Multi-Language Benchmark Generation via L-Systems

VINICIUS FRANCISCO DA SILVA, UFMG, Brazil
HEITOR LEITE, UFMG, Brazil
FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

L-systems are a mathematical formalism proposed by biologist Aristid Lindenmayer with the aim of simulating organic structures such as trees, snowflakes, flowers, and other branching phenomena. They are implemented as a formal language that defines how patterns can be iteratively rewritten. This paper describes how such a formalism can be used to create artificial programs written in programming languages such as C, C++, Julia and Go. These programs, being large and complex, can be used to test the performance of compilers, operating systems, and computer architectures. This paper demonstrates the usefulness of these benchmarks through multiple case studies. These case studies include a comparison between `clang` and `gcc`; a comparison between C, C++, Julia and Go; a study of the historical evolution of `gcc` in terms of code quality; a look into the effects of profile guided optimizations in `gcc`; an analysis of the asymptotic behavior of the different phases of `clang`'s compilation pipeline; and a comparison between the many data structures available in the Gnome Library (GLib). These case studies demonstrate the benefits of the L-System approach to create benchmarks, when compared with fuzzers such as CSmith, which were designed to uncover bugs in compilers, rather than evaluating their performance.

CCS Concepts: • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: L-System, Benchmark, Synthesis

1 Introduction

Language processing systems, such as compilers, interpreters and static analyzers are complex tools whose validation depends on programs written in the target language. However, the number of available benchmarks for any given compiler is typically limited [39]. To address this limitation, several tools have been developed to automatically generate test programs [41]. This process, known as *fuzzing* [26], is widely used to uncover bugs such as crashes and memory leaks. In the compiler domain, fuzzers such as Csmith [41] and YARPGen [25] generate random C programs to stress-test compilers and support static analysis. More recently, fuzzers like Fuzz4All [40] have leveraged Large Language Models (LLMs) to generate test programs not only for compilers but also for constraint solvers, interpreters, and other software systems with accessible APIs.

Despite the abundance of program generators [41], existing tools exhibit important limitations. A key issue is the lack of control over the size of the generated programs. For example, Csmith, the most widely used C compiler fuzzer, does not allow users to tune the output size. Instead, it produces programs whose sizes follow a normal distribution. When compiled with `clang` v9.0.1 at `-O0`, these programs contain on average 20,190 LLVM instructions, with a standard deviation of 3,650 and a median of 19,161 instructions [12]. Other fuzzers, such as YARPGen [25], LDRGen [2], and Orange3 [23], show similar behavior. As a consequence, these tools are ill-suited for performance evaluation of compiler components such as parsing, semantic analysis, and code generation.

Programs via L-Systems. To overcome this limitation, this paper proposes a methodology for stress-testing compilers with a focus on *performance* rather than solely on *correctness*. Our approach enables the controlled generation of programs whose size can be precisely tuned by the user. This makes it possible to construct synthetic code of virtually arbitrary size, constrained only by practical resources such as generation time and storage space.

Authors' Contact Information: Vinicius Francisco da Silva, UFMG, Belo Horizonte, Minas Gerais, Brazil, silva.vinicius@dcc.ufmg.br; Heitor Leite, UFMG, Belo Horizonte, Minas Gerais, Brazil, heitor.leite@dcc.ufmg.br; Fernando Magno Quintão Pereira, UFMG, Belo Horizonte, Minas Gerais, Brazil, fernando@dcc.ufmg.br.

The central insight is that programs often exhibit recursive, self-similar structures, as discussed in Section 2. For instance, the branches of a control construct (e.g., `if-then-else`) are themselves programs that may recursively contain further control constructs. To exploit this property, we introduce a generation technique based on *L-systems* [24]. Originally developed by Aristid Lindenmayer to model the growth of biological organisms, L-systems provide a grammar-based framework that we extend to code generation. This paper shows how families of self-similar programs can be encoded as L-grammars. Programs are then generated by iterative rewriting, where each resulting string encodes the blueprint of a program organized around a central data structure such as an array or a list. As discussed in Section 3, this technique supports multiple programming languages, yields complex control-flow graphs, avoids undefined behavior, and enables the manipulation of different data structures.

Summary of Contributions. To demonstrate the benefits of representing program growth with L-systems, we introduce BenchGen, a multi-language benchmark generator. Section 4 evaluates BenchGen through seven case studies, including: a comparison between `gcc` and `clang`; a comparison of Go, Julia, C, and C++; an asymptotic analysis of different components of `clang`; a longitudinal analysis of the evolution of `gcc`; an evaluation of profile-guided optimizations in `clang`; a comparison of data structures from GLib; and a comparison between programs generated by BenchGen and by Csmith. BenchGen is publicly available under the GPL 3.0 license, and has already been used to generate benchmarks for several languages beyond those discussed in this paper, including Rust and Vale.

2 L-Systems

An L-system (or *Lindenmayer system*) is a formal model based on rewriting rules, originally devised to describe the growth patterns of plants and other fractal-like structures. It comprises an alphabet of symbols, a set of production rules that define how symbols are transformed, and an initial *string* (the axiom) that serves as the starting point. At each iteration, the rules are recursively applied to the current string, producing increasingly complex sequences. Example 2.1 illustrates how this formalism operates.

Example 2.1. Figure 1 presents an example of an L-system. The rules used in this system generate geometric patterns through string rewriting. Starting from the axiom A, the productions specify how symbols evolve at each step: $A \rightarrow B - A - B$ and $B \rightarrow A + B + A$. Here, the symbols $-$ and $+$ represent rotations of 60 and 300 degrees, respectively. Repeated application of these rules generates sequences that, when interpreted graphically, produce intricate fractal curves, such as the well-known Sierpinski Triangle.

2.1 Programs as Self-Similar Structures

L-systems exhibit a property known as *self-similarity*, meaning that structures contain smaller copies of themselves across different scales. In the context of L-systems, this feature arises naturally from the recursive application of rewriting rules, producing patterns that preserve the same shape at progressively finer levels of detail. This behavior is evident in fractals like the Sierpinski Triangle seen in Example 2.1, where each component is a scaled-down replica of the whole. Such hierarchical repetition is key to modeling phenomena like plant growth, coastlines, and tree branching.

Self-similarity also emerges in computer programs, which often embody recursive and hierarchical structures. Many programs are composed of smaller functions that may invoke themselves or be embedded within one another, as in `if-then-else` blocks or loops. Modularity and code

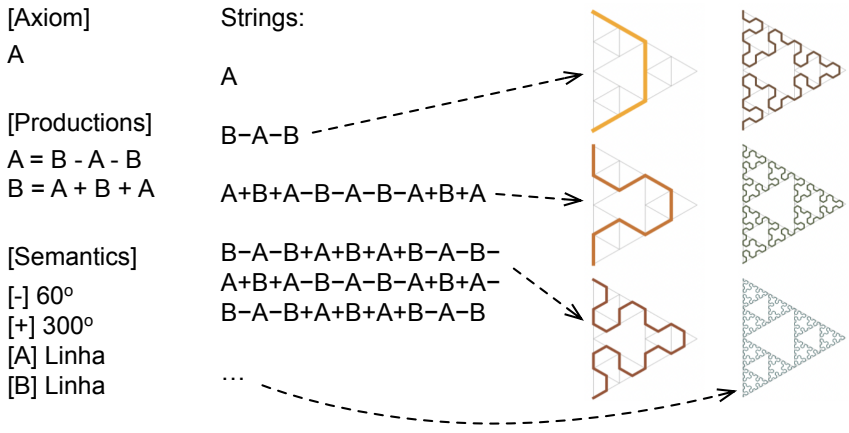

Fig. 1. L-system describing the Sierpinski Triangle.

reuse further reinforce this pattern: generic routines can be instantiated repeatedly across different abstraction levels, as Example 2.2 explains.

Example 2.2. Figure 2 illustrates the concept of self-similarity in code using a nested `if-then-else` block. Initially, a function `g(x)` contains a single conditional. However, it can be recursively expanded to `g(x) = if g(x) then g(x) else g(x)`, forming a self-referential structure. Such recursive definitions naturally lead to self-similarity and are common in syntactical constructs that encode control-flow in programs.

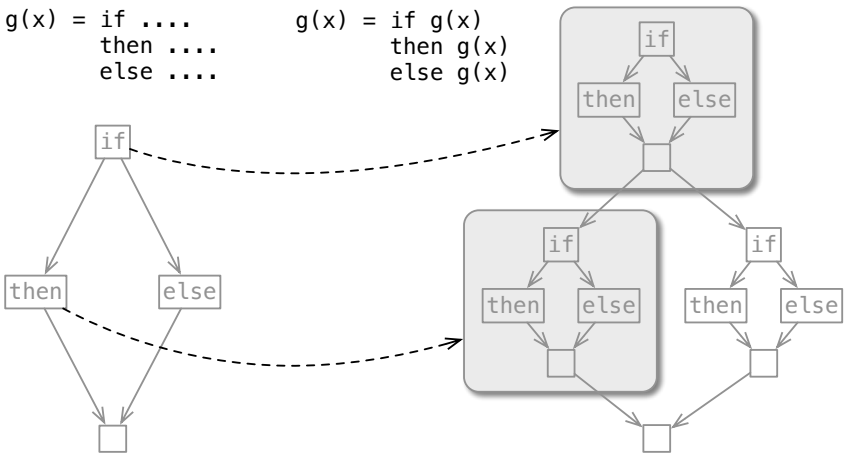

Fig. 2. The self-similar nature of computer code.

Summary of Ideas This paper leverages the principle of self-similarity to generate C programs that are both well-defined and arbitrarily complex. The generation model is based on an L-grammar, akin to the one illustrated in Example 2.1, but instead of producing geometric patterns, it synthesizes C code constructions with executable semantics. Figure 3 shows an example of two versions of a program that BenchGen produces for C and Julia. This file is part of a 52-file benchmark that we use in Section 4.2 to compare the performance of different language processing systems.

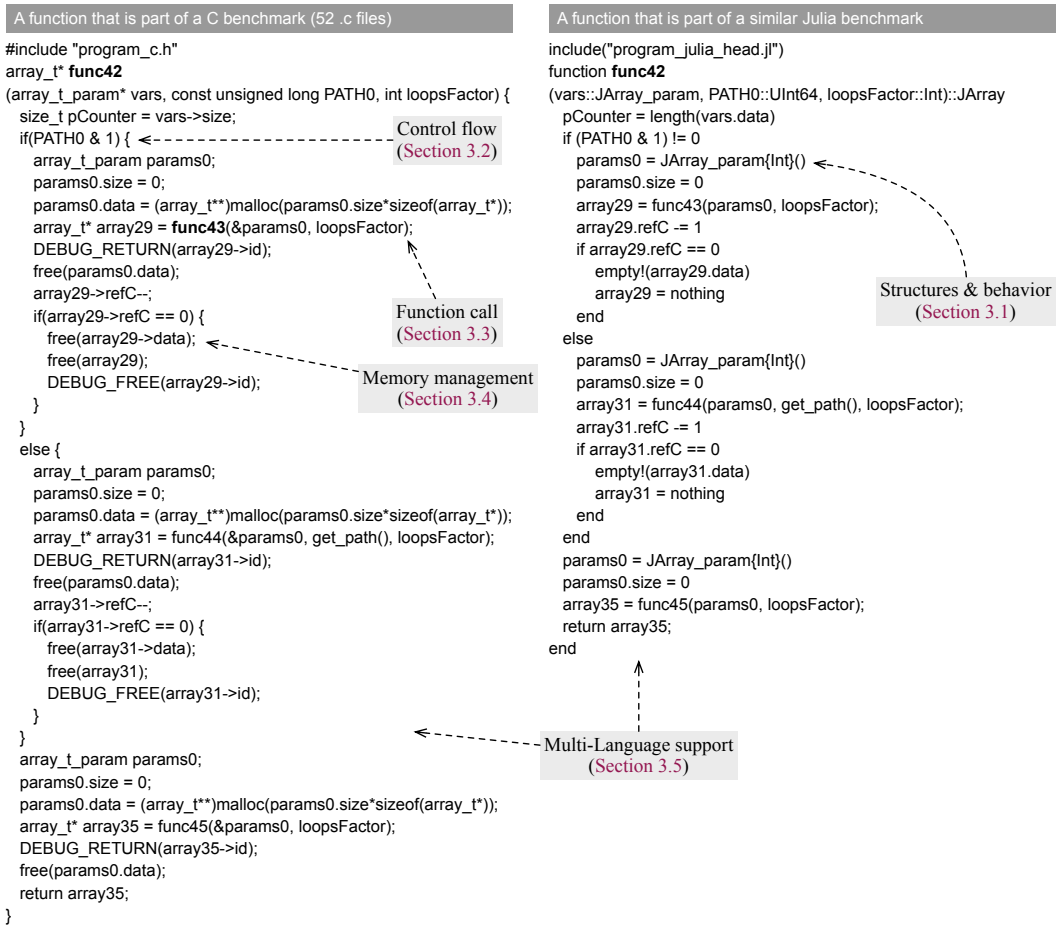

Fig. 3. Example of code that BenchGen produces for C and Julia that manipulate arrays.

3 Code Generation via L-Systems

The tool BenchGen, developed in this work, generates programs that manipulate data structures, according to the schema seen in Figure 4. Section 3.1 introduces the core building blocks used in program construction, while Section 3.2 describes semantics of these building blocks.

3.1 Syntactical Building Blocks

The L-systems described in this paper are built from two families of constructs:

- **Structure:** elements that define the control flow of a program, including `IF`, `LOOP`, and `CALL`.
- **Behavior:** operations that specify how data is manipulated, including `new`, `insert`, `remove`, and `contains`.

3.1.1 Structure Blocks. The control flow in programs generated by BenchGen arises from combining four types of code blocks, as specified by the grammar below:

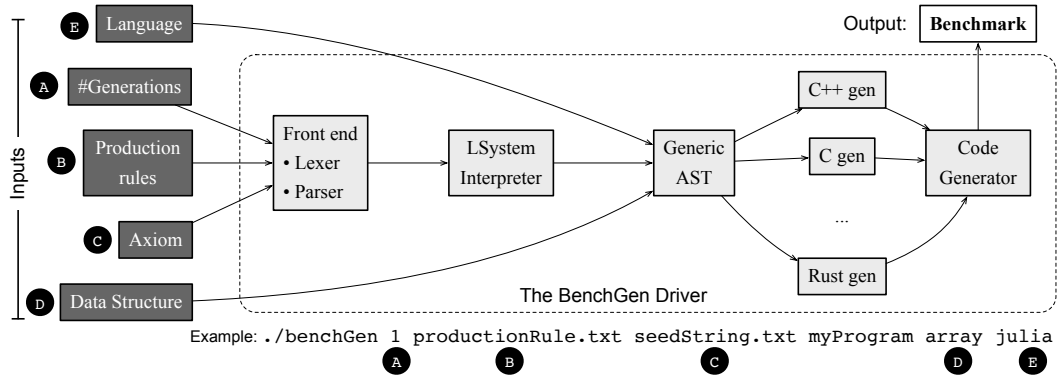

Fig. 4. BenchGen is a tool that generates programs in different programming languages. It receives as input the description of an L-System, which consists of a set of production rules, plus a starting symbol (the axiom). It interprets these rules, producing strings (the L-Strings), which guide the construction of genertic ASTs. These trees can be converted to programs in different programming languages using different containers. A code generator then converts the language-specific tree to a program.

$$
\begin{array}{rcll}
b & ::= & \mathrm{IF}(b_{\mathrm{cond}}, b_{\mathrm{then}}) & ;; \textit{if\_then} \\
 & | & \mathrm{IF}(b_{\mathrm{cond}}, b_{\mathrm{then}}, b_{\mathrm{else}}) & ;; \textit{if\_then\_else} \\
 & | & \mathrm{LOOP}(b_{\mathrm{cond}}, b_{\mathrm{body}}) & ;; \textit{while} \\
 & | & \mathrm{CALL}(b) & ;; \textit{function\_call}
\end{array}
$$

Each of these constructs corresponds to a familiar programming construct: conditional branches (`if-then`, `if-then-else`), loops (`while`), and function calls. Example 3.1 provides an illustration.

Example 3.1. Figure 5 shows a grammar designed to synthesize programs. The right-hand side illustrates two derivation steps from this L-grammar, along with a corresponding (simplified) C program produced from the second derivation.

3.1.2 Behavior Blocks. The dynamic behavior of BenchGen programs emerges from interactions with data structures. All data structures within a program must share the same type. Currently, BenchGen supports arrays of integers and sorted linked lists of integers. Additionally, the C code generator also supports any of the twelve data structures available in the GNOME Library. Adding support for new containers involves extending four C++ classes. These classes define the behavior of four constructs in the L-grammar:

- **new:** Creates and initializes a data structure.
- **insert:** Adds an element to a data structure in scope.
- **remove:** Deletes an element from a data structure in scope.
- **contains:** Checks whether an element exists in a data structure in scope.

Notice that these four behavior blocks can be customized by BenchGen users in any way. For instance, BenchGen provides a "*scalar mode*" to generate programs that manipulate integer variables. In this case, we have configured `insert` to increment variables; `remove` to decrement them; `new` to declare and initialize them with zero; and `contains` to test if they are zero. As a more concrete example, below we show how three such operations may be defined for programs working with arrays.

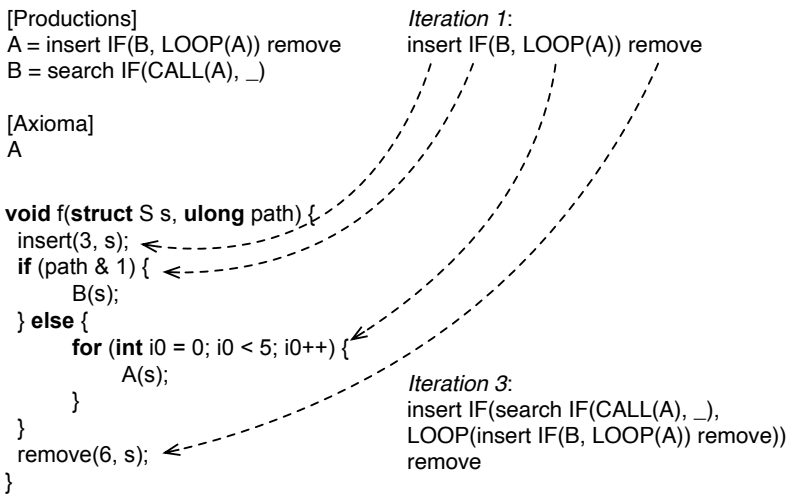

Fig. 5. Example of an L-grammar used to define programs. The figure shows a simplified version of a C program. Figure 3 provides a more concrete view.

Example 3.2. Figure 6 shows C code generated to manipulate arrays. Although a program may handle many arrays, the example focuses on a specific variable, `array156`. The operations `new()`, `insert()`, and `contains()` must be customized by the user to define the behavior of the generated code.

3.2 Execution Flow

Programs generated by BenchGen are executable. Their control flow is governed by a variable named `PATH`, of type `unsigned long`, which determines the outcome of conditional branches. Algorithm BitPath in Figure 1 correlates the outcome of branches with this `PATH` variable. The algorithm use a counter stack to ensure each path is uniquely identifiable via the `PATH` variable. It assigns a unique bitmask to each branch condition by:

(1) **Tracking Nesting Depth**: Using a stack to manage counters for each level of nested branches.
(2) **Resolving Joins**: Propagating the "maximum counter" upward when branches merge, ensuring no bitmask collisions.

The way Algorithm 1 determines how control-flow is taken lets us use BenchGen to test the effectiveness of profile-guided optimizatins, as Section 4.5 will demonstrate. This mechanism is clarified in Example 3.3.

Example 3.3. Figure 7 depicts the control flow graph of a program with the `PATH` variable combined with masks created by the Algorithm 1. Notice that the counter used as the bitmask is updated due to nesting or sequencing. Nesting causes the increment of the counter on the top of the counter stack. Whenever we analyze a branch, its counter will be the maximum of all the possible control-flows that reache it.

3.3 Function Calls

BenchGen supports function calls through the `CALL` clause. When an L-string includes a construction of the form `CALL`(e), the entire substring e is extracted and defined as a separate function.

new

```
array_t* array156;
array156 = (array_t*)malloc(sizeof(array_t));
array156->size = 42;
array156->refC = 1;
array156->id = 156;
array156->data = (unsigned int*)malloc(array156->size*sizeof(unsigned int));
memset(array156->data, 0, array156->size*sizeof(unsigned int));
DEBUG_NEW(array157->id);
```

insert

```
unsigned int loop46 = 0;
unsigned int loopLimit46 = (rand()%loopsFactor)/2 + 1;
for(; loop46 < loopLimit46; loop46++) {
  for (int i = 0; i < array156->size; i++) {
    array156->data[i]--;
  }
}
```

constains

```
unsigned int loop46 = 0;
unsigned int loopLimit46 = (rand()%loopsFactor)/2 + 1;
for (int i = 0; i < array156->size; i++) {
  if (array156->data[i] == 61) {
    break;
  }
}
```

Fig. 6. Examples of block definitions for `new`, `insert`, and `contains`.

Algorithm 1: BitPath assignment of unique bitmasks to branch conditions

Input: A function f
Output: A mapping of each branch condition in f to a unique bitmask
Initialize the Stack: push a single counter $Counter_0 = 1$, representing the least significant bit (LSB);

Assign Masks to Branches:;
- **Same Nesting Level**: use the current top-of-stack counter to generate the bitmask, e.g., `PATH & (1 « (counter - 1));`
- **Nested Branch**: push a new counter, incremented by 1 from the parent counter, ensuring fresh bits for deeper nesting;

Handle Join Points (after `if-then-else`):;
- Pop the current counter from the stack;
- Update the parent counter (now on top of the stack) to the maximum of:;
  - (a) the parent's original value;
  - (b) the popped (child) counter;
- This ensures that parent branches account for the deepest nesting level beneath them;

Independent Branches: after resolving a join, subsequent branches at the same level reuse the updated parent counter;

The functions generated from a given L-specification can be either grouped into a single file or

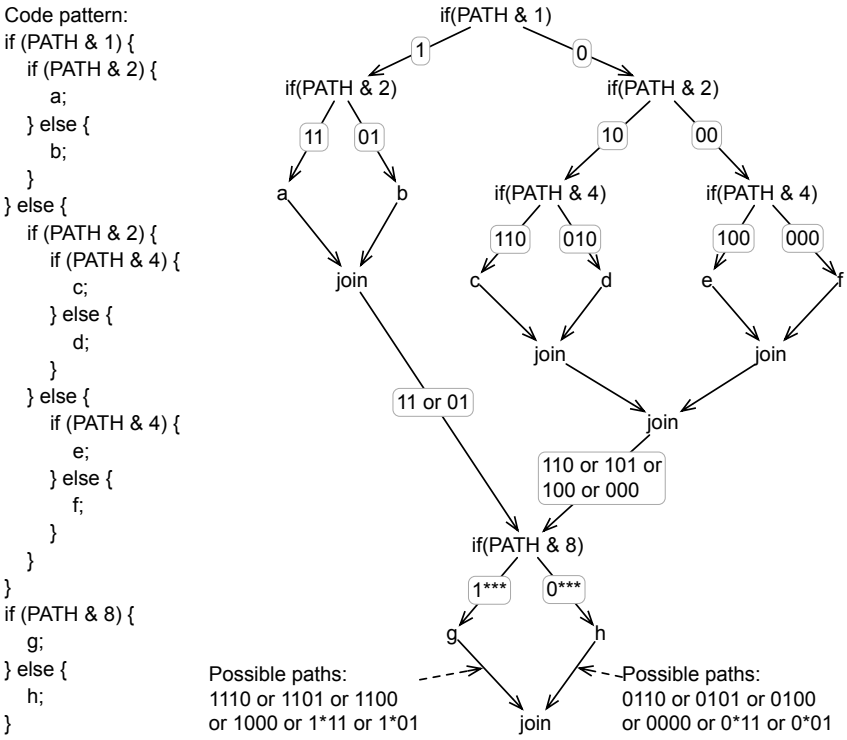

Fig. 7. The control flow of a synthetic program is determined by the `path` parameter.

distributed across multiple files, depending on a configuration parameter in BenchGen. Example 3.4 illustrates how BenchGen generates interprocedural code.

Example 3.4. Figure 8 depicts the control flow produced by a `CALL` block. The enclosed string gives rise to a new function, which becomes part of the synthesized program. This new function is invoked at the point in the L-string where the `CALL` clause appears.

All occurrences of `CALL`(b) with identical strings b result in calls to the same function. To avoid redundancy, BenchGen maintains a table mapping strings to functions, ensuring that identical strings refer to the same function instance. Example 3.5 shows how the same function can be called multiple times. Notice that BenchGen does not support the creation of recursive function calls. If a function `f0` calls another function `f1`, then `f0` is produced by a string that is strictly larger than the string that generates `f1`.

Example 3.5. Figure 9 shows an L-System that generates a program with two functions, `f0` and `f1`. The latter is called twice, because it is produced by a string that appears two times in a given iteration of the expansion process.

Parameter Passing. Functions generated by BenchGen receive two parameters:

- **Data:** an array of data structures that enables sharing between caller and callee functions.
- **Path:** a control variable that governs execution flow, as described in Example 3.3.

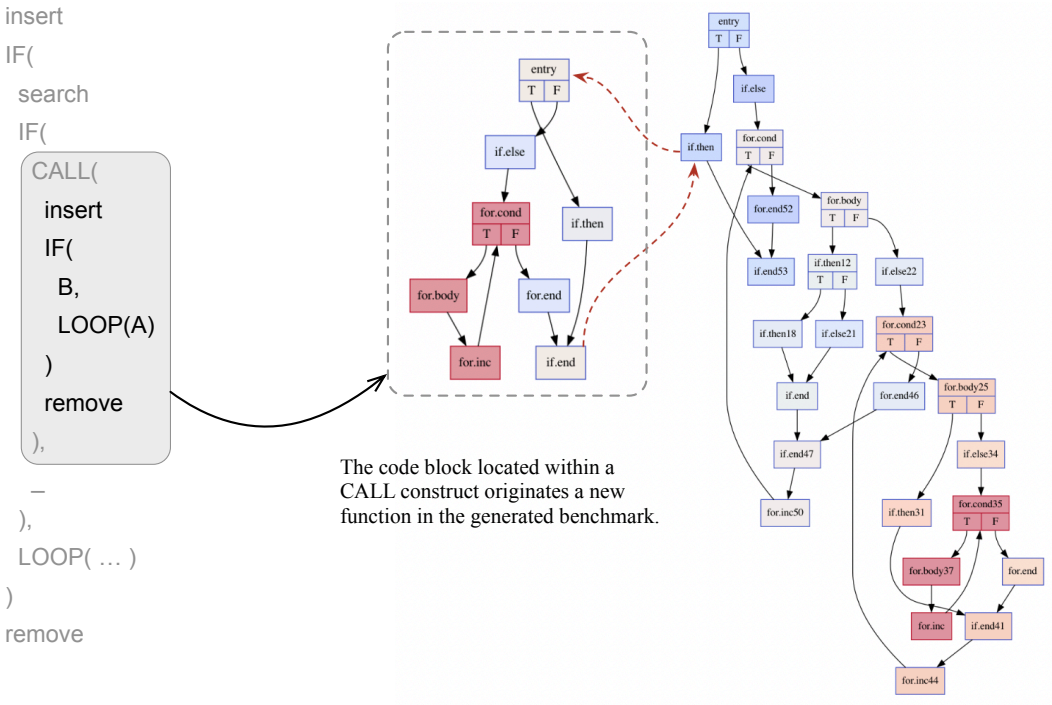

Fig. 8. Control flow of a program containing a CALL block.

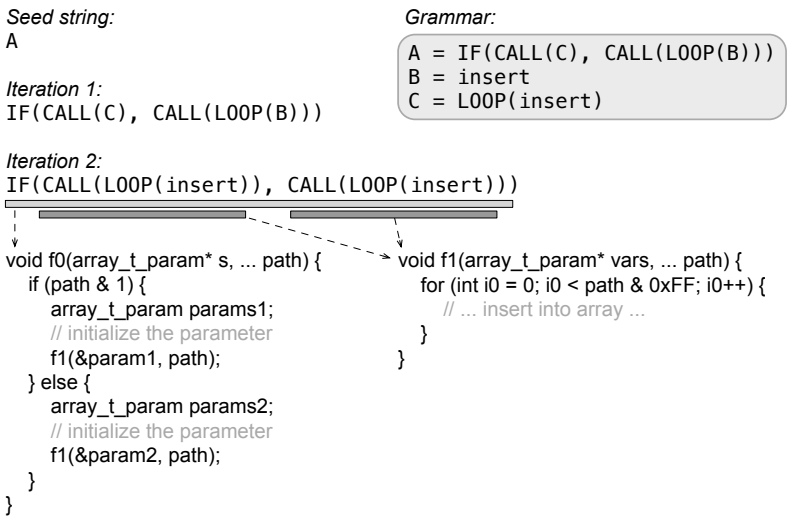

Fig. 9. An L-System that would generate multiple calls to the same function.

The **Data** array is populated using a reaching definitions analysis, which determines which variables in the caller function are available to be passed as arguments at the call site. Example 3.6 illustrates this mechanism.

Example 3.6. Figure 10 illustrates parameter passing in a program generated by BenchGen. At the function call site, a reaching definitions analysis identifies three available variables: `array1`, `array156`, and `array157`. Pointers to these variables are inserted into the `param.data` structure, which is passed to function `func0`. Inside the callee, the passed variables are copied into new ones using the `new` construct. Each variable is copied exactly once. If no additional parameters are available for copying, subsequent `new` operations will create fresh variables, as seen in Example 3.3.

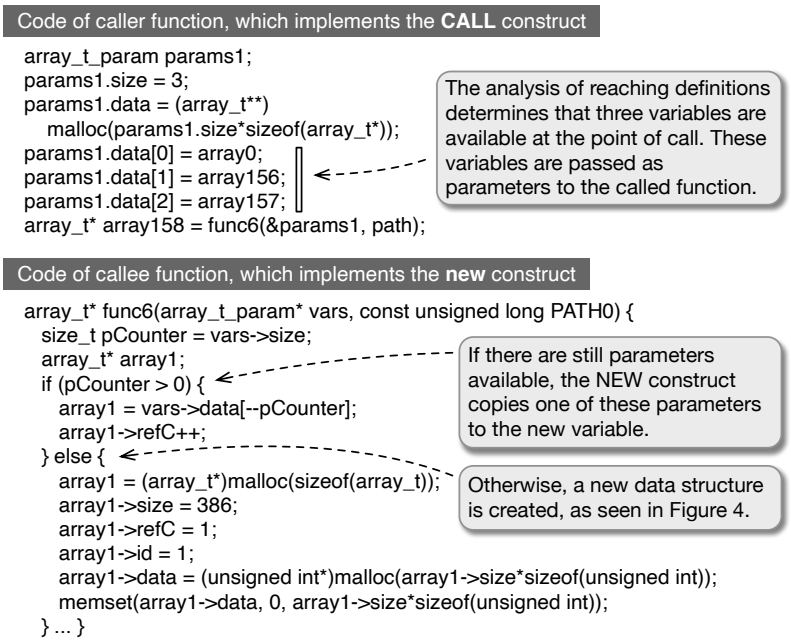

Fig. 10. Parameter passing in programs created by BenchGen.

3.4 Memory Management

Programs generated by BenchGen do not suffer from memory leaks, despite frequently relying on heap allocation. To prevent leaks, BenchGen employs a reference-counting garbage collector. This approach tracks how many references (pointers) exist to each dynamically allocated object. When a reference is created, the count is incremented; when it is removed, the count is decremented. Once the count reaches zero, the object is no longer reachable and can be safely deallocated. To support this mechanism, each structure created by BenchGen includes an additional field, `refC`, which stores the current reference count. Example 3.7 shows how reference counting is used in BenchGen to prevent memory leaks from happening.

Example 3.7. Figure 11 illustrates how BenchGen uses reference counting to manage memory. In this example, two variables, `x` and `y`, initially point to two separate heap-allocated structures. Each of these structures contains a `refC` field that holds the number of active references to the object. When the assignment `x := y` occurs, the reference count of the structure originally pointed to by `x` is decremented, as `x` no longer refers to it. If this decrement causes `refC` to reach zero, the structure is automatically deallocated. Meanwhile, the reference count of the structure pointed to by `y` is incremented to account for the new reference from `x`. After the assignment, both `x` and `y`

point to the same object, whose `refC` now reflects two active references. This mechanism ensures that heap-allocated memory is reclaimed as soon as it is no longer reachable, preventing memory leaks in programs generated by BenchGen.

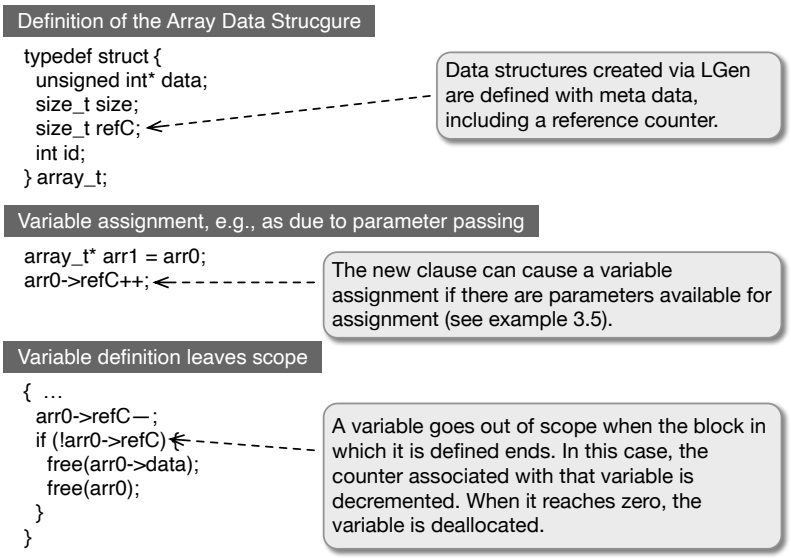

Fig. 11. Reference counter implementation.

3.5 Multilanguage Support

BenchGen is currently available in a beta version that supports the creation of benchmarks for multiple programming languages. Some of these languages are evaluated in Section 4.2. The benchmark generator itself is implemented in C++. To add support for a new language, users must implement new C++ classes that define the semantics of the structure and behavior blocks described in Section 3.1. This process also requires modifying the templates that generate code for the structure blocks (`IF`, `LOOP`, and `CALL`) and the behavior blocks (`insert`, `remove`, `contains`, and `new`).

Extending BenchGen therefore involves recompilation and is not yet fully automated. In addition to implementing new C++ classes, users must register the language module in the main BenchGen driver, which requires changing one line of code in the driver itself. In future work, we aim to fully automate this process and decouple the tool from the specific languages it supports. To this end, we are considering extending BenchGen with a domain-specific language that would allow users to specify new code-generation directives declaratively.

4 Experimental Evaluation

The goal of this section is to demonstrate how BenchGen can be used in practice. To this end, we explore its usage in the following *Case Studies*:

- **CS1**: Comparison between `gcc` and `clang` in terms of execution speed, binary size, and compilation time.
- **CS2**: Comparison of the C and C++ compiler backends available in the Gnu Compiler Collection.

- **CS3**: Analysis of the asymptotic behavior of different optimization levels in `clang` and `gcc`.
- **CS4**: Examination of the evolution of `gcc` from version 5 to 14, focusing on execution speed, binary size, and compilation time.
- **CS5**: Evaluation of the impact of profile-guided optimizations in `clang`.
- **CS6**: Comparison of different data structures in the Gnome Library (GLib) with respect to insertion, search, and deletion times.
- **CS7**: Comparison between BenchGen and CSmith, a compiler fuzzer.

Experimental Setup: All experiments were conducted on an Intel(R) Xeon(R) CPU E5-2680 v2 running at 2.80 GHz, with Linux Ubuntu 5.15.0-139-generic. The specific compiler and library versions used are detailed in each case study.

Checking for Undefined Behavior: We analyzed the C programs evaluated in Section 4.1 using four tools capable of detecting undefined behavior: UBSan and ASan from `clang` [36], Valgrind [31], the EVA plugin of Frama-C [4], and KCC [13]. In addition, we verified that every program evaluated in Section 4.2, when executed in debug mode, produces the same trace[1]. While these measures do not formally prove that BenchGen always generates sound benchmarks, they provide strong evidence that this is highly probable.

4.1 CS1: `gcc` vs `clang`

The GNU C Compiler (`gcc`) and LLVM's `clang` are the two most widely used C compilers, and comparisons between them are common in the literature. This case study contributes one more data point to this ongoing discussion by evaluating both compilers across three performance metrics:

- Execution time of the compiled program.
- Compilation time required to build the program.
- Binary size of the compiled program.

The comparison covers six optimization levels for `gcc` 14.2: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, and `-Ofast`; and seven for `clang` 21.0: `-O0`, `-O1`, `-O2`, `-O3`, `-Oz`, `-Os`, and `-Ofast`. Binary sizes were collected using the Linux `size` command, considering only the `text` section, which represents the size of the executable code in bytes. Compilation and execution times were measured with `hyperfine`, configured with three warm-up runs and at least ten benchmark runs. To compare the compilers, we have used four different L-Systems to generate four programs. We chose generations that ensured a mix of different execution times; hence, this custom benchmark collection contains programs that run within 10, 50 and 200 seconds once compiled with `gcc -O3`.

Discussion. Figure 12 summarizes the results of this comparison, showing, under each plot, the L-System and the generation that produced those numbers. The scatter plots show compilation time against execution time, with point size proportional to binary size. We observe that `gcc` and `clang` tend to produce programs with similar execution times at the lowest and highest optimization levels; however, `gcc`'s compilation times are generally lower.

A clear trend emerges for `gcc`: it often exhibits a smooth trade-off between compilation time and execution time, forming a Pareto-like curve where longer compilation typically results in faster execution. This pattern is less evident for `clang`. Instead, `clang` shows a marked distinction between the lowest optimization levels (`-O0`, `-O1`) and the higher ones, while the most aggressive optimizations (`-O2`, `-O3`, and `-Ofast`) yield very similar results. This observation echoes findings by Curtsinger and Berger in their work on Coz [11], who reported no statistically significant difference between `-O2` and `-O3` in `clang`[2]. Regarding binary size, both compilers generate smaller executables

[1] BenchGen provides a debug mode in which each program logs every behavioral operation defined in Section 3.1.2.
[2] See https://youtu.be/r-TLSBdHe1A?t=1438 at 23:58

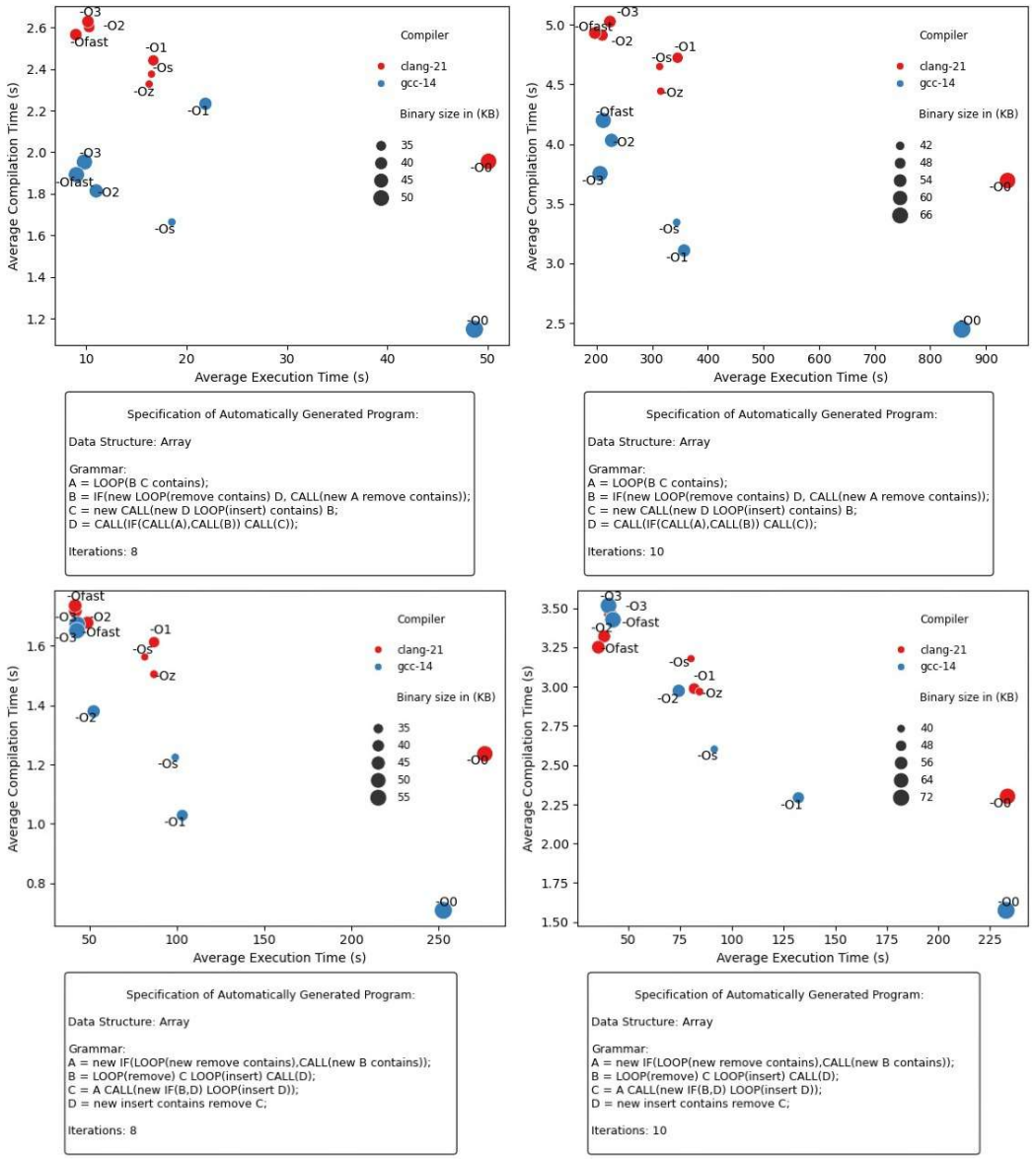

Fig. 12. Comparison of gcc 14.2 and clang 21.0 across optimization levels.

at -Os, with clang holding a slight edge (2–3%) at the -Oz level. Nevertheless, both can reduce code size by nearly 50% when moving from -O3 to -Os.

4.2 CS2: Comparison between Different Programming Languages

As explained in Section 3.5, BenchGen can be customized to synthesize benchmarks in multiple programming languages. This section leverages its multi-language support to compare C, C++, Julia, and Go. Support for these languages is currently available in BenchGen's official repository;

however, the tool has also been extended to generate benchmarks in other languages, including Rust and Vale. These extensions, however, are not maintained by the BenchGen authors. To perform this comparison, we used the first grammar shown in Figure 12 to produce eleven generations of a program (reporting data for generations 6 through 11). Programs in each language were executed with the following tools:

C: `gcc` 13.3.0
C++: `g++` 13.3.0
Go: `go` 1.25.0
Julia: `julia` 1.11.6

Note that Julia runs in interpreted mode with just-in-time compilation, whereas the other languages are compiled ahead of time. This evaluation considers only execution time, excluding compilation time. Also, each language produces the same number of source files, except for C and C++, which generate one extra header file. For example, in the 11th generation, BenchGen produces 52 files for Julia and Go, and 53 for C and C++.

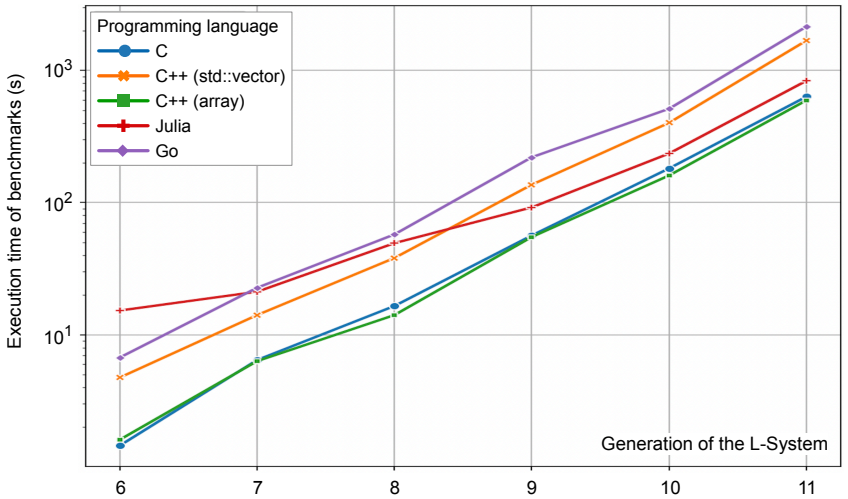

Fig. 13. Execution time of benchmarks produced by BenchGen in different programming languages.

Discussion. Figure 13 compares execution time across languages. The results largely align with expectations: they highlight the performance differences between low-level, statically compiled languages (C and C++) and higher-level languages with different compilation models (Go and Julia). All languages exhibit an exponential increase in execution time as program depth grows, which is inherent to the fractal-based benchmark generator. Beyond this general trend, we note several specific findings:

- **C vs. C++ with arrays.** Performance is nearly identical: there is no statistically significant difference at the 95% confidence level when comparing programs of the same generation. This is because both are compiled with the same compiler infrastructure (`gcc` 13.3.0), which produces highly similar assembly code.
- **Julia.** Julia shows excellent performance, particularly for larger programs—a behavior already observed in previous work [6]. Its just-in-time (JIT) compiler, built on LLVM, can apply aggressive runtime optimizations such as loop unrolling and SIMD vectorization by

specializing code based on runtime types and execution patterns. As program complexity increases in later generations, these optimizations give Julia a significant advantage, allowing it to outperform C++ with vectors.

- **C++ with vectors.** Programs using `std::vector` are slower than those using raw arrays. This overhead arises from additional metadata (size, capacity) and the cost of reallocation when capacity is exceeded, which requires copying elements to new memory. In benchmarks where data grows quickly, this reallocation cost is significant. Indeed, from the 8th to the 9th generation, Julia's JIT optimizations allow it to surpass C++ with vectors in performance.
- **Go.** Go is consistently the slowest language. This is unsurprising, and follows trends already reported and discussed in open forums and technical books [37]. While Go is compiled, its runtime includes garbage collection and applies less aggressive optimizations, which introduces overhead not present in C, C++.

The trends observed in Figure 13 are consistent with those documented on the well-known "Benchmark Game" website[3]. A key distinction, however, lies in the implementation strategy. While the Benchmark Game compares hand-optimized solutions that may differ syntactically and semantically, BenchGen produces structurally similar code across languages. This methodology offers a more precise comparison of compiler performance by eliminating variability introduced by differing algorithmic implementations.

4.3 CS3: Asymptotic Behavior

The ability to gradually vary the size of programs generated by BenchGen makes it a suitable tool for conducting empirical asymptotic analyses of language processing systems. In this section, we illustrate this capability by empirically evaluating the asymptotic complexity of different phases of `clang` (front end, middle end, and back end), as well as the full compilation pipeline of `gcc`. For this purpose, we employ a standard L-System to produce eight generations of a program (from the 5th to the 12th) and feed these programs to the compilers.

Discussion. Figure 14 relates the running time of the different compilation phases to the size of the binary processed at each generation of the target program. This analysis yields several insights, discussed below:

- All analyzed tools exhibit linear behavior for large programs, with very strong correlations (above 0.9) across different compilers and optimization levels. This applies to both `clang` and `gcc`. However, for `clang`'s front end, linear growth only becomes evident in later generations (iterations 9–12), due to the heavy startup costs of the parser and IR generator.
- There is little statistical difference among the higher optimization levels of `clang` (`-O2`, `-O3`, and `-Ofast`). This observation corroborates the findings reported in Section 4.1. It appears valid, at least in this experiment, across all three phases of `clang`'s compilation pipeline, and is especially pronounced in `llc`, the machine code generator.
- Although `gcc` also shows strong linear behavior at all optimization levels, the constant factors vary considerably. The size-optimization level (`-Os`) exhibits the steepest growth, which is expected since `gcc -Os` produces the smallest binaries.

Overall, these results suggest that, at least when compiling BenchGen programs, neither `clang` nor `gcc` exhibit asymptotic performance bugs of the kind reported in previous work [18, 32].

[3]Available at https://benchmarksgame-team.pages.debian.net/benchmarksgame/ as of September 20th.

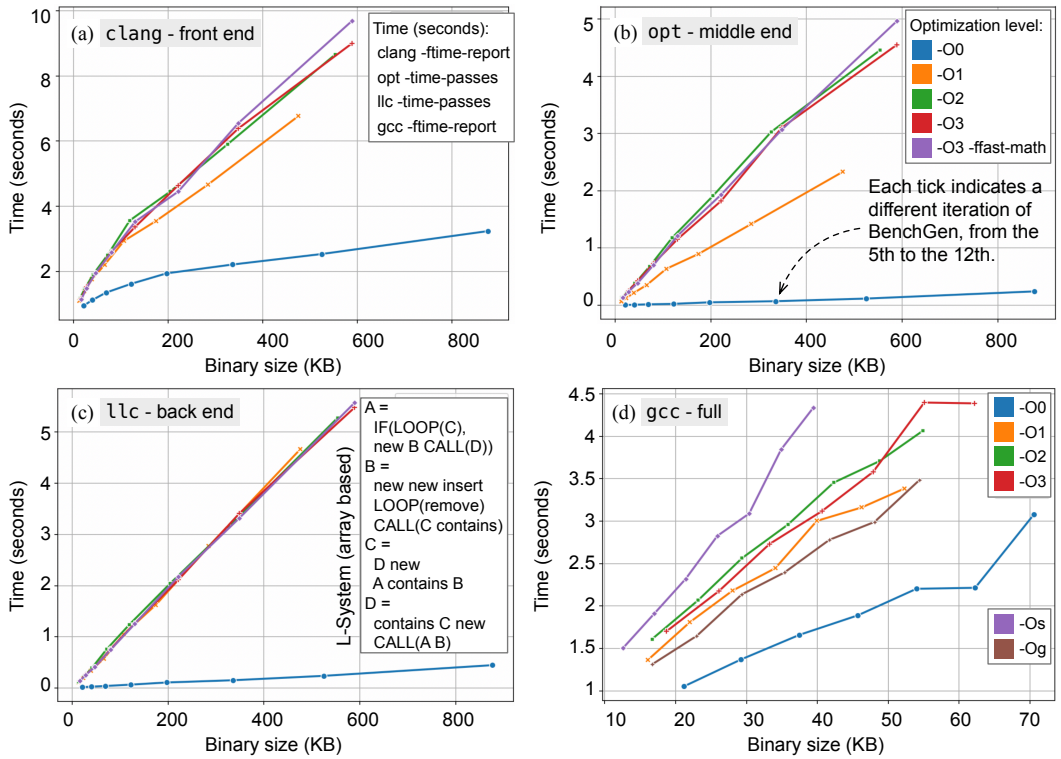

Fig. 14. Empirical evaluation of the asymptotic behavior of different compilation phases of `clang` and of the full compilation pipeline of `gcc`.

4.4 CS4: The Evolution of gcc

GNU `gcc`, currently at version 14, is one of the oldest and most widely used C compilers. Over the years, it has undergone many changes, including support for additional languages and continuous performance improvements. This case study compares different versions of `gcc` across multiple optimization levels using three metrics:

- Execution time of the compiled program (measured with `hyperfine`).
- Compilation time required to build the program (also measured with `hyperfine`).
- Binary size of the `.text` segment of the executable (collected with the Linux `size` utility).

For this comparison, we used BenchGen with the first L-System shown in Figure 12. The program generated at the 10th iteration of this grammar was compiled with six versions of `gcc`.

Discussion. Figure 15 summarizes results for six optimization levels: `-O0`, `-Og`, `-O1`, `-Os`, `-O2`, and `-O3`. Each group of bars corresponds to a compiler version: `gcc5`, `gcc7`, `gcc9`, `gcc11`, `gcc13`, and `gcc14`. Below we discuss some of the insights of this study:

Compilation time: We observe a steady increase in compilation time with each new version of `gcc`, particularly at higher optimization levels such as `-O3`. Comparing versions 5 and 14, compilation time rose by about 42% over nine years, with a 31% increase even at `-O0`. This is unsurprising: new optimization passes have been added over time, which increases compile time in exchange for better code quality.

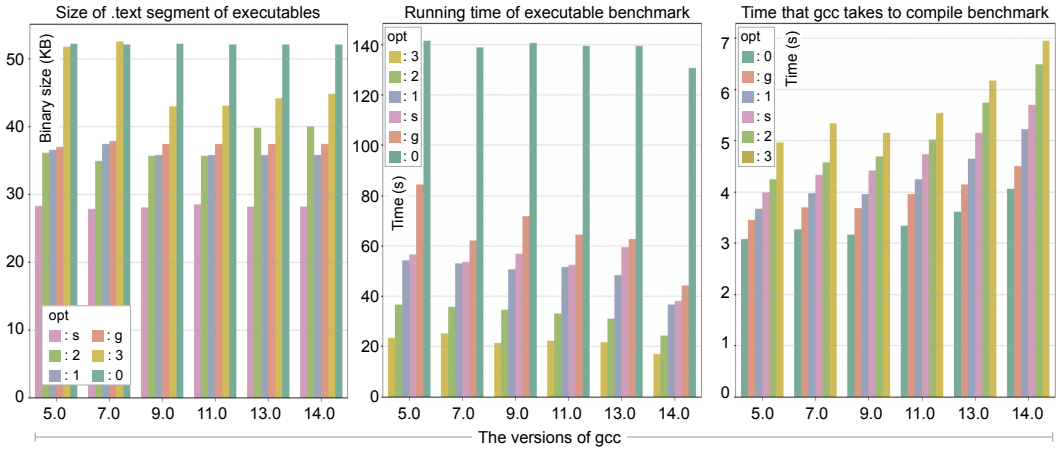

Fig. 15. Evolution of gcc from version 5 to 14 across six optimization levels. Results are shown for (left) binary size of the `.text` segment, (center) execution time of the benchmark, and (right) compilation time.

Execution time: Performance has improved at the higher optimization levels. Comparing `gcc5` to `gcc14`, execution time decreased by roughly 27% at `-O3`, 32% at `-O1`, and as much as 47% at `-Og`. The gains at `-O0` were modest, about 7%. These results indicate that newer versions of `gcc` generate more efficient code, especially at intermediate optimization levels.

Binary size: Binary size has remained stable across versions. At `-Os`, code size decreased only 0.34% between `gcc5` and `gcc14`. At `-O3`, binary size initially increased from `gcc5` until `gcc7`, dropping noticeably in `gcc9`. Past this point, we again observe a small increase until `gcc14`. However, in this version code is still almost 10% smaller than in `gcc5`.

Over the evolution from version 5 to 14, `gcc` shows a clear trade-off: longer compilation times in exchange for more efficient executables. While binary size has changed little, execution performance has improved noticeably, particularly at `-Og` and `-O3`. These results highlight an important trend: the `gcc` community has consistently prioritized code quality and runtime efficiency over compilation speed, accepting longer build times in return for measurable performance gains.

4.5 CS5: Profile Guided Optimizations in `clang`

Profile-Guided Optimization (PGO) is a compilation technique that leverages runtime information collected from representative executions of a program to improve the quality of the generated code. By replacing purely static heuristics with empirical execution data, PGO enables the compiler to make more informed decisions regarding branch prediction, function inlining, loop unrolling, and code layout, thereby enhancing both performance and efficiency. In typical C, C++ and Rust compilers, PGO is integrated into the standard optimization pipeline rather than constituting a separate optimization level. Specifically in `clang`, PGO serves as a refinement to existing optimization levels, most notably `-O2` and `-O3`. When profile data is provided via the `-fprofile-use` flag, `clang` applies the chosen baseline optimizations but augments them with profile-driven insights, generating code that more accurately reflects the observed runtime behavior of the program.

This section illustrates how BenchGen can be used to evaluate the effectiveness of profile-guided optimizations (PGO). As described in Section 3.2, the control flow of a program generated by BenchGen is governed by the variable `PATH`. To assess the performance gains provided by PGO, we conduct the following experiment:

(1) Generate four program variants (Generations 6, 7, 8, and 9) using the following grammar with array structures:

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

(2) Compile each program at the `-O2` optimization level with PGO enabled[4].
(3) During the "*Training Phase*," collect profile information by running each program with $\texttt{PATH} = 2^0$ (i.e., $\texttt{PATH} = 1$).
(4) During the "*Testing Phase*," for $i = 1$ to 63:
   (a) Execute each program with $\texttt{PATH} = 2^i - 1$, e.g., $\texttt{PATH} = 0b1$, $\texttt{PATH} = 0b11$, etc.

Let t denote the execution time of the program compiled with `clang -O2` without profile data, and let t_i denote the execution time with $\texttt{PATH} = 2^i$. Then, the ratio t/t_i quantifies the relative speedup due to profile-guided optimizations as the control-flow path diverges by i bits.

Discussion. Figure 16 shows the data collected during this experiment. The results in Figure 16 reveal two distinct behaviors of profile-guided optimizations. In the expected case (Generations 6–8), the performance trend matches our intuition: when the execution path during testing closely resembles the path exercised during training, we observe significant speedups, often reaching 2x. Similar results on real applications have been reported in previous work [8, 27, 34].

As the number of bit flips in `PATH` increases, however, the executed control flow diverges from the profiled run, and the benefit of PGO diminishes. Eventually, for highly divergent inputs, the profiled and non-profiled binaries converge to similar performance, as the compiler's profile-driven decisions no longer apply.

Profile Overfitting: The unexpected case arises in Generation 9. Here, PGO yields a very large speedup when execution follows the training path (more than 2.2x). Yet, as the path diverges, the optimized binary becomes slower than its non-profiled counterpart. This behavior suggests an instance of profile overfitting: because PGO biases optimizations toward the observed hot path, code along unprofiled paths may suffer from unfavorable decisions.

A plausible explanation for this effect is the difference in the number of functions inlined with and without profiling data. Clang/LLVM makes inlining decisions at the call-site level. A function call along the hot path may be inlined aggressively, while the same call in a cold path may be left uninlined, leading to inconsistencies in call overhead, code size, and optimization opportunities. Combined with profile-driven code layout and branch prediction, this selective optimization can make non-profiled paths slower than in a neutral, non-PGO binary. Thus, Generation 9 highlights a limitation of PGO: while it can deliver substantial performance gains when training inputs are representative, it may degrade performance when profile data overfits to a narrow subset of execution paths.

4.6 CS6: Data Structures in GLib

The GNOME Library (GLib) is a C library developed as part of the GNOME project that provides a wide range of data structure implementations, utility functions, and portability abstractions. Among its generic containers are dynamic arrays (e.g., GArray, GPtrArray), byte arrays (GByteArray), singly linked lists (GSList), doubly linked lists (GList), double-ended queues (GQueue), hash tables (GHashTable), balanced binary trees (GTree), and others. This variety exists because no single container is optimal for all purposes: some favor fast insertion or deletion (especially at specific positions), others provide efficient random access, better memory locality, sorted traversal,

[4]We report results for `clang` at the `-O2` level because the speedups achieved with PGO are comparable to those observed at `-O3`, and consistently higher than those obtained at `-O1`.

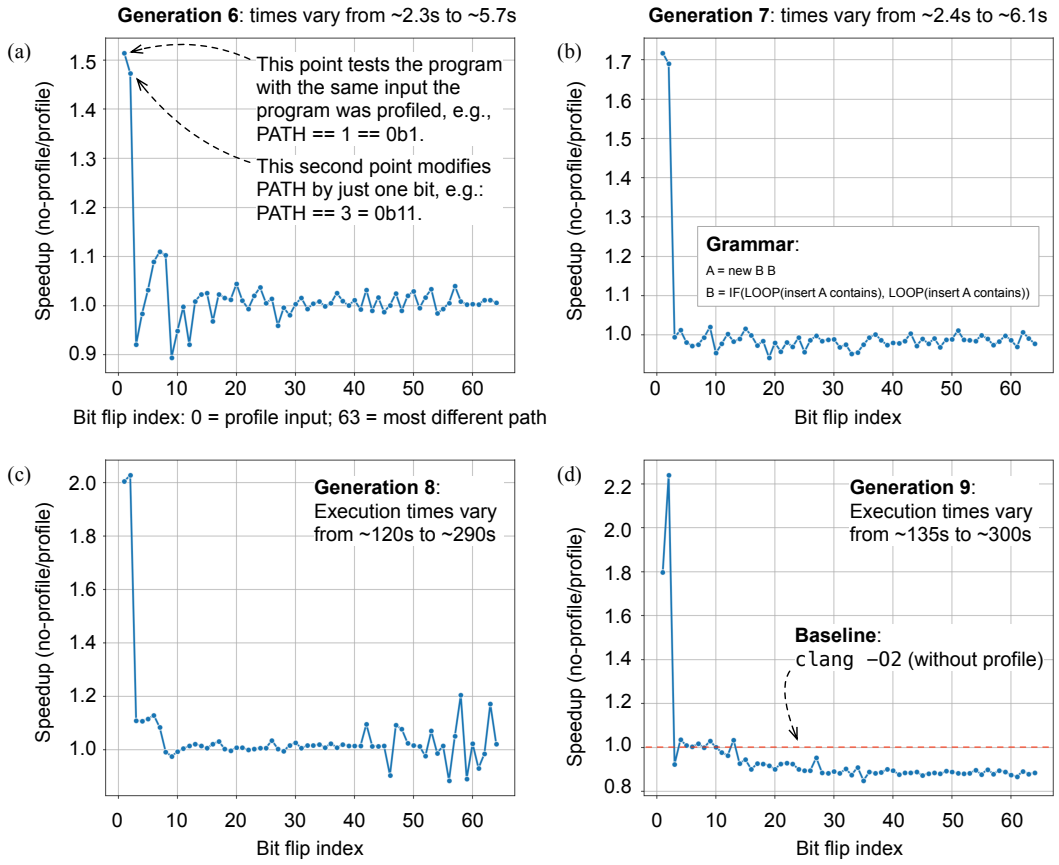

Fig. 16. The impact of profile guided optimizations on `clang -O2`.

or minimal overhead. By offering multiple data structures, GLib allows programmers to select the container best suited to their performance, memory, and semantic requirements.

BenchGen enables the systematic evaluation of the dynamic behavior of these data structures. As discussed in Section 3.1.2, the current implementation of the C code generator supports all twelve containers provided in the default distribution of the library[5]. This section demonstrates how such support can be used to compare the performance characteristics of different containers. To this end, we employ three grammars designed to stress insertion (`insert`), deletion (`remove`), and search (`contains`) operations through a large number of executions. The adopted methodology is as follows:

(1) Use the non-recursive L-System below to generate the last iteration of a program:

```
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;
```

[5]A detailed description of these containers is available at https://docs.gtk.org/glib/data-structures.html.

(2) The `operations` symbol acts as a placeholder, expanded into a random sequence of operations (`insert`, `remove`, and `contains`) in varying proportions.
(3) The seed string for the grammars is always "`new A0 B`".
(4) Measure the running time of each program using `hyperfine`. Only one warm-up execution is performed due to the long total runtime of all experiments.
(5) Programs with a similar number of operations are grouped together, and we report the average running time within each group.

The first three productions of the grammar above populate the container with a minimum of random integers. The last step of the above methodology—grouping programs by the number of operations—is necessary because it is not possible to precisely control how many iterations are executed within loops generated by BenchGen. Although the maximum loop trip count is bounded as an input parameter, the actual number of iterations is chosen randomly within that bound.

Discussion. Figure 17 reports the running time of programs generated using the methodology described above. As the number of operations grows, hash tables and balanced trees clearly outperform lists and queues. Between these latter two containers, the queue consistently performs better than the list, except for very small numbers of insertions. This behavior is expected: while both data structures support $O(1)$ insertions, in the doubly linked GList two pointers must be updated, whereas in GQueue only one pointer is modified.

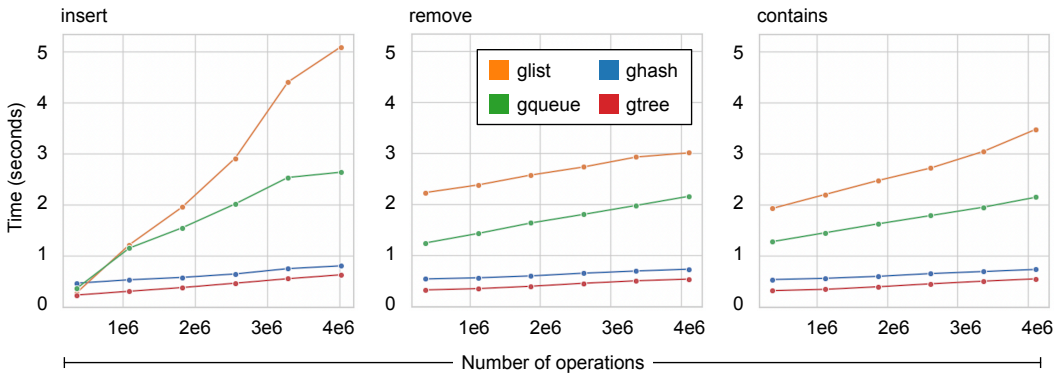

Fig. 17. Comparison of four data structures from GLib: GHashTable, GTree, GQueue, and GList.

At the scale of operations shown in Figure 17, GTree appears consistently faster than GHashTable, regardless of the operation. However, this situation changes when the number of operations increases further, as illustrated in Figure 18. Around 6 million insertions, the hash table overtakes the balanced tree, and similar crossover points can be observed for deletion and search operations.

This behavior seen in Figure 18 can be explained by the different asymptotic properties and implementation trade-offs of the two data structures. Balanced trees guarantee $O(\log n)$ time per operation, while hash tables typically achieve $O(1)$ average time, at the cost of higher constant factors due to hashing and possible collisions. For smaller workloads, the overhead of maintaining a hash table, such as computing hash values and managing bucket arrays, outweighs the logarithmic costs of tree operations, making GTree faster in practice. As the number of operations grows, however, the logarithmic scaling of trees eventually becomes the dominant factor, allowing GHashTable to surpass GTree. Thus, the crossover point observed in Figure 18 reflects the causal interaction between constant-factor overheads and asymptotic growth rates in these data structures.

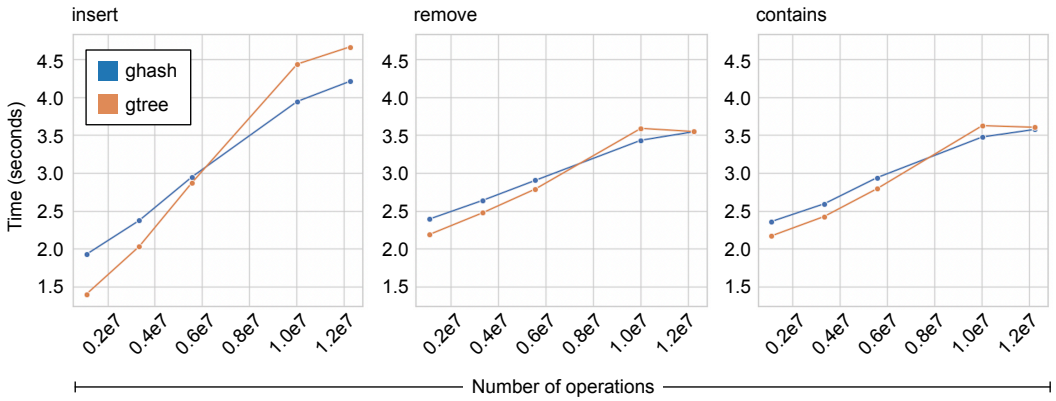

Fig. 18. Comparison of balanced trees (GTree) and hash tables (GHashTable) from GLib.

4.7 CS7: Comparison with CSmith

Several programming language fuzzers enjoy widespread popularity. The main goal of such tools, including CSmith [41] and **YarpGen** [25] (for C) or ChiGen [38] (for Verilog), is to uncover bugs in language processing systems such as compilers or interpreters. The purpose of BenchGen, however, is different: it was designed to evaluate the performance of these systems. Its ability to expose bugs is limited, since the programs it produces are less diverse than those generated by a fuzzer like CSmith. On the other hand, BenchGen can stress-test compiler performance in ways that CSmith and related tools cannot. This section illustrates this point through the following challenge: generate a collection of programs whose execution time is comparable to the average runtime of Spec Cpu2017 programs, and whose behavior under optimization resembles that of real-world workloads.

Discussion. To compare CSmith and BenchGen under this challenge, we have used these two program generators to produce a collection of four benchmarks and compare them with the programs from Spec Cpu2017. The adopted methodology is described below:

(1) We ran the 28 Spec Cpu2017 programs that `clang`/LLVM 20.1 can compile.
(2) Using CSmith, we generated 10,000 programs and selected the four whose execution time was closest to the average runtime of the 28 Spec Cpu2017 programs compiled without optimizations.
(3) Using the four grammars described in Section 4.1, we generated four BenchGen programs whose runtime approximated the same average.
(4) We then optimized each collection of benchmarks (28, 4, and 4 programs, respectively) under different optimization flags.

Table 1 presents the results of this experiment. We stopped BenchGen as soon as we obtained programs whose running time was above the average Spec time, and chose the generation immediately before. However, it is not possible to steer CSmith in such a way. In other words, users cannot control CSmith to produce programs that execute within a given time frame. Most CSmith programs tend to run for only a very short time. Moreover, the impact of optimizations on these programs differs substantially from what we observe in real-world benchmarks such as those in the Spec collection, whereas the effects observed on BenchGen programs more closely resemble the behavior of genuine workloads. This difference between the effects of compiler optimizations on CSmith and real-world programs had been noticed in previous work [12].

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

Table 1. Mean cycles and ratios for different benchmarks across optimization levels.

5 Related Work

BenchGen is a tool designed to synthesize benchmarks. A number of works in the compiler literature have also proposed techniques for benchmark generation. In what follows, we review this body of work, emphasizing aspects that make BenchGen particularly well-suited for performance analysis.

Compiler Fuzzers. The key distinction between compiler fuzzers and BenchGen lies in their primary objectives. Most existing tools are designed to uncover bugs in language processing systems, whereas BenchGen is specifically aimed at analyzing their performance. Consequently, BenchGen would be less effective than a fuzzer such as CSmith for identifying errors in C compilers. On the other hand, it enables tasks that are not easily supported by traditional fuzzers, such as studying the asymptotic behavior of compiler optimizations, comparing compilers in terms of compilation time, code size, and execution speed, or analyzing the evolution of a single compiler across multiple versions with respect to these same metrics.

Benchmark Synthesizers. The development of compilers requires *benchmarks*. For this reason, some of the most celebrated papers in programming languages describe benchmark suites, such as Spec CPU2006 [20], MiBench [17], Rodinia [7], etc. These benchmarks are manually curated and usually comprise a small number of programs. A few years ago, Cummins et al. [9] demonstrated that this small size fails to cover the space of program features that a compiler is likely to explore during its lifetime.

The generation of benchmarks for tuning predictive compilers has been an active research field over the last ten years. Early efforts aimed at the development of predictive optimizers used synthetic benchmarks designed to find compiler bugs. Examples of such synthesizers include CSmith [41], LDRGen [2], and Orange3 [29, 30]. Although conceived as test case generators, these tools have also been used to improve the quality of optimized code emitted by mainstream C compilers [3, 19]. Even CompilerGym [10], a tool for applying machine-learning-based code optimization techniques, provides randomly produced CSmith programs. However, more recent developments indicate that synthetic codes tend to poorly reflect the behavior of programs written by humans; consequently, they produce deficient training sets [12, 15]. Section 4.7 discusses precisely these issues with CSmith: the difficulty of using it to generate programs that run for a reasonable amount of time and the ease with which compilers optimize such programs.

In order to produce more realistic benchmarks, a vast literature on extracting programs from repositories has recently emerged, seeking to build benchmark suites for training compilers. Some of these works aim to generate benchmarks to feed machine learning models [14, 16, 33], for example. This literature has some shortcomings when compared to BenchGen:

- Very large benchmark suites, such as AnghaBench [12] or Poj [28], compile but do not run.
- Benchmark suites designed to run sometimes exhibit undefined behavior, such as ExeBench [1] and ColaGen [5].
- Techniques that generate benchmarks using LLMs, such as the system of Italiano and Cummins [21], fail to produce large programs and take a long time to generate even a small number of programs, as recently shown by Guimarães *et al* [35].
- The Jotai collection [22], which was built from programs that run without undefined behavior, contains only very small programs. In our attempt to run the programs available in CompilerGym [10], the longest-running program took only 0.017 seconds when compiled with `clang -O0`.

6 Conclusion

This paper has presented a methodology for generating benchmarks based on the observation that programs often exhibit self-similar structures, and thus can be described as derivations of L-Systems. We validated this idea through the design and implementation of BenchGen, a tool capable of producing benchmarks in multiple programming languages. In contrast to typical fuzzers, BenchGen is multilingual and allows users to generate programs of arbitrary size whose execution exercises complex control-flow patterns.

Future Work. Beyond the concrete implementation of BenchGen, the key contribution of this paper is the proposal that programs can be systematically generated as instances of L-Systems. The particular L-System flavor explored here employs a combination of seven constructs (three control-flow structures and four data-structure behaviors). Nevertheless, the approach can be extended to a much broader set of syntactic features. For example, the current version of BenchGen generates programs restricted to a single data structure at a time, whereas nothing in the formalism prevents richer combinations of data structures. Likewise, additional control-flow constructs such as `switch`, `do-while`, or even `goto` could be naturally incorporated. We leave the investigation of such extensions as promising directions for future work.

Acknowledgment

This project was supported by Google and by FAPEMIG (Grant APQ-00440-23). We are grateful to Xinliang (David) Li and Victor Lee for their efforts in making the Google sponsorship possible. We also thank Lucas Victor Silva for producing the data in Table 1, and Bruno Pena Baêta and Matheus Alcântara for their work on an earlier version of BenchGen.

References

[1] Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F. P. O'Boyle. Exebench: an ml-scale dataset of executable c functions. In *MAPS*, page 50–59, New York, NY, USA, 2022. Association for Computing Machinery.

[2] Gergö Barany. Liveness-driven random program generation. In *LOPSTR*, pages 112–127, Heidelberg, Germany, 2017. Springer.

[3] Gergö Barany. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 82–92, New York, NY, USA, 2018. ACM.

[4] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free c programs: the frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, July 2021.

[5] Maksim Berezov, Corinne Ancourt, Justyna Zawalska, and Maryna Savchenko. COLA-Gen: Active Learning Techniques for Automatic Code Generation of Benchmarks. In Francesca Palumbo, João Bispo, and Stefano Cherubin, editors, *PARMA-DITAM*, volume 100 of *Open Access Series in Informatics (OASIcs)*, pages 3:1–3:14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[6] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. Julia: dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, Washington, DC, USA, 2009. IEEE.

[8] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: automatic feedback-directed optimization for warehouse-scale applications. In *CGO*, page 12–23, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *CGO*, pages 86–99, Piscataway, NJ, USA, 2017. IEEE.

[10] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. Compilergym: Robust, performant compiler optimization environments for ai research. In *CGO*, pages 92–105, New York, USA, 2022. IEEE.

[11] Charlie Curtsinger and Emery D. Berger. Coz: finding code that counts with causal profiling. In *SOSP*, page 184–197, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. AnghaBench: A suite with one million compilable C benchmarks for code-size reduction. In *CGO*, pages 378–390, Los Alamitos, CA, USA, 2021. IEEE.

[13] Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. *SIGPLAN Not.*, 47(1):533–544, January 2012.

[14] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *FSE*, page 254–265, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. A case study on machine learning for synthesizing benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pages 38–46, New York, NY, USA, 2019. ACM.

[16] Georgios Gousios and Diomidis Spinellis. Mining software engineering data from github. In *ICSE-C*, page 501–502, Washington, DC, US, 2017. IEEE Press.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC*, pages 3–14, Washington, DC, USA, 2001. IEEE.

[18] Xue Han and Tingting Yu. An empirical study on performance bugs for highly configurable software systems. In *ESEM*, New York, NY, USA, 2016. Association for Computing Machinery.

[19] Atsushi Hashimoto and Nagisa Ishiura. Detecting arithmetic optimization opportunities for c compilers by randomly generated equivalent programs. *IPSJ Transactions on System LSI Design Methodology*, 9:21–29, 2016.

[20] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[21] Davide Italiano and Chris Cummins. Finding missed code size optimizations in compilers using large language models. In *International Conference on Compiler Construction*, page 81–91, New York, NY, USA, 2025. Association for Computing Machinery.

[22] Cecilia Conde Kind, Michael Canesche, and Fernando Magno Quintao Pereira. Jotai: a methodology for the generation of executable c benchmarks. Technical report, Technical Report 02-2022. Universidade Federal de Minas Gerais, 2022.

[23] Kota Kitaura and Nagisa Ishiura. Random testing of compilers' performance based on mixed static and dynamic code comparison. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 38–44, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299, 1968.

[25] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[26] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[27] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. Vespa: static profiling for binary optimization. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

[28] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, page 1287–1293, Palo Alto, CA, US, 2016. AAAI Press.

[29] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions. *IPSJ Trans. System LSI Design Methodology*, 7:91–100, 2014.

[30] Kazuhiro Nakamura and Nagisa Ishiura. Introducing loop statements in random testing of c compilers based on expected value calculation, 2015.

[31] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

[32] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *PLDI*, page 369–378, New York, NY, USA, 2015. Association for Computing Machinery.

[33] David N Palacio, Alejandro Velasco, Daniel Rodriguez-Cardenas, Kevin Moran, and Denys Poshyvanyk. Evaluating and explaining large language models for code using syntactic structures, 2023.

[34] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *CGO*, page 2–14. IEEE Press, 2019.

[35] Gabriel Ricardo, Natanael Santos Junior, Flavio Figueiredo, and Fernando Pereira. On the practicality of llm-based compiler fuzzing. In *SBLP*, pages 59–66, Porto Alegre, RS, Brasil, 2025. SBC.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address sanity checker. In *USENIX ATC*, page 28, USA, 2012. USENIX Association.

[37] Bob Strecansky. *Hands-On High Performance with Go: Boost and optimize the performance of your Golang applications at scale with resilience.* Packt Publishing Ltd, 2020.

[38] João Victor Amorim Vieira, Luiza de Melo Gomes, Rafael Sumitani, Raissa Maciel, Augusto Mafra, Mirlaine Crepalde, and Fernando Magno Quintão Pereira. Bottom-up generation of verilog designs for testing eda tools, 2025.

[39] Zheng Wang and Michael F. P. O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[40] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.