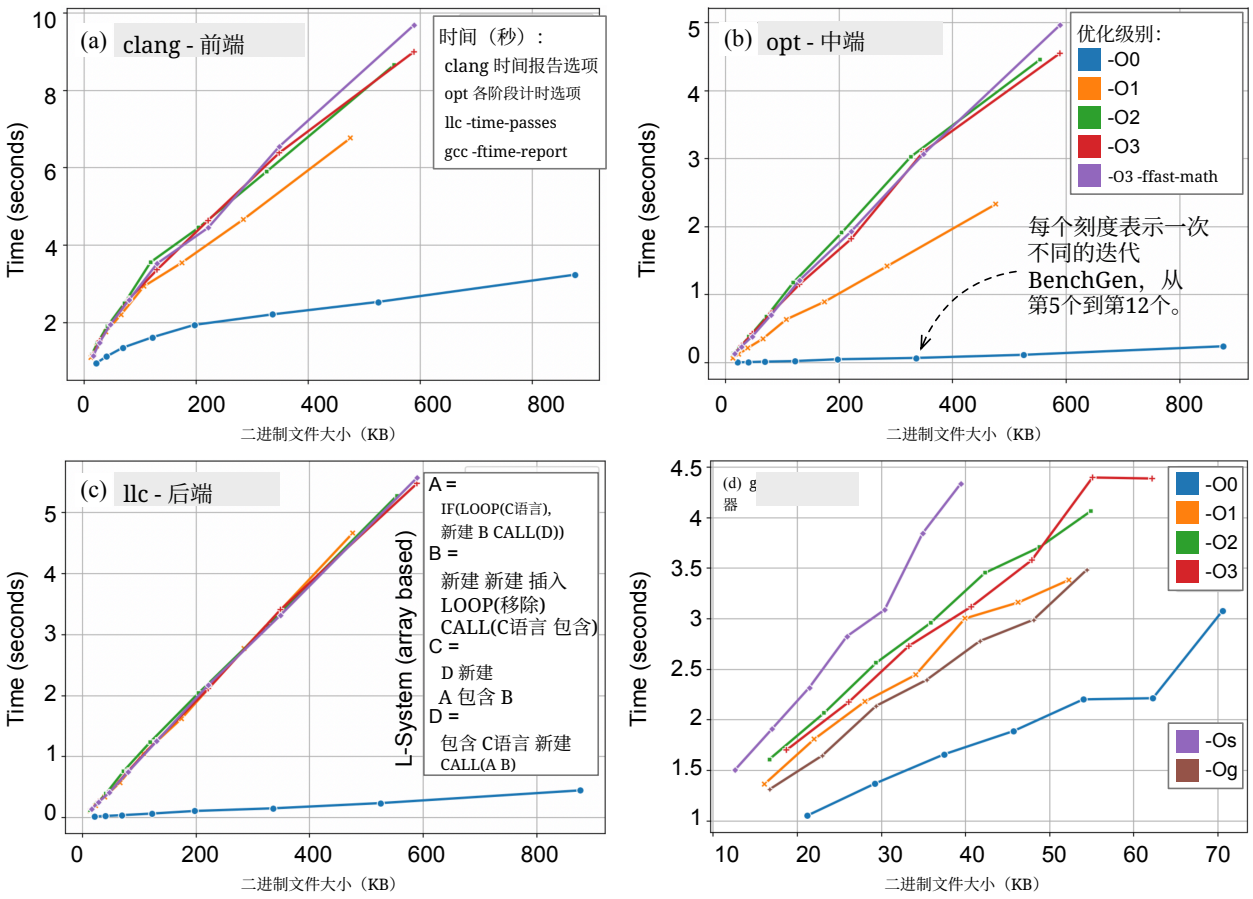

图 15：对 `clang` 的不同编译阶段以及 `gcc` 的完整编译流水线渐近行为的实证评估。

4.5 CS5： `clang` 中的配置文件引导优化

配置文件引导优化（PGO）是一种编译技术，它利用从程序代表性执行中收集的运行时信息来提升生成的代码质量。通过以实测执行数据取代纯静态启发式方法，PGO 使编译器能够在分支预测、函数内联、循环展开和代码布局方面做出更明智的决策，从而同时提升性能和效率。在典型的 C 语言、C++ 和 Rust 编译器中，PGO 被集成到标准优化流水线中，而不是作为独立的优化级别存在。具体而言，在 `clang`中，PGO 是对现有优化级别的改进，尤其是 `-O2` 和 `-O3`。通过 `-fprofile-use` 标志提供配置文件数据后，`clang` 会应用所选的基线优化，并结合配置文件驱动的洞察，生成更准确反映程序观测到的运行时行为的代码。

本节展示如何使用 BenchGen 评估配置文件引导优化（PGO）的效果。如第3.2节所述，BenchGen 生成的程序的控制流由变量 `PATH` 控制。为了评估 PGO 带来的性能提升，我们进行了以下实验：

1. 使用以下包含数组结构的语法生成四个程序变体（第6、7、8和9代）：

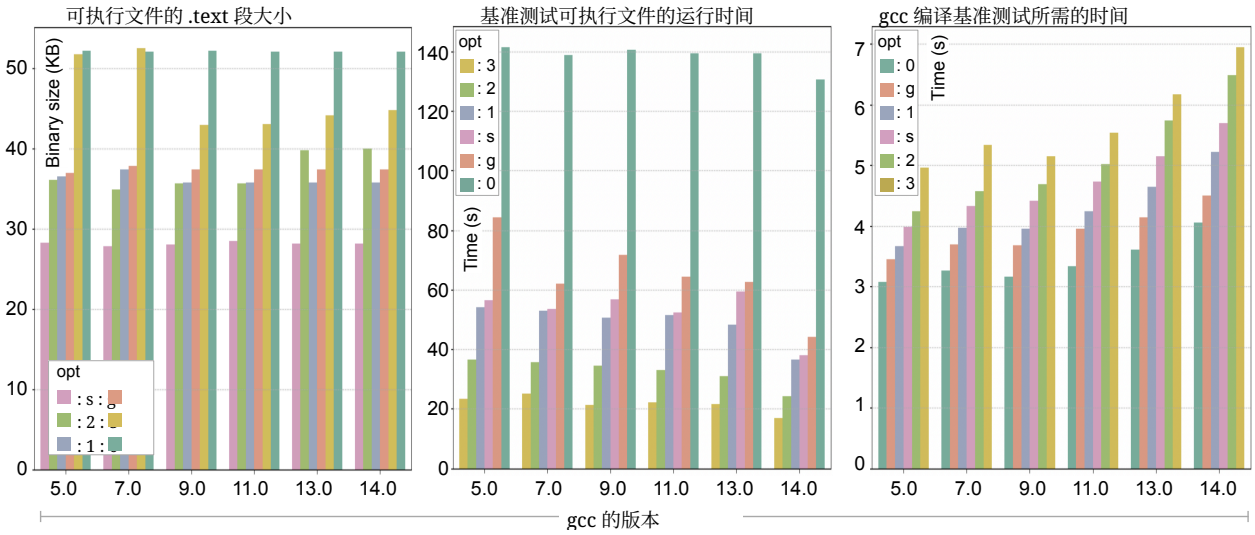

图 16：在六个优化级别下， `gcc` 从版本 5 到 14 的演变。结果分别展示了（左） `.text` 部分的二进制文件大小、（中）基准测试的执行时间，以及（右）编译时间。

```
A = new B B
B = IF(LOOP(insert A contains), LOOP(insert A contains))
```

2. 在启用 PGO 的情况下，以 `-O2` 优化级别编译每个程序[4]。

3. 在“训练阶段”期间，通过使用 `PATH` $=2^0$ （即 `PATH` $= 1$）运行每个程序来收集配置文件信息。

4. 在“测试阶段”期间，对 $i = 1$ `to` 63:

   (a) 使用 `PATH` $= 2^i - 1$ 执行每个程序，例如 `PATH` $= 0b1$、 `PATH` $= 0b11$ 等 c.

令 t表示使用 `clang -O2` 编译且未使用配置文件数据的程序执行时间，令 t_i 表示使用 `PATH` $= 2^i$时的执行时间。那么，比率 t/t_i 量化了控制流路径偏离 i位时，配置文件引导优化带来的相对加速比。

讨论。 图17展示了本实验收集的数据。图17中的结果揭示了配置文件引导优化的两种截然不同的表现。在符合预期的情况下（第6至第8代），性能趋势与我们的直觉相符：当测试期间的执行路径与训练期间经历的路径非常相似时，我们会观察到显著的加速比，通常可达 2 倍。先前的研究 [9, 36, 29]也报告了真实应用中类似的结果。

然而，随着 `PATH`中的位翻转数量增加，执行的控制流会偏离经过性能分析的运行，PGO 带来的收益也随之减小。最终，对于偏离程度很大的输入，经过性能分析和未经性能分析的二进制文件会呈现相近的性能，因为编译器基于性能分析的决策已不再适用。

性能分析过拟合： 这种意外情况出现在第9代。在这里，当执行遵循训练路径时，PGO 带来了非常显著的加速（超过 2.2 倍）。然而，随着路径逐渐偏离，

[4]我们在 `-O2` 级别报告 `clang` 的结果，因为使用配置文件引导优化（PGO）获得的加速比与在 `-O3` 观察到的相当，并且始终高于在 `-O1` 获得的加速比。

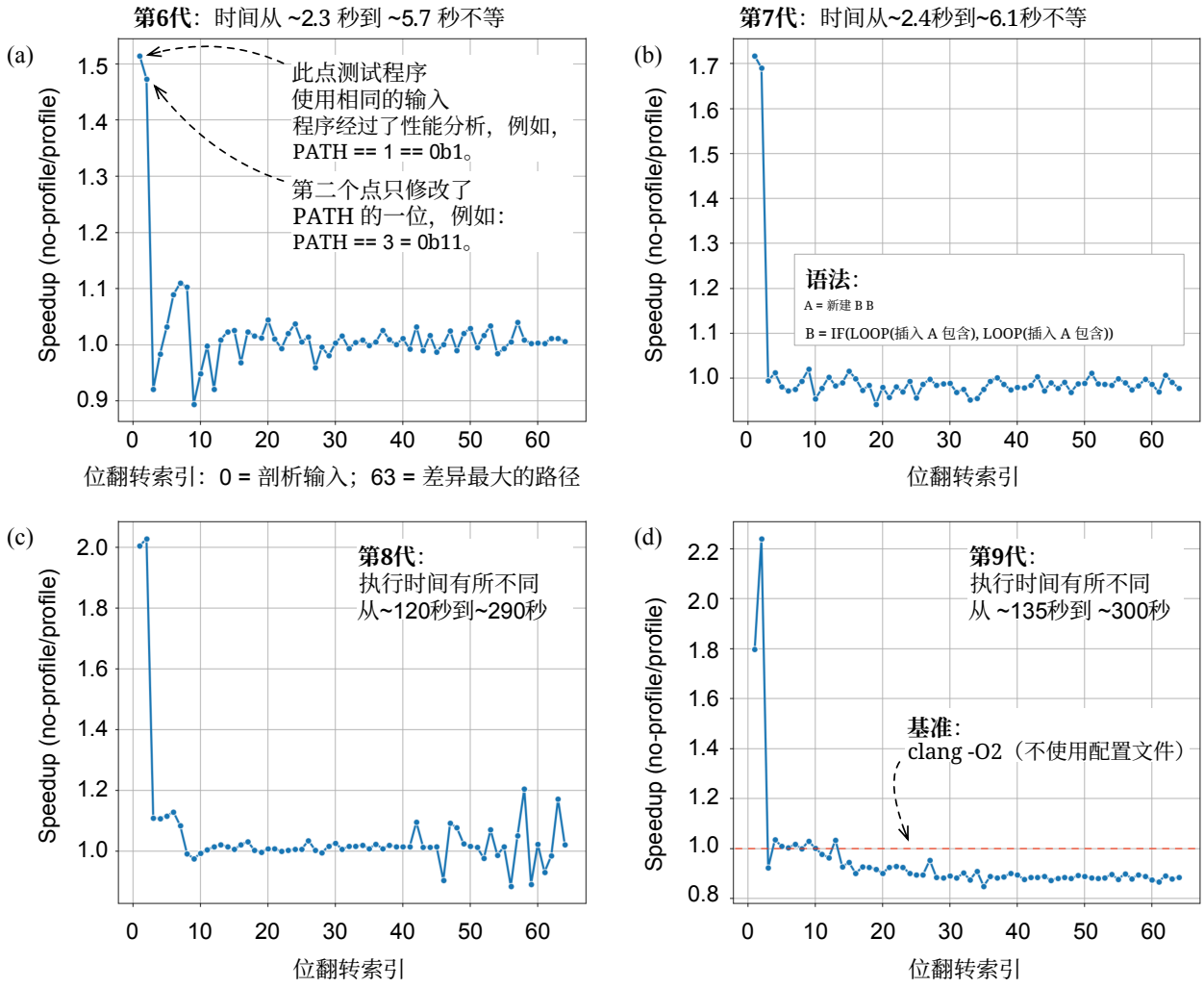

图17：配置文件引导优化对 `clang -O2` 的影响。

优化后的二进制文件比未经性能分析的对应版本更慢。这种现象表明出现了性能分析过拟合：由于 PGO 会将优化偏向观测到的热路径，未经过性能分析的路径上的代码可能会受到不利决策的影响。

通过读取 `perf` 计数器，我们观察到，随着训练路径与测试路径逐渐偏离，使用配置文件数据优化的程序比未使用配置文件数据优化的程序产生了更多缓存访问和缓存未命中。相比之下，随着路径偏离，分支未命中的比率并未表现出更大的差异。对此现象的一种合理解释是，使用和未使用配置文件数据时内联的函数数量不同。Clang/LLVM 在调用点级别做出内联决策。热路径上的函数调用可能会被积极内联，而冷路径上的同一调用则可能不内联，从而导致调用开销、代码大小和优化机会出现不一致。结合配置文件驱动的代码布局和分支预测，这种选择性优化可能使未经过配置文件优化的路径比中性的非 PGO 二进制文件运行得更慢。因此，第9代揭示了 PGO 的一个局限：当训练输入具有代表性时，PGO 能带来显著的性能提升；但当配置文件数据过度拟合少数执行路径时，性能可能会下降。

2

4.6 CS6：GLib 中的数据结构

GNOME 库（GLib）是一个 C 语言库，作为 GNOME 项目的一部分开发，提供了丰富的数据结构实现、实用函数和可移植性抽象。其通用容器包括动态数组（例如 GArray、GPtrArray）、字节数组（GByteArray）、单向链表（GSList）、双向链表（GList）、双端队列（GQueue）、哈希表（GHashTable）、平衡二叉树（GTree）等。之所以提供多种容器，是因为没有一种容器适用于所有场景：有些容器擅长快速插入或删除（尤其是在特定位置），另一些则能提供高效的随机访问、更好的内存局部性、有序遍历或更低的开销。通过提供多种数据结构，GLib 让程序员可以根据性能、内存和语义方面的需求，选择最合适的容器。

BenchGen 可对这些数据结构的动态行为进行系统评估。如第 3.1.2 节所述，当前的 C 代码生成器实现支持该库默认发行版提供的全部十二种容器[5]。本节将展示如何利用这一支持来比较不同容器的性能特征。

实验设置。 我们重点考察三类操作： `insert`、 `remove`和 `contains`。程序由一个 L 系统生成，该系统(i)快速填充容器，(ii)展开模板，并在模板中用随机操作序列填充占位符：

```
A0 = A1 A1 A1 A1 A1 A1 A1 A1;
A1 = A2 A2 A2 A2 A2 A2 A2 A2;
A2 = A3 A3 A3 A3 A3 A3 A3 A3;
A3 = insert insert;
B = LOOP(CALL(B) C);
C = B <operations> B;
```

上述初始化组件会产生 $8^3 \cdot 2 = 1024$ 次插入；必须以深度 ≥ 4 运行 BenchGen，才能在此次展开中生成操作。 `<operations>` 占位符会展开为由 `insert`、 `remove`和 `contains` 标记随机排列而成的序列；我们将每种标记的数量依次设为 $0, 2, 4, 6, 8, 10$，从而为每种容器生成 $6^3 = 216$ 个程序。

每个生成的程序都会编译两次：一次启用逐操作打印（以统计实际操作频率），另一次禁用打印（以测量不受 I/O 开销影响的运行时间）。使用 `hyperfine` 对二进制文件进行基准测试；由于总运行时间较长，每次测量只进行一次预热运行。我们会将某项操作次数相近的程序分组，并报告每组的平均运行时间。

讨论。 图 18 展示了平均运行时间随操作次数变化的情况。随着工作负载增大，哈希表和平衡树的表现优于链表、数组和队列。在后几种容器中，GList 的表现优于 GQueue；这是意料之中的，因为 GQueue 基于 GList 实现，因此还会产生额外的指针管理开销。

结果中一个值得注意的现象是，随着删除次数增加，平均执行时间反而下降。这并非正确性问题，而是分组方式造成的：删除操作较多的程序往往会使容器几乎变空，因此后续操作涉及的元素要少得多，导致每组的平均运行时间降低。

在图18所示的操作规模下，GTree 始终比 GHashTable 快。不过，当工作负载进一步增大时（我们针对更大的参数

[5]有关这些容器的详细介绍，请参阅 `https://docs.gtk.org/glib/data-structures.html`。

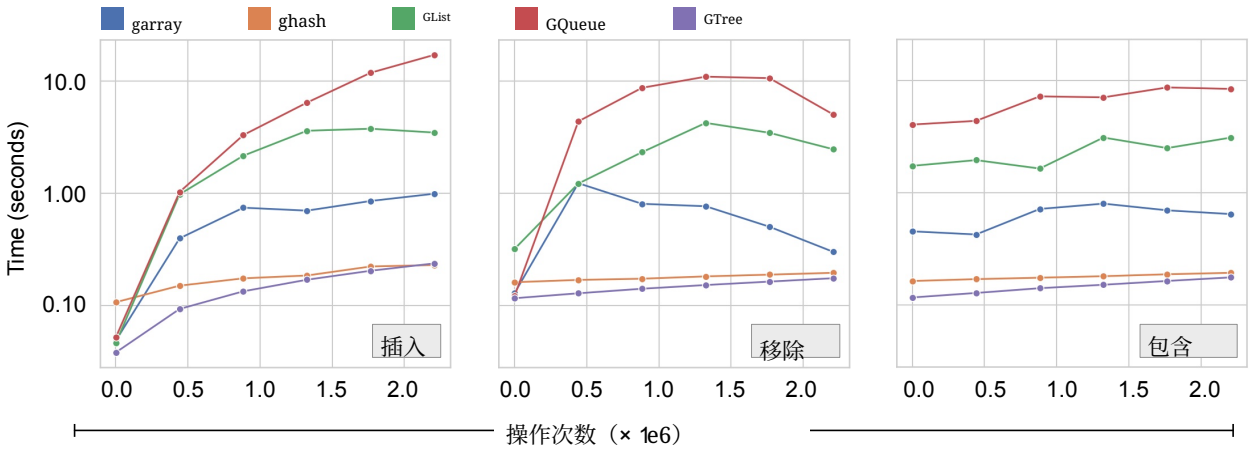

图 18：GLib 中五种数据结构的比较：GHashTable、GTree、GArray、GQueue 和 GList。

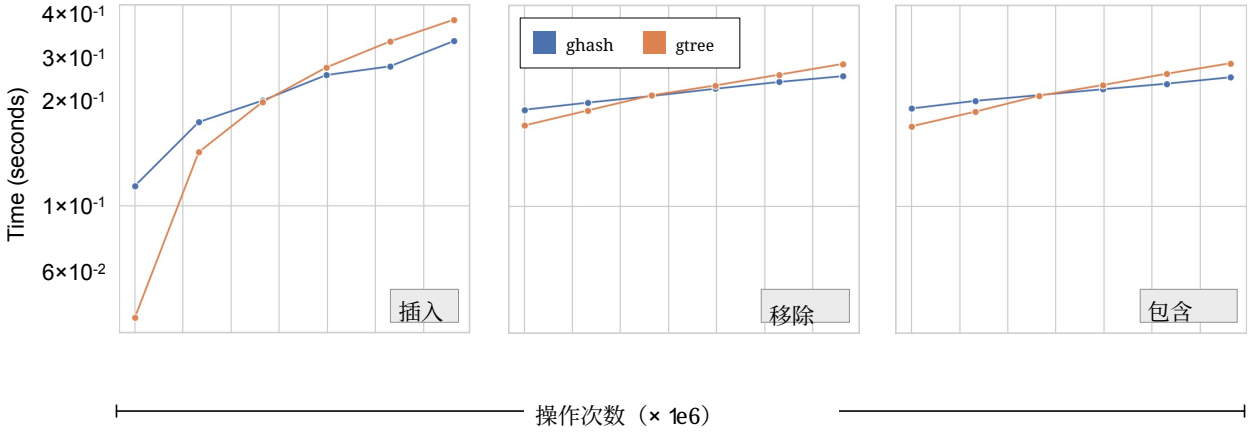

图 19：比　　较 GLib 中的平衡树（GTree）和哈希表（GHashTable）。

值重复了比较），GHashTable 最终会超越 GTree（见图 19）。这种交叉源于实现上的权衡：平衡树提供 $O(\log n)$ 的保证，而哈希表通常具有摊销 $O(1)$ 的表现，但哈希、桶管理和冲突处理会带来更高的常数开销。对于较小的工作负载，哈希表的常数开销占主导，因此 GTree 更有优势；对于较大的工作负载，不同的渐近增长率则使哈希表表现更佳。因此，观察到的性能交叉反映了这些容器实现中常数因子开销与渐近扩展之间的相互作用。

4.7 CS7：与 CSmith 的比较

多种编程语言模糊测试器广受欢迎。这类工具的主要目标，包括 CSmith [45] 和 **YarpGen** [27]（针对 C语言）或 ChiGen [42]（针对 Verilog），是发现编译器或解释器等语言处理系统中的缺陷。然而，BenchGen 的目的不同：它旨在评估这些系统的性能。它发现缺陷的能力有限，因为它生成的程序不如 CSmith 等模糊测试器生成的程序多样。另一方面，BenchGen 能够以 CSmith 及相关工具无法做到的方式，对编译器性能进行压力测试。本节将通过以下挑战说明这一点：

	-O0	-O1	-O2	-O3
BenchGen				
周期（平均值）	1.44E+11	5.21E+10	5.20E+10	5.14E+10
比率	1	2.76	2.77	2.80
SPEC C2017				
周期（平均值）	5.63E+11	2.43E+11	2.37E+11	2.30E+11
比率	1	2.32	2.37	2.44
CSmith				
周期（平均值）	2.85E+06	1.57E+06	1.49E+06	1.44E+06
比率	1	1.85	1.91	1.98

Tab表 1：不同优化级别下各基准测试的平均周期数和比率 s.

生成一组程序，使其执行时间与 SPEC CPU2017 程序的平均运行时间相当，并且在优化下的行为类似于真实工作负载。

讨论。 为了在这一挑战中比较 CSmith 和 BenchGen，我们使用这两个程序生成器生成了由四个基准测试组成的集合，并将其与 SPEC CPU2017 中的程序进行比较。所采用的方法如下：

1. 我们运行了 `clang`/LLVM 20.1 能够编译的 28 个 SPEC CPU2017 程序。2. 使用 CSmith，我们生成了 10,000 个程序，并选出其中执行时间最接近未进行优化编译的 28 个 SPEC CPU2017 程序平均运行时间的四个程序。3. 使用第 4.1 节中介绍的四种语法，我们生成了四个运行时间接近同一平均值的 BenchGen 程序。4. 随后，我们在不同的优化标志下对每组基准测试（分别为 28、4 和 4 个程序）进行了优化。

表 1 展示了本实验的结果。我们在获得运行时间超过 Spec 平均时间的程序后立即停止 BenchGen，并选择此前一代生成的程序。然而，无法以这种方式控制 CSmith。换言之，用户无法控制 CSmith，使其生成能在指定时间范围内执行的程序。大多数 CSmith 程序的运行时间都很短。此外，这些程序受到优化的影响，与我们在 Spec 集合中的真实基准测试中观察到的情况有很大差异；相比之下，BenchGen 程序所受的影响更接近真实工作负载的表现。以往的研究已注意到，编译器优化对 CSmith 程序与真实程序的影响存在这种差异 [13]。

5 相关工作

BenchGen 是一款旨在合成基准测试的工具。编译器文献中也有不少研究提出了基准测试生成技术。接下来，我们将回顾这

方面的研究，并着重介绍使 BenchGen 特别适用于性能分析的特点。

编译器模糊测试器。 编译器模糊测试工具与 BenchGen 的关键区别在于各自的主要目标。大多数现有工具旨在发现语言处理系统中的缺陷，而 BenchGen 则专门用于分析这些系统的性能。因此，在识别 C语言编译器中的错误方面，BenchGen 不如 CSmith 这样的模糊测试器有效。另一方面，它能够支持传统模糊测试器难以完成的任务，例如研究编译器优化的渐近行为、比较不同编译器的编译时间、代码大小和执行速度，或根据这些相同指标分析单个编译器在多个版本间的演变。

有几种工具能够根据上下文无关文法的产生式规则生成随机字符串 [22, 5, 42, 46]。事实上，语法引导模糊测试的核心思想可以追溯到 20 世纪 70 年代 [37]。然而，这种方法的一个众所周知的局限是，对于编程语言，通常难以保证生成的字符串对应于可编译的程序。尽管基于语法的模糊测试器能够确保语法正确性，但成功编译通常还取决于超出上下文无关文法表达能力的语义约束。

即使是专门用于生成可执行程序的模糊测试器，也可能无法生成可编译的代码。例如，Ricardo 等人 [38] 报告称，使用 YarpGen [27] 生成 C语言/C语言++程序时，编译成功率为 78.49%；而 Vieira 等人 报告称，使用 ChiGen 模糊测试器生成 Verilog 设计时，成功率约为 60 %。这一局限与我们开发 BenchGen 时的经验一致：大量工程工作都投入到确保每个生成的程序都能在没有警告的情况下通过编译，并且运行时不会出现异常。

基准测试合成器。 编译器的开发需要基准测试。因此，编程语言领域一些最具影响力的论文介绍了基准测试套件，例如 Spec CPU2006 [21], 、MiBench [18], 、Rodinia [8], 等。这些基准测试由人工整理，通常只包含少量程序。几年前，Cummins 等人 [10] 证明，这种较小的规模无法覆盖编译器在其生命周期中可能探索的程序特征空间。

过去十年来，为调优预测编译器而生成基准测试一直是一个活跃的研究领域。早期旨在开发预测优化器的工作使用了用于发现编译器缺陷的合成基准测试。此类合成器的例子包括 CSmith [45], 、LDRGen [2], 和 Orange3 [31, 32]。尽管这些工具最初是作为测试用例生成器设计的，但也被用于提升主流 C 编译器生成的优化后代码的质量 [20, 3]。然而，CompilerGym [11], 这一用于应用基于机器学习的代码优化技术的工具，也提供随机生成的 CSmith 程序。不过，近期研究表明，合成代码往往难以反映人类编写的程序的行为，因此会产生质量欠佳的训练集 [13, 16]。第 4.7 节将详细讨论 CSmith 的这些问题：难以用它生成运行时间合理的程序，以及编译器很容易对这类程序进行优化。

为生成更贴近现实的基准测试，近期涌现了大量从代码仓库中提取程序的研究，旨在构建用于训练编译器的基准测试套件。例如，其中一些工作旨在生成基准测试，以供机器学习模型使用 [15, 17, 35], 。与 BenchGen 相比，这些研究存在一些不足：

- 非常大的 be 基准测试套件，例如 AnghaBench [13] 或 Poj [30], ，可以编译但无法运行 .

- 旨在运行的基准测试套件有时会出现未定义行为，例如 ExeBench [1]和 ColaGen [6]。

- 使用大语言模型生成基准测试的技术，例如 Italiano 和 Cummins 的系统 [23], ，无法生成大型程序；正如 Guimarães等人 [39]最近指出的那样，即使只生成少量程序，也需要很长时间。

- Jotai 集合 [24], 由能够无未定义行为地运行的程序构成，其中仅包含非常小的程序。在我们尝试运行 CompilerGym [11], 中提供的程序时，运行时间最长的程序在使用 `clang -O0`编译后也仅运行了 0.017 秒。

6 结论

本文提出了一种生成基准测试的方法，其依据是程序通常呈现自相似结构，因而可以描述为 L 系统的推导过程。我们通过设计并实现 BenchGen 验证了这一理念。BenchGen 是一种能够生成多种编程语言基准测试的工具。与典型的模糊测试器不同，BenchGen 支持多语言，并允许用户生成任意大小的程序，其执行过程能够运行复杂的控制流模式。

未来工作。 除了 BenchGen 的具体实现之外，本文的核心贡献是提出可以将程序系统地生成为 L 系统的实例。本文探索的这种 L 系统变体采用了七种结构的组合（三种控制流结构和四种数据结构行为）。不过，这种方法可以扩展到范围广得多的语法特征。例如，当前版本的 BenchGen 一次只能生成使用单一数据结构的程序，而形式体系并未限制更丰富的数据结构组合。同样，也可以自然地纳入其他控制流结构，例如 `switch`、 `do-while`，甚至 `goto` 。我们将对此类扩展的研究留作未来颇具前景的方向。

数据可用性声明

BenchGen 可在 `https://github.com/lac-dcc/BenchGen` 获取，采用 Apache-2.0 许可证。

致谢

本项目获得了 Google 和 FAPEMIG（资助编号 APQ-00440-23）的支持。我们感谢 Xinliang (David) Li 和 Victor Lee 努力促成 Google 的赞助。我们还感谢 Lucas Victor Silva 整理表 1 中的数据，以及 Bruno Pena Baêta 和 Matheus Alcântara 为 BenchGen 早期版本所做的工作。

引用

[1] Jordi Armengol-Estapé、Jackson Woodruff、Alexander Brauckmann、José Wesley de Souza Magalhães 和 Michael F. P. O'Boyle。《ExeBench：一个机器学习规模的可执行 C 函数数据集》。载于 *MAPS*，第 50–59 页，美国纽约州纽约市，2022年。计算机协会。

[2] Gergö Barany。《活性驱动的随机程序生成》。载于*LOPSTR*，第 112–127 页，德国海德堡，2017年。Springer。[3] Gergö Barany。《通过差分测试发现遗漏的编译器优化》。载于第*27*届国际编译器构造会议论文集，CC 2018，第 82–92 页，美国纽约州纽约市，2018年。ACM。[4] Patrick Baudin、François Bobot、David Bühler、Loïc Correnson、Florent Kirchner、Nikolai Kosmatov、André Maroneze、Valentin Perrelle、Virgile Prevosto、Julien Signoles 和 Nicky Williams。《为实现无缺陷 C 程序而不懈努力：Frama-C 软件分析平台》。《*ACM*通讯》，64(8):56–68，2021年7月。[5] Bachir Bendrissou、Cristian Cadar 和 Alastair F. Donaldson。《用于测试输入解析器的语法变异》。《*ACM*软件工程与方法论汇刊》，34(4)，2025年4月。[6] Maksim Berezov、Corinne Ancourt、Justyna Zawalska 和 Maryna Savchenko。《COLA-Gen：用于基准测试自动代码生成的主动学习技术》。载于 Francesca Palumbo、João Bispo 和 Stefano Cherubin 编，*PARMA-DITAM*，《信息学开放获取系列（OASIcs）》第 100 卷，第 3:1–3:14 页，德国达格施图尔，2022年。Schloss Dagstuhl – Leibniz信息学中心。[7] Jeff Bezanson、Jiahao Chen、Benjamin Chung、Stefan Karpinski、Viral B. Shah、Jan Vitek 和 Lionel Zoubritzky。《Julia：通过设计协调动态性与性能》。《*ACM*程序设计语言学报》，2(OOPSLA)，2018年10月。[8] Shuai Che、Michael Boyer、Jiayuan Meng、David Tarjan、Jeremy W. Sheaffer、Sang-Ha Lee 和 Kevin Skadron。《Rodinia：异构计算基准测试套件》。载于*IISWC*，第 44–54 页，美国华盛顿特区，2009年。IEEE。[9] Dehao Chen、David Xinliang Li 和 Tipp Moseley。《AutoFDO：面向仓库级应用的自动反馈导向优化》。载于*CGO*，第 12–23 页，美国纽约州纽约市，2016年。计算机协会。[10] Chris Cummins、Pavlos Petoumenos、Zheng Wang 和 Hugh Leather。《合成用于预测建模的基准测试》。载于*CGO*，第 86–99 页，美国新泽西州皮斯卡塔韦，2017年。IEEE。[11] Chris Cummins、Bram Wasti、Jiadong Guo、Brandon Cui、Jason Ansel、Sahir Gomez、Somya Jain、Jia Liu、Olivier Teytaud、Benoit Steiner 等。《CompilerGym：稳健、高性能的人工智能研究编译器优化环境》。载于*CGO*，第 92–105 页，美国纽约，2022年。IEEE。[12] Charlie Curtsinger 和 Emery D. Berger。《Coz：通过因果分析找到影响性能的代码》。载于*SOSP*，第 184–197 页，美国纽约州纽约市，2015年。计算机协会。[13] Anderson Faustino da Silva、Bruno Conde Kind、José Wesley de Souza Magalhães、Jerônimo Nunes Rocha、Breno Campos Ferreira Guimarães 和 Fernando Magno Quintão Pereira。《AnghaBench：一个包含一百万个可编译 C 基准测试、用于缩减代码大小的套件》。载于*CGO*，第 378–390 页，美国加利福尼亚州洛斯阿拉米托斯，2021年。IEEE。[14] Chucky Ellison 和 Grigore Rosu。《具有应用的 C 语言可执行形式语义》。《*SIGPLAN*通知》，47(1):533–544，2012年1月。

[15] Jaroslav Fowkes 和 Charles Sutton. 跨 GitHub 的无参数概率 API 挖掘. 载于*FSE*，第 254–265 页，美国纽约州纽约市，2016年。计算机协会。[16] Andrés Goens、Alexander Brauckmann、Sebastian Ertel、Chris Cummins、Hugh Leather 和 Jeronimo Castrillon. 基于机器学习合成基准测试的案例研究. 载于第 *3* 届 *ACM SIGPLAN*机器学习与编程语言国际研讨会论文集，MAPL 2019，第 38–46 页，美国纽约州纽约市，2019年。ACM。[17] Georgios Gousios 和 Diomidis Spinellis. 从 GitHub 挖掘软件工程数据. 载于*ICSE-C*，第 501–502 页，美国华盛顿特区，2017年。IEEE出版社。[18] M. R. Guthaus、J. S. Ringenberg、D. Ernst、T. M. Austin、T. Mudge 和 R. B. Brown. MiBench：一个免费且具有商业代表性的嵌入式基准测试套件. 载于*WWC*，第 3–14 页，美国华盛顿特区，2001年。IEEE。[19] Xue Han 和 Tingting Yu. 高度可配置软件系统中性能缺陷的实证研究. 载于 *ESEM*，美国纽约州纽约市，2016年。计算机协会。[20] Atsushi Hashimoto 和 Nagisa Ishiura. 通过随机生成等价程序发现 C 编译器的算术优化机会. *IPSJ*系统*LSI*设计方法学汇刊，9：21–29，2016年。[21] John L. Henning. SPEC CPU2006 基准测试描述. *SIGARCH Comput. Archit. News*，34(4)：1–17，2006年9月。[22] Renáta Hodován、Ákos Kiss 和 Tibor Gyimóthy. Grammarinator：一种基于语法的开源模糊测试工具. 载于第 *9* 届 *ACM SIGSOFT*测试用例设计、选择与评估自动化国际研讨会论文集，A-TEST 2018，第 45–48 页，美国纽约州纽约市，2018年。计算机协会。[23] Davide Italiano 和 Chris Cummins. 使用大语言模型发现编译器中遗漏的代码尺寸优化 . 载于编译器构造国际会议，第 81–91 页，美国纽约州纽约市，2025年。计算机协会。[24]Cecilia Conde Kind、Michael Canesche 和 Fernando Magno Quintao Pereira. Jotai：一种生成可执行 C 基准测试的方法. 技术报告，技术报告 02-2022。米纳斯吉拉斯联邦大学，2022年。[25] Kota Kitaura 和 Nagisa Ishiura. 基于静态与动态代码混合比较的编译器性能随机测试. 载于第 *9* 届 *ACM SIGSOFT*测试用例设计、选择与评估自动化国际研讨会论文集，A-TEST 2018，第 38–44 页，美国纽约州纽约市，2018年。计算机协会。[26] Aristid Lindenmayer. 发育过程中细胞相互作用的数学模型 I：具有单侧输入的丝状体.《理论生物学杂志》，18(3)：280–299，1968年。[27] Vsevolod Livinskii、Dmitry Babokin 和 John Regehr. 使用 YARPGen 对 C 和 C++编译器进行随机测试.《*ACM*程序设计语言学报》，4(OOPSLA)，2020年11月。[28] Valentin J.M. Manes、HyungSeok Han、Choongwoo Han、Sang Kil Cha、Manuel Egele、Edward J. Schwartz 和 Maverick Woo. 模糊测试的艺术、科学与工程：综述.《*IEEE*软件工程汇刊》，47(11)：2312–2331，2021年。

[29] Angélica Aparecida Moreira、Guilherme Ottoni 和 Fernando Magno Quintão Pereira. Vespa：用于二进制优化的静态性能剖析.《*ACM*程序设计语言学报》，5(OOPSLA)，2021年10月。[30] Lili Mou、Ge Li、Lu Zhang、Tao Wang 和 Zhi Jin. 面向编程语言处理的树结构卷积神经网络. 载于*AAAI*，第 1287–1293 页，帕洛阿尔托，美国加利福尼亚州，2016年。AAAI Press。[31] Eriko Nagai、Atsushi Hashimoto 和 Nagisa Ishiura. 通过扩大表达式的大小和数量，强化 C 编译器算术优化的随机测试. *IPSJ*《系统*LSI*设计方法学汇刊》，7：91–100，2014年。[32] Kazuhiro Nakamura 和 Nagisa Ishiura. 基于期望值计算的 C 编译器随机测试中引入循环语句，2015年。[33] Nicholas Nethercote 和 Julian Seward. Valgrind：一种重量级动态二进制插桩框架. 载于*PLDI*，第 89–100 页，美国纽约州纽约市，2007年。计算机协会。[34] Oswaldo Olivo、Isil Dillig 和 Calvin Lin. 静态检测集合遍历中的渐近性能缺陷. 载于*PLDI*，第 369–378 页，美国纽约州纽约市，2015年。计算机协会。[35] David N Palacio、Alejandro Velasco、Daniel Rodriguez-Cardenas、Kevin Moran 和 Denys Poshyvanyk. 使用句法结构评估并解释大语言模型对代码的处理，2023年。[36] Maksim Panchenko、Rafael Auler、Bill Nell 和 Guilherme Ottoni. Bolt：一种面向数据中心及其他场景的实用二进制优化器. 载于*CGO*，第 2–14 页。IEEE出版社，2019年。[37] Paul Purdom. 用于测试解析器的句子生成器. *BIT*，12(3)：366–375，1972年9月。[38] Gabriel Ricardo、Natanael Santos Junior、Flavio Figueiredo 和 Fernando Pereira. 基于大语言模型的编译器模糊测试的实用性. 载于*SBLP*，第 59–66 页，阿雷格里港，RS，巴西，2025年。SBC。[39] Gabriel Ricardo、Natanael Santos Junior、Flavio Figueiredo 和 Fernando Pereira. 基于大语言模型的编译器模糊测试的实用性. 载于*SBLP*，第 59–66 页，阿雷格里港，RS，巴西，2025年。SBC。[40] Konstantin Serebryany、Derek Bruening、Alexander Potapenko 和 Dmitry Vyukov. AddressSanitizer：一种快速的地址有效性检查器. 载于*USENIX ATC*，第 28 页，美国，2012年。USENIX协会。[41] Bob Strecansky. *Go* 高性能实战：借助弹性能力，大规模提升并优化 *Golang* 应用程序性能。Packt Publishing Ltd，2020年。[42] João Victor Amorim Vieira、Luiza de Melo Gomes、Rafael Sumitani、Raissa Maciel、Augusto Mafra、Mirlaine Crepalde 和 Fernando Magno Quintão Pereira. 自底向上生成用于测试电子设计自动化工具的 Verilog 设计，2025年。[43] Zheng Wang 和 Michael F. P. O'Boyle. 编译器优化中的机器学习.《*IEEE*论文集》，106(11)：1879–1901，2018年。

[44] Chunqiu Steven Xia、Matteo Paltenghi、Jia Le Tian、Michael Pradel 和 Lingming Zhang。Fuzz4all：基于大语言模型的通用模糊测试。载于*IEEE/ACM 第46*届国际软件工程会议论文集，ICSE '24，美国纽约州纽约市，2024年。计算机协会。[45] Xuejun Yang、Yang Chen、Eric Eide 和 John Regehr。在 C 编译器中查找并理解缺陷。载于第*32*届 *ACM SIGPLAN* 编程语言设计与实现会议论文集，PLDI '11，第283–294页，美国纽约州纽约市，2011年。计算机协会。[46] José Antonio Zamudio Amaya。在基于语法的模糊测试中塑造测试输入。载于*ISSTA*，第1901–1905页，美国纽约州纽约市，2024年。计算机协会。