Benchmark Generation via L-Systems

Vinicius Francisco da Silva
UFMG
Belo Horizonte, Brazil
silva.vinicius@dcc.ufmg.br

Fernando Magno Quintão Pereira
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br

***Abstract*—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 the C programming language. 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 in two ways. First, it explains how such programs allow check of the complexity of different phases of the compilation process. Second, it shows how these programs allow for the comparison of C compilers, such as gcc, clang, and tcc, in terms of compilation time, size, and speed of the generated code.**

***Index Terms*—Fractal, Code generation, Program synthesis.**

I. Introduction

Compilers are complex tools whose testing relies on programs written in the target language. However, the number of available benchmarks for any given compiler is often limited [7]. To address this shortcoming, several tools have been developed that automatically generate test programs [9]. This process, known as *fuzzing* [6], is a widely adopted methodology for uncovering bugs such as crashes and memory leaks. In the domain of compilers, fuzzers like Csmith [9] and YARPGen [5] are capable of generating random C programs for stress testing and static analysis. More recently, fuzzers such as Fuzz4All [8] have leveraged Large Language Models (LLMs) to produce test programs not only for compilers but also for constraint solvers, interpreters, and software systems with accessible APIs.

Despite the abundance of tools for generating random compiler inputs [9], current systems exhibit several limitations. One key issue is the lack of control over the size of the generated programs. For example, Csmith—the most well-known fuzzer of C compilers—does not provide any mechanism to tune the output size. Instead, it produces programs whose sizes follow a normal distribution. When compiled with Clang v9.0.1 using the `-O0` flag, these programs typically contain an average of 20,190 LLVM instructions, with a standard deviation of 3,650 and a median of 19,161 instructions [2]. Other fuzzers, such as YARPGen [5], LDRGen [1], and Orange3 [3], display similar behavior. As a result, these tools are less suited for performance testing of compiler components such as parsing, semantic analysis, or code generation.

Programs via L-Systems. To overcome this limitation, this paper proposes a methodology to stress-test compilers for *performance*, instead of *correctness*. This methodology allows the generation of programs whose size can be precisely controlled by the user. This approach enables the creation of synthetic code of virtually arbitrary size, constrained only by available resources such as code generation time and storage space.

The central idea of this work builds on the observation that programs often exhibit recursive, self-similar structure. For instance, the branches of a control structure (e.g., *if-then-else*) are themselves programs that may contain further control structures. To exploit this observation, we introduce a program generation method based on *L-systems* [4]. Originally developed by Aristid Lindenmayer to model the growth of biological organisms, L-systems provide a formal grammar-based framework that we extend to code generation. This paper describes how families of self-similar programs can be captured by a specific L-grammar. From such grammars, program structures are generated through iterative rewriting. Each resulting string corresponds to the blueprint of a program built around a core data structure, such as an array or a list.

II. 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 1 illustrates how this formalism operates.

Example 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.*

A. Programs as Self-Similar Structures

L-systems exhibit a property known as *self-similarity*, meaning that structures contain smaller copies of themselves across

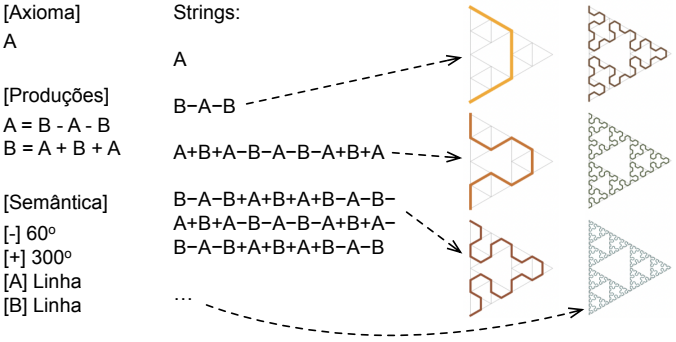

Fig. 1. L-system describing the Sierpinski Triangle.

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 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 reuse further reinforce this pattern: generic routines can be instantiated repeatedly across different abstraction levels, as Example 2 explains.

Example 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.*

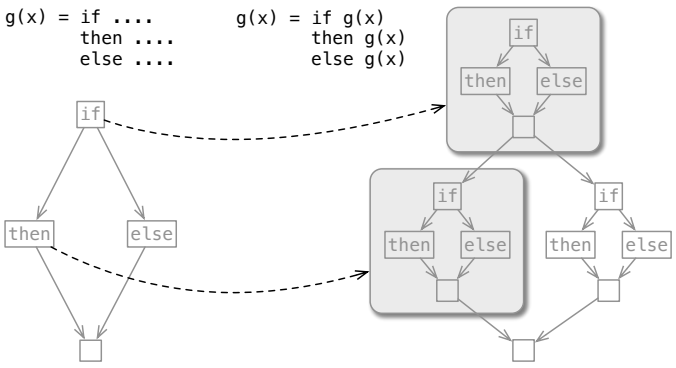

Fig. 2. The self-similar nature of computer code.

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 1, but instead of producing geometric patterns, it synthesizes C code constructions with executable semantics.

III. Code Generation via L-Systems

The tool LGen, developed in this work, generates C programs that manipulate data structures. Section III-A introduces the core building blocks used in program construction, while Section III-B describes semantics of these building blocks.

A. 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`.

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

$$
\begin{array}{lcll}
b & ::= & \text{IF}(b_{\text{cond}}, b_{\text{then}}) & ;; \mathit{if\_then} \\
& \mid & \text{IF}(b_{\text{cond}}, b_{\text{then}}, b_{\text{else}}) & ;; \mathit{if\_then\_else} \\
& \mid & \text{LOOP}(b_{\text{cond}}, b_{\text{body}}) & ;; \mathit{while} \\
& \mid & \text{CALL}(b) & ;; \mathit{function\_call}
\end{array}
$$

Each of these constructs corresponds to a familiar programming construct: conditional branches (`if-then`, `if-then-else`), loops (`while`), and function calls. Example 3 provides an illustration.

Example 3. *Figure 3 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.*

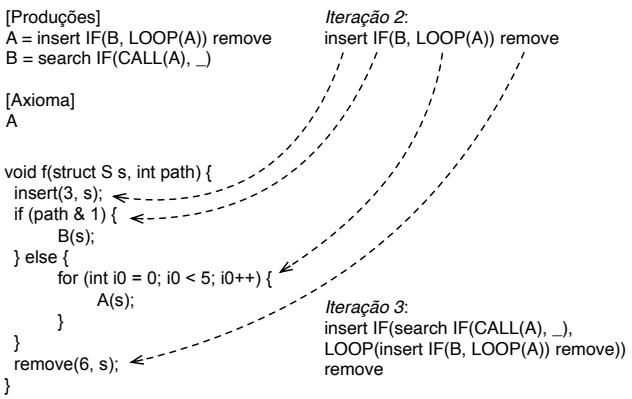

Fig. 3. Example of an L-grammar used to define programs.

2) Behavior Blocks: The dynamic behavior of LGen programs emerges from interactions with data structures. All data structures within a program must share the same type. Currently, LGen supports arrays of integers and sorted linked lists of integers. The supported operations are defined by 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.

Users of LGen define the implementation of each behavior block. As such, the semantics of these operations depends on user-provided code. Example 4 demonstrates how three such operations may be defined for programs working with arrays.

Example 4. *Figure 4 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.*

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

B. Execution Flow

Programs generated by LGen are executable. Their control flow is governed by a variable named `path`, of type `unsigned long`, which determines the outcome of conditional branches. Specifically, the i-th bit of `path` defines the result of all conditional tests at nesting depth i. This mechanism is clarified in Example 5.

Example 5. *Figure 5 depicts the control flow graph of a program organized into nested regions. Each region has a defined nesting depth: a region R at depth d is nested within d other regions. The outcome of the conditional that initiates R is controlled by the d-th bit of* `path`*. For example, in Figure 5, the conditional in the block* `path % 9` *is governed by* `path & 2`*, since it lies within two enclosing regions.*

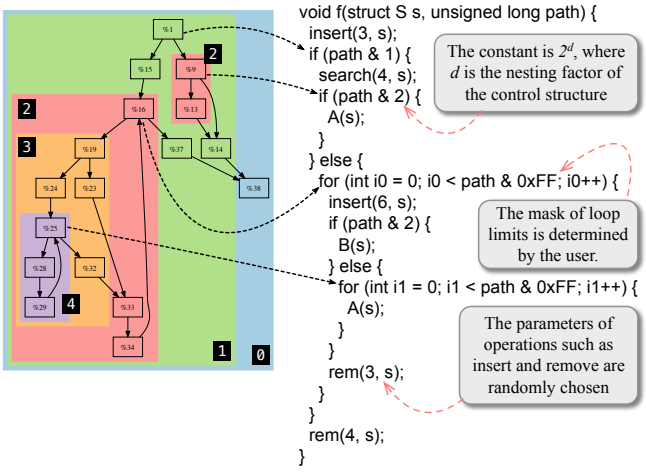

Fig. 5. The control flow of a synthetic program is determined by the `path` parameter.

1) Function Calls: LGen 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. Example 6 illustrates in detail how interprocedural code generation is handled.

Example 6. *Figure 6 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.*

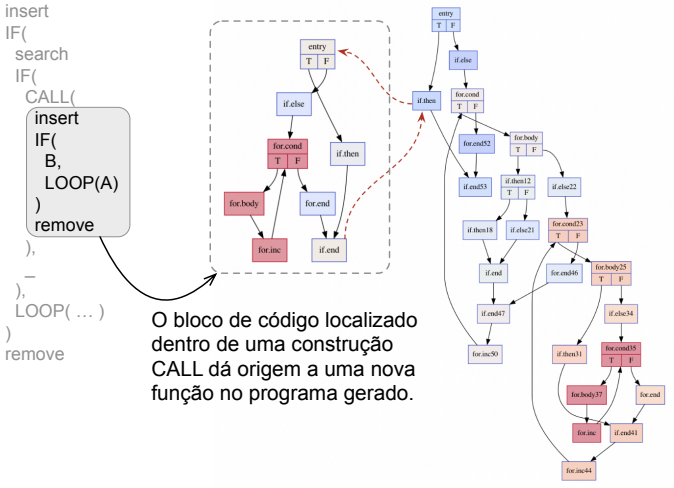

Fig. 6. Control flow of a program containing a `CALL` block.

Note that all occurrences of `CALL`(b) with identical strings b result in calls to the same function. To avoid redundancy, LGen maintains a table mapping strings to functions, ensuring

that identical strings refer to the same function instance. The functions generated from a given L-specification can be either grouped into a single file or distributed across multiple files, depending on a configuration parameter in LGen.

a) Parameter Passing: Functions generated by LGen 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 5.

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 7 illustrates this mechanism.

Example 7. *Figure 7 illustrates parameter passing in a program generated by LGen. 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 5.*

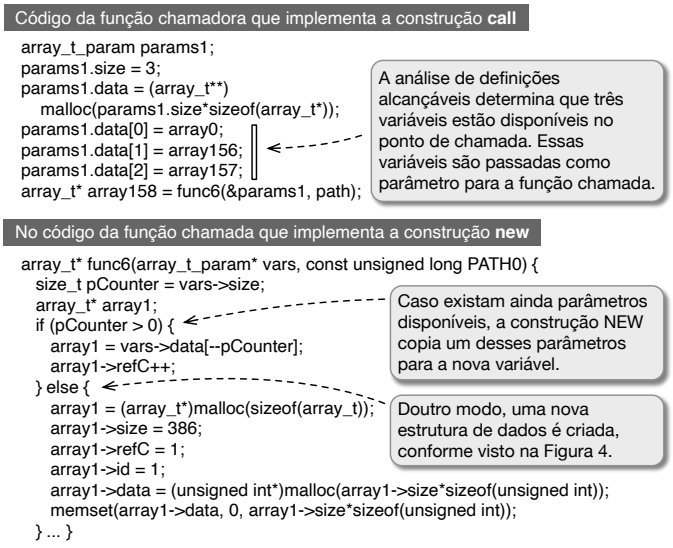

Fig. 7. Parameter passing in programs created by LGen.

2) Memory Management: Programs generated by LGen do not suffer from memory leaks, despite frequently relying on heap allocation. To prevent leaks, LGen 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 LGen includes an additional field, `refC`, which stores the current reference count. Example 8 shows how reference counting is used in LGen to prevent memory leaks from happening.

Example 8. *Figure 8 illustrates how LGen 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 LGen.*

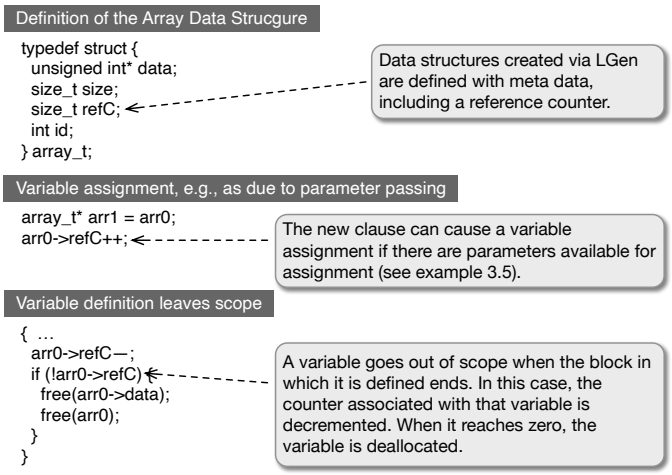

Fig. 8. Reference counter implementation.

Acknowledgment

This project is supported by Google and by FAPEMIG (Grant APQ-00440-23). We thank Xinliang (David) Li and Victor Lee for their efforts in making the Google sponsorship possible.

References

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

[2] 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.

[3] 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.

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

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

[6] 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.

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

[8] 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.

[9] 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.