Bear: BRIL Compiler Fuzzer

STEPHEN VERDERAME

Stephen Verderame. 2023. Bear: BRIL Compiler Fuzzer. 1, 1 (December 2023), 9 pages.

1 INTRODUCTION

BRIL (Big Red Intermediate Language) is a compiler intermediate representation used in CS 6120. As part of that class, students write various compiler passes optimizing BRIL programs including local value numbering (LVN), dead code elimination (DCE), and loop-invariant code motion (LICM). For testing BRIL programs, there is a benchmark suite of around 100 programs, some handwritten and some generated from one of the BRIL frontends. A student will typically test their compiler passes by writing a few handwritten tests to explicitly invoke corner cases of their optimization, and then running their optimization on all benchmarks, ensuring the program output doesn't change and (hopefully) observing a reduction in the number of dynamic instructions.

While this isn't too bad, it's also not great either. Benchmarks aren't specially designed to test certain optimizations, and it's pretty easy to miss a corner case that just doesn't occur in the limited set of benchmarks. In fact, there have been many cases where I caught a bug in one of my earlier compiler passes during work on an assignment later in the course.

To solve these problems, we introduce Bear, a compiler fuzzer which generates random BRIL programs designed to exercise common compiler passes students develop during the course. Bear is able to generate, run, and examine the results of around 9 tests per second, exploring the program space orders of magnitude more quickly than humanly possible.

2 OVERVIEW

Bear is a compiler fuzzer for mid-level optimization passes on the BRIL compiler intermediate representation. Bear aims to synthesize complex control flow and data dependency patterns to test optimizations and passes including local value numbering and other forms of redundancy elimination, simple peephole optimizations, dead code elimination, SSA conversion, and loop optimizations such as loop-invariant code motion and induction variable elimination. Bear always generates terminating, error-free programs which can run in a relatively short amount of time. To achieve this, Bear generates programs in a DSL with a grammar that enforces typing rules. It also uses abstract interpretation to enforce certain constraints on generated programs such as the worst-case number of loop iterations.

Programs are generated in the Bear DSL, Bare C, which are then lowered to the BRIL intermediate representation. The lowered program can then be run directly by the BRIL reference interpreter and the results can be compared with the result of running the program by the compiler being tested. Differences in the results would indicate an error in the compiler under test.

Bear aims to continuously generate programs to expose more bugs in the compiler under test. To do this, we employ novelty search to generate programs in disparate areas of the space of possible Bare C programs.

Novelty search is a genetic algorithm that rewards novelty (distance in feature space of the current program from previous programs) instead of taking fitness into account. Program feature space characterizes the runtime features of a program that make it distinct such as the number of loop iterations ran, the number of branches taken, and the amount of each type of control-flow structure ran. We construct a feature vector for each program consisting of this information and attempt to explore programs that are spaced apart in this feature space.

Author's address: Stephen Verderame, sev47@cornell.edu.

2023. 2023/12

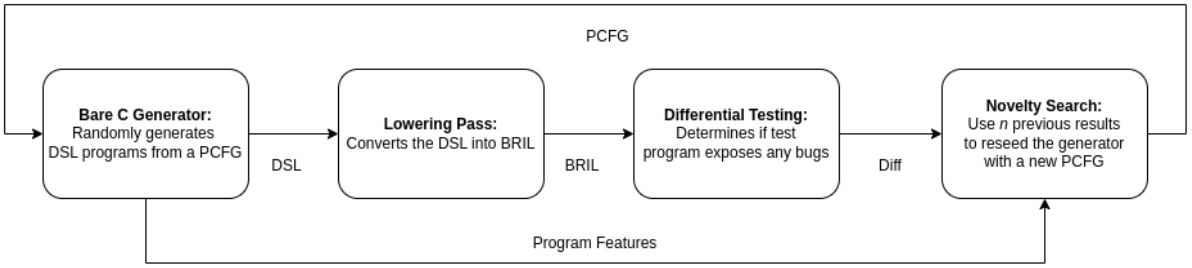

Fig. 1. Architecture of Bear Fuzzer

In §3, we discuss the Bear DSL and generating random programs. In §4, we discuss lowering the DSL into BRIL and performing differential testing. In §5, we discuss the novelty search algorithm. Finally, in §6, we evaluate the effectiveness of Bear in finding compiler bugs.

3 BEAR DSL

The Bear DSL (referred to as Bare C), is a language designed to stress complex control flow and data dependencies in BRIL. It consists of basic arithmetic and boolean operators, conditionals, `for`, `while`, and do-while loops, switches, `break`, `continue`, intra-procedural (local) try-catch, `throw`, `print`, and `return`. Values in the DSL can be one of two types: 64-bit signed integers or booleans. In order to always generate correctly typed programs, the grammar of the language enforces that only correctly typed programs can be formed into valid DSL abstract syntax trees.

The grammar is divided into 5 types of productions: arithmetic expressions, boolean expressions, statements, loop statements, and blocks. Arithmetic and boolean expressions are expressions that result in integral and boolean values, respectively. Statements do not result in a value and include things such as a variable assignment or a `return`. Loop statements are a superset of statements and also include loop-specific commands such as `break` and `continue`. Finally, blocks consist of one or more statements and control flow structures such as if-else, loops, and try-catch. We also include switch statements in Bare C to generate programs where one basic block has many predecessors to stress SSA conversion, and we include local try-catch, `break`, and `continue` to use BRIL's jump instruction in situations we can ensure program validity and termination at the DSL level. Bare C also includes a special syntactic construct for Duff's Device to generate programs with irreducible control flow.

Besides these unique block productions, Bare C also consists of other syntactic constructs not typical in many other languages for the sole purpose of allowing the generator to easily trigger optimizations. This strategy is used in various other compiler fuzzers such as YARPGen [4]. Bare C has non-terminals for redundant expressions, loop-invariant expressions (expressions whose values are constant from one loop iteration to the next), and induction variables to ensure programs that trigger redundancy elimination and loop optimizations are generated. A basic induction variable is an assignment inside a loop of the form i += c where c is loop-invariant. A derived induction variable is an assignment inside a loop of the form $j = i * a + b$ where a, and b are loop invariant and i is a basic induction variable.

The DSL also has a `dead` block which contains only dead code. Dead code does not have any side effects (such as prints) nor does it modify any variables that may be necessary for a computation with side effects.

3.1 Generating Programs

To generate programs in the Bear DSL, we randomly probe a PCFG, generating programs top-down. While the grammar of Bare C enforces things like `break` and `continue` only occurring in the body of loops, nothing

enforces that a `throw` statement must occur in the body of a `try` block. For constraints like this, we simply regenerate the invalid statement or expression until a valid one is yielded. This filtering and retry approach has seen success in fuzzers like CSmith [5]. Furthermore, unlike traditional synthesis problems, we aren't particularly interested in the shortest program. In fact, some argue that longer programs are preferred because they find more bugs [5]. Thus, a systematic enumeration approach like the one employed by Euphony would not work [2].

```
aexpr           ::=  aexpr + aexpr
                 |   aexpr - aexpr
                 |   aexpr / aexpr
                 |   aexpr * aexpr
                 |   n
                 |   redundant(aexpr)
                 |   loop-invariant(aexpr)
                 |   induction-var(aexpr * aexpr + aexpr)
                 |   var
bexpr           ::=  aexpr cmp aexpr
                 |   ...
expr            ::=  aexpr
                 |   bexpr
statement       ::=  var : = expr
                 |   return expr?
                 |   throw(n) expr?
                 |   print expr+
loop-statement  ::=  statement
                 |   break(n)
                 |   continue(n)
                 |   step
block           ::=  statement
                 |   if bexpr { block* } else { block* }
                 |   do { loop-block* } while var cmp aexpr
                 |   while var cmp aexpr { loop-block* }
                 |   for var in aexpr .. aexpr by aexpr { loop-block* }
                 |   match aexpr { (n => block*)+ }
                 |   try { block* } catch var? { block* }
                 |   switch aexpr { do { (case n:  loop-block*)+ } for var in aexpr .. aexpr by aexpr }
                 |   dead { block* }
program         ::=  block+
```

Fig. 2. Slightly simplified context free grammar of the Bear DSL. '?', '*', and '+' applied after a nonterminal production name indicate the nonterminal is optional, occurs 0 or more time, or one or more times, respectively.

The PCFG for Bear is not quite a unigram or a bigram model. To avoid a PCFG with thousands of probabilities, we do not duplicate all parts of the grammar, however, we do duplicate some parts of the grammar such as productions for blocks and statements in loops, and expressions that are part of conditional and loop guards. The idea is to allow the PCFG to be expressive enough to capture potential interesting relationships, especially

between conditionals and loops, but not be too expressive such that the explorable space of PCFGs becomes too large.

3.2 Incremental Dataflow Analysis

The Bare C language has loop-invariant and redundant nonterminals. When the generator selects to add one of these to the program being generated, it needs to know what expressions are redundant and loop invariant. To do this we perform an incremental dataflow analysis to keep track of defined variables, available expressions (which can be reused in an redundant expression), and loop-invariant variables.

Every time a new statement is generated, we perform dataflow analysis on that statement. For example, when the assignment `x := 10` is generated, we must kill all previous available expressions that use `x`. We keep track of the analysis facts in a hierarchical context, where each level of the context corresponding to the current level of scope nesting the generator is currently at. When we exit a child scope (such as the body of the `if`), we merge the corresponding child contexts with all of its siblings (in this example, the else block of the if-else) and then merge this final output with the parent context.

For loop invariance, we avoid the need to perform a traditional iterate until convergence algorithm by randomly selecting some variables to be immutable throughout the body of the loop before we generate any blocks in the loop body. This technique is also used to avoid generating programs which modify the loop iteration variable in ways that create nonterminating programs.

3.3 Validity and Termination

The approach as previously described will produce a *syntactically valid* program, however, it might cause errors such as division by zero or it may not terminate. We want to further constrain programs such that they terminate in a reasonable amount of time and do not contain any errors. We achieve this with an incremental abstract interpretation to compute the possible ranges every integer variable and expression may take on. Similar to the incremental analysis, with every statement generated, we abstractly interpret it to determine every variable's possible range of values.

This information allows us to solve simple algebraic equations to enforce the desired constraints. Consider the following: `x := 1000 / a`, where `a` has the range $[-100, 100]$. We want to enforce that $a \neq 0$. The simple approach currently used is to add to the divisor the absolute value of the divisor's lower bound, plus one. Therefore, we get the statement `x := 1000 / (a + 101)` where `(a + 101)` has the range $[1, 201]$.

Beyond division by zero. We also want to enforce program termination. We cap each loop to a maximum of 10,000 iterations. For generating `for` loops, we first randomly determine if the loop counter will increase or decrease. Then we generate an initializer expression, limit expression, and loop counter step. To avoid non-termination, the loop step must be positive for increasing loop counters, and negative for decreasing loop counters. We also enforce that the difference between the limit's upper bound and the initializer's lower bound, divided by the step's lower bound, must be less than max number of loop iterations for an increasing loop. Or for a decreasing loop, that the initializer's upper bound, minus the limit's lower bound, divided by the step's upper bound is less than the maximum number of iterations. These constraints are enforced by trying up to 3 times to generate initializers, limits, and steps that satisfy these constraints. If after 3 tries the maximum loop iteration is still too large, we apply algebraic correction to the last set of expressions similarly to how we correct for division by zero.

Generating `while` loops follows a similar procedure. We do not allow the loop guard to be any arbitrary expression, instead it must take the form of a comparison between an existing variable and a generated limit expression. The difference between `while` and `for` loops is that that `while` loops do not prescribe an initial value or a step expression. Instead, it chooses an existing variable as the loop counter and uses a worst-case

loop step of 1 (or -1 for decreasing loops) to generate a limit that is at most 10,000 away from the initial value of the loop variable. Then, when generating the body, it will continue generating blocks until all paths contain a step statement, `return`, `throw`, or `break`. A step statement takes the form of `i := i + e` where e is a positive expression for increasing loops and a negative expression for decreasing loops and i is the loop variable.

```
// a,b in [-100, 100]
x := a + 75; // x in [-25, 175]
while x < a * 50 + 100 {
    // worst-case: 5100 - -25 = 5125 iterations
    // x in [-25, 5100]
    match b {
        75 => {
            // break out of the closest parent
            break(0);
        }
        60 => {
            x := x + a + 200;
            // step: x in [75, 5400]
        }
        30 => {
            x := x + 1000;
            // step: x in [975, 6100]
            continue(0);
        }
        _ => {
            return;
        }
    }
    // x in [75, 6100]
}
// x in [-25, 6100]
```

Fig. 3. Example result of abstract interpretation to prove termination of a while loop

4 LOWERING AND DIFFERENTIAL TESTING

The way we observe program behavior is through the `print` statement. Therefore, during lowering we artificially insert extra `print` statements to ensure that we can observe differences in the unoptimized and optimized program behavior. Once lowered, we run the program directly with the BRIL reference interpreter. This serves as our ground truth program output. We then compare this output with the output produced by the reference interpreter after performing the passes and optimizations that we seek to test. This approach is known as *differential testing*, and it is how we can determine whether we've identified a bug (mismatch in outputs).

While running the program for differential testing, we also want to be able to observe the runtime behaviors of each program to characterize them. To achieve this, we first insert extra NOP instructions during lowering that have attached information on the high-level construct being run. For example, at the start of a loop, we insert an NOP instruction that contains an argument that specifies the current nested loop depth. During differential

testing, we run tests on a modified interpreter that will dump out a trace of all the instructions run (including these NOPs). The fuzzer can then examine the trace, and use this information to build a *behavior vector*.

A behavior vector is essentially a way to encode quantities that describe a program's runtime behavior as a vector of floating point values. For example, one element in the vector represents the maximum loop depth that was reached during program execution. Another element in the vector is a numeric representation of how the program failed. For example, success is encoded as a 0, failing by causing an exception or crash in any of the compiler stages being tested is encoded as a 6, producing less output than expected is encoded as a 2, producing the same amount, but a different trace of output is encoded as a 1, optimizations which cause a divide by zero is encoded as a 7, etc. This information is vital for novelty search, as we'll soon discuss.

5 NOVELTY SEARCH

The approach described so far can be summarized as:

(1) Randomly generate a Bare C program from a PCFG
(2) Lower the program to BRIL
(3) Pass the BRIL program through the compiler passes under test and determine if the result is different from running the program directly with the BRIL interpreter.

We can repeat this process, and continually use random probing to generate new programs, an approach that has seen success in fuzzers like CSmith [5]. However, it would require carefully tuning the probabilities of the PCFG, a problem made more difficult by the fact that there is no corpus of good Bare C programs.

Instead, we use a genetic algorithm known as *novelty search*, which rewards being different rather than optimizing a specific metric [3]. We use such an approach because we don't have a great metric to optimize. We could optimize the number of bugs found, but there can be vast areas in the space of programs that are bug-free and the gradient for this could be near 0. Furthermore, this metric could lead to the fuzzer getting stuck in a local optimum where it continually generates similar programs that expose the same bug. Therefore, the idea is to ensure the fuzzer generates many different tests that are adequately spread across the entire space of programs and exercise many different aspects of BRIL.

The workflow of novelty search for Bear is as follows:

(1) Generate a population of some number of random PCFGs.
(2) Use each PCFG to generate 3 random programs.
(3) Run each random program and use the trace to compute a behavior vector for each program.
(4) Using the behavior vectors, compute a *novelty* score for each program and assign the PCFG to have the median novelty of the programs it generated.
(5) Select a subgroup of the PCFG population to be used as parents for the new generation, with PCFGs that generate programs with a higher median novelty being more likely to be in this subgroup.
(6) Mutate, crossover, and reproduce individuals in the selected subgroup to spawn a new population of the next generation of PCFGs.
(7) Return to step 2.

5.1 Computing Novelty

For this to work, we need to define some notion of novelty. The way we do this is to first model runs of a program as behavior vectors in behavior space, as discussed in §4. Then, we define novelty as the average distance of a behavior vector to its k-nearest neighbors. For Bear, we just use a standard Euclidean distance but scale each element slightly differently so elements that are inherently larger numbers, such as the instruction count, don't dominate the novelty. Specifically, we classify measurements that scale with the number of dynamic instructions on a log scale before using them as elements in a behavior vector. This is because we care more about the

difference between 50 divisions and 0 divisions than the difference between 1200 and 8000 divisions. The former is a difference of 2 orders of magnitude while the latter is a difference of 0 orders of magnitude. So, for a program's behavior vector v, nearest neighbor vectors $\{x_1, ..., x_k\}$, and scaling function s, we compute:

$$\rho(v) = \frac{1}{k} \sum_{i=1}^{k} \sqrt{(s(v) - s(x_i)) \cdot (s(v) - s(x_i))}$$

where, for a program with n dynamic instructions and behavior vector element $a \geq 0$

$$s(a) = \begin{cases} \log(a+1) & \text{if } a \propto n \\ a & \text{otherwise} \end{cases}$$

Another view of novelty ($\rho(v)$) is the sparseness of the area around a point in behavior space. The more sparse the area around a behavior vector is, the more novel the behaviors of the program are.

The algorithm keeps track of an *archive* of individuals that, when they were generated, had a novelty exceeding a particular threshold. The nearest neighbors used to compute the novelty of a behavior vector are taken from individuals in this archive and the current generation.

5.2 Spawning the Next Generation

After assigning all individuals in the population a novelty score, we can begin spawning the next generation. Taking some ideas from *grammatical evolution*, we represent PCFGs linearly as a list of floating point probabilities during reproduction [1]. We call this the *genotype* since it encodes the set of programs we can generate and their observable behaviors (or *phenotype*). We use the PCFG to generate programs instead of direction mutation of the abstract syntax tree to more easily enforce validity constraints such as avoiding division by zero.

For a population $\mathcal{P}$, of size p, we select a subset $\mathcal{R}$ of size $r < p$ probabilistically, without replacement, based on the distribution where every $x \in \mathcal{P}$ has probability mass:

$$\frac{\rho(x)}{\sum_{x_i \in \mathcal{P}} \rho(x_i)}$$

$\mathcal{R}$ will be our reproductive set of individuals that can pass their genes onto the next generation. The way we selected $\mathcal{R}$ is known as *roulette wheel* selection. We use this method because it's possible for a good PCFG to generate 3 unoriginal programs, thus obtaining a low novelty score. Therefore, we want to prioritize PCFGs that we sampled novel programs from, without explicitly excluding any individuals. Once the reproducing set has been selected, we uniformly sample this set (with replacement) to select parents until enough offspring have been generated to fill the next generation. Each parent can generate an offspring in three ways, in order of most to least likely: crossover, mutation, and reproduction.

During crossover, another individual in $\mathcal{R}$ is chosen at random to be the second parent. The first parent and the second parent then swap a random contiguous section of their linearized PCFGs (genotype) to produce two offspring that enter the next generation. For a given pair of offspring and parents, this swapping can occur between 1 and 5 times, chosen at random.

During mutation, a single parent's genotype is mutated between 1 and 10 times, chosen at random. Each mutation will (in order of most to least likely) increase or decrease a probability in the PCFG by a random amount, or swap two probabilities in a PCFG.

Finally, during reproduction, a parent is simply copied as is, without modification, to the next generation.

Other methods of mutation and crossover are possible, along with different amounts and rates of these reproduction operators. These are simply determined by intuition and trial and error.

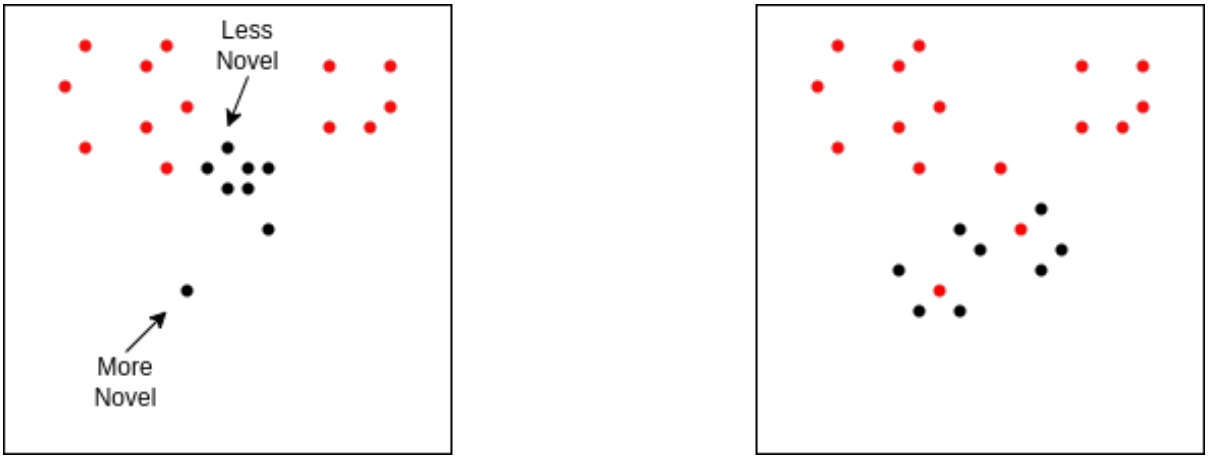

Fig. 4. Depiction of the difference between one generation and the next (on the right) using novelty search on a 2D behavior space. The archived individuals are displayed in red, and the current generation is displayed in black. In the next generation (on the right), the most novel individuals of the previous generation were saved to the archive and chosen to be the parents of the next generation. Thus, the next generation has individuals with behaviors somewhat similar to their parents.

5.3 Parallelizing Search

To speed up the search, and run more test programs, we also employ a *multi-population* search. Each population evolves, as described above, mostly independently and in separate threads. Every 8 generations, all of the populations will push the genotype of an individual from their most recent generation into a collective "gene pool." Each population will then use the genes from the gene pool to generate t individuals via crossover, mutation, and reproduction that will belong to each population's next generation. The remaining $p - t$ individuals will be generated as previously described.

This approach is especially useful to Bear, where much of the time spent by each thread will be on running test programs, reading traces, and dumping logs. This is because BRIL compiler passes can be written in any language, and typically take input and produce output on `stdin` and `stdout`, respectively. Thus, for greatest generality, invoking a compiler pass to test involves spawning a subprocess and performing file I/O. Therefore, by using many different threads, we can fill the CPU with active work to be done whenever a thread needs to stall to perform one of these activities.

6 EVALUATION

As is the case with many genetic algorithms, there are many knobs to tune, far more than I could meaningfully play with in the time I had to work on this project. Ideally, I would have some quantifiable measurements determining that certain settings are better than others, but unfortunately, this isn't the case. Ignoring time constraints, even something as tangible as bugs found doesn't equally compare different settings at this small scale due to the non-deterministic nature of generating random programs. Despite this, it's clear that Bear *works*, to some degree, by being able to find significant bugs in my compiler passes without devoting significant time to running the fuzzer.

In particular, within an hour of running the fuzzer, it already found several unique bugs as seen in Table 1.

DCE	LVN	SSA	LICM
≥ 2	0	1	≥ 2

Table 1. Number of unique bugs found by Bear in an hour for each optimization

DCE	LVN	SSA	LICM
≥ 2	1	1	≥ 2

Table 2. Number of total unique bugs found by Bear (~ 5 hours or less runtime for each optimization)

I haven't gone through and debugged everything that Bear exposed, so these numbers are the minimum number of unique bugs. It's possible that multiple bugs resulted in the same type of failure, which I would not be able to determine until I investigate the bugs further.

Any generated test that causes a failure when run directly with the interpreter is immediately thrown out. Therefore, we know that all detected bugs are problems with the compiler passes and not with Bear itself. Bad tests shouldn't be generated anyway, but there is logic to explicitly not count them to be sure the detected bug counts are accurate.

All the bugs for DCE and LICM were found extremely quickly, within the first minute of running Bear. Bear found a bug in SSA in about 30 minutes and a bug in LVN after about 4 hours. In all cases, I stopped running the fuzzer after it found its first set of bugs to minimize multiple tests exposing the same bug.

Despite testing these passes to the best of my ability when developing them, Bear was able to find bugs (as expected). I expected most bugs to be found in the more complicated passes: SSA and LICM, so I was pleasantly surprised by its ability to find issues with DCE and LVN as well.

In terms of runtime performance, for 8 populations, Bear averages at 9.885 tests generated, run, and examined, per second for a 2-stage compiler pipeline (testing a pipeline where the program is converted into SSA form and back out of SSA). This was measured by running Bear for about a minute, dividing the number of tests run by the total running time, and averaging the measured tests per second over 3 trials. These tests were run on an Ubuntu 22.04 machine with Linux kernel version 6.2.0-37. The machine has an Intel i7-13620H with 16 Gb RAM.

The performance will be heavily dependent on the performance of the compiler passes being tested. For example, running time is halved for a single-stage compiler pipeline consisting only of my LVN implementation.

REFERENCES

[1] Ian Dempsey, Michael O'Neill, and Anthony Brabazon. 2009. *Foundations in grammatical evolution for dynamic environments*. Vol. 194. Springer.

[2] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices* 53, 4 (2018), 436–449.

[3] Joel Lehman and Kenneth O Stanley. 2011. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation* 19, 2 (2011), 189–223.

[4] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1826–1847.

[5] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.