Growing A Test Corpus with Bonsai Fuzzing

Vasudev Vikram  
University of California, Berkeley  
Berkeley, CA, USA  
avasumv@berkeley.edu

Rohan Padhye  
Carnegie Mellon University  
Pittsburgh, PA, USA  
rohanpadhye@cmu.edu

Koushik Sen  
University of California, Berkeley  
Berkeley, CA, USA  
ksen@cs.berkeley.edu

Abstract—This paper presents a coverage-guided grammar-based fuzzing technique for automatically generating a corpus of concise test inputs for programs such as compilers. We walkthrough a case study of a compiler designed for education and the corresponding problem of generating meaningful test cases to provide to students. The prior state-of-the-art solution is a combination of fuzzing and test-case reduction techniques such as variants of delta-debugging. Our key insight is that instead of attempting to minimize convoluted fuzzer-generated test inputs, we can instead grow concise test inputs by construction using a form of iterative deepening. We call this approach bonsai fuzzing. Experimental results show that bonsai fuzzing can generate test corpora having inputs that are 16–45% smaller in size on average as compared to a fuzz-then-reduce approach, while achieving approximately the same code coverage and fault-detection capability.

Index Terms—test-case generation, grammar-based testing, fuzz testing, small scope hypothesis, test-case reduction

I. INTRODUCTION

This paper describes a new technique for automatically generating a concise corpus of test inputs having a well-defined syntax and non-trivial semantics (e.g., for a compiler).

This project originated when the authors were faced with the task of generating a test corpus for use in an undergraduate compilers course. The course project targets the ChocoPy programming language [1]. ChocoPy is a statically typed subset of Python, designed specifically for education. In a ChocoPy-based course, students are expected to build a compiler in Java that statically checks and then translates ChocoPy programs to RISC-V assembly. Student projects can be auto-graded by comparing their compilers’ output at various stages—parser, type checker, and code generator—with the corresponding output produced by a reference implementation. The ChocoPy project ships with a full formal specification of the language, the reference compiler, and a auto-grading infrastructure for use in such a course. However, a test suite—on which the auto-grader should run—is not provided to instructors. In this paper, we focus on the problem of automatically generating test cases that exercise the typechecker, since generating well-typed programs is known to be a difficult problem [2]–[5].

For instructors, this task presents two conflicting challenges: (1) the generated test suite must be comprehensive enough to cover various features of the language, including tricky corner cases; (2) the test suite must be concise and readable; in particular, each test case should be small in size so that test failures can guide students towards identifying which feature was incorrectly implemented. The conflict is apparent from previous work [6] which indicates that automated test generation for covering difficult program branches works better when test cases have larger size.

Much work has been done on automatically generating concise and comprehensive unit test suites [7]–[9]. However, this work mainly focuses on generating test code as sequences of method calls while minimizing the number of test cases or size of the entire test suite. Our goal is to generate non-trivial test inputs (e.g., strings) while minimizing the individual size of each test case, on average. This is because our conciseness goals are related to readability and debuggability [10], [11] rather than reducing the cost of test execution [12].

The state-of-the-art in concise automatic test case generation for structured input domains such as compilers is as follows: first, perform some form of random fuzzing [13], [14] to automatically discover unexpected or coverage-increasing inputs. Then, perform test-case reduction [15], [16] on every fuzzer-saved input in order to find a corresponding (locally) minimal input that causes the test program to exhibit the same behavior. For example, CSmith [2] and C-Reduce [17] complement each other by respectively generating and minimizing C programs for automated testing of C compilers.

By their very nature, fuzzer-generated inputs exercise program features chaotically. This can make isolating the most significant features of a fuzzer-saved test input challenging both for humans and for minimization algorithms. Further, to make the test-case minimization problem tractable, algorithms
such as delta-debugging \cite{15} and its variants perform only local optimization.

In this paper, we present bonsai fuzzing, a technique for automatically generating a concise and comprehensive test corpus of structurally complex inputs. Our key insight is that instead of reducing large convoluted inputs that exercise many program features at once, we can grow a concise test corpus \textit{bottom-up}. Bonsai fuzzing generates small inputs \textit{by construction} in an iterative evolutionary algorithm: the first round generates tiny trivial inputs and then each successive round generates inputs of slightly larger size by mutating inputs saved in a previous round. In particular, we first define a procedure to sample syntactically valid inputs from a grammar specification using bounds on the number of identifiers, linear repetitions, and nested expansions in the resulting derivation trees. We then define a partial order over coverage-guided bounded grammar fuzzers (CBGFs). For any given desired size bound, this partial order results in a lattice of CBGFs. A corpus produced by each CBGF in this lattice is used as a set of seed inputs for all successive CBGFs. The bottom element of the lattice has minimum size bound—a fuzzer with no good seed inputs—and the top element has the maximum desirable size bound—a fuzzer that produces the final test corpus. Fig. \ref{fig:example} visualizes bonsai fuzzing for a given size bound.

Experimental results on the ChocoPy typechecker indicate that bonsai fuzzing produces inputs that are 42.5\% smaller than those produced by the fuzz-then-reduce approach, while retaining 98.5\% of coverage and 98.8\% of the mutation score.

Although we developed bonsai fuzzing to solve a specific problem related to the use of ChocoPy, the technique is more generally applicable. We report results of applying bonsai fuzzing to the Google Closure Compiler, which optimizes JavaScript programs: bonsai fuzzing results in test corpora that are 16.5\% smaller on average than those produced by the conventional fuzz-then-reduce approach, while achieving approximately the same code coverage.

II. BACKGROUND AND MOTIVATION

A. ChocoPy

ChocoPy \cite{1} is a statically typed subset of Python 3.6. It uses Python’s type annotation syntax, but enforces static type checking. Figures \ref{fig:chocopy1} and \ref{fig:chocopy2} show examples of well-typed ChocoPy programs demonstrating a variety of language features borrowed from Python.

ChocoPy is primarily used in undergraduate compilers courses. For the programming assignments, students implement a Java-based ChocoPy compiler, whose output is compared against that produced by a publicly available reference compiler. Autograding is supported by the ChocoPy infrastructure out-of-the-box, though instructors are expected to create their own test suites.

In this paper, we concern ourselves with autograding the type-checking component of student-developed ChocoPy compilers: on semantically valid programs, their output is expected to match the type-annotated ASTs (in JSON format) with those produced by the reference compiler; on invalid programs, error messages and corresponding line numbers are compared. A comprehensive test suite therefore consists of both valid and invalid ChocoPy programs that exercise various features in the reference compiler. Although invalid programs are error-prone. Further, we want to have the option of quickly adding and removing language features in ChocoPy to evolve its scope. We therefore want a mechanism to automatically generate a test corpus, given only a syntax definition (i.e., a grammar) and a reference compiler implementation.

B. Problem Definition

Our high-level goal in this paper is to automatically generate test cases for the ChocoPy typechecker that are not only comprehensive but also concise. We expand on these primary goals as follows:

1) \textit{Automatic}: Manual test creation is cumbersome and error-prone. Further, we want to have the option of quickly adding and removing language features in ChocoPy to evolve its scope. We therefore want a mechanism to automatically generate a test corpus, given only a syntax definition (i.e., a grammar) and a reference compiler implementation.

2) \textit{Comprehensiveness}: We want the automatically generated test corpus to have high \textit{code coverage} and \textit{fault-detection ability}. We focus on optimizing for branch coverage in the reference compiler and also measure mutation scores where applicable.

3) \textit{Conciseness}: We want to generate minimal test cases that exercise various features in the reference compiler. We focus on optimizing for individual test-case size, though we also measure the size of the test corpus in number of test cases.

4) \textit{Semantic Validity}: We want a high fraction of semantically valid programs. Although invalid programs are necessary to cover specific aspects (e.g. error messages) of the typechecker, we prefer generally prefer test cases to be semantically valid as they are more representative examples of language features.

```python
# Fig. 2. ChocoPy Program illustrating functions, variables, and static typing. Prints True when executed.
def is_zero(items:[int], idx:int) -> bool:
    val:int = 0
    return val == 0

def equals(self: "A", y:int) -> bool:
    self.x = y
    return self.x == y

class A(object):
    x:int = 1
    def setx(self: "A", y:int):
        self.x = y

a:A = None
a = A()
if True:
    if a.equals(0):
        a.setx(3)
print(a.x)
```

```python
# Fig. 3. ChocoPy Program illustrating classes, methods, objects, and conditional statements.
class A(object):
    x:int = 1
    def setx(self: "A", y:int):
        self.x = y
    def equals(self: "A", y:int) -> bool:
        return self.x == y

a:A = None
a = A()
if True:
    if a.equals(0):
        a.setx(3)
print(a.x)
```
Finally, it only makes sense to invest in automation if our efforts can be applied to more than one testing target. We therefore also add a secondary goal:

5) General: We would like the technique to generalize to at least one other testing target.

On the surface, this seems like a standard automated testing problem. Why do we need a new technique? We next briefly discuss prior work in the context of our application goals and why we felt the need to develop a novel solution.

III. PRIOR WORK AND CHALLENGES

A. Systematic Testing

Since our goal is to generate concise test cases, a natural approach to consider is simply enumerating a bounded space of inputs or program behaviors.

1) Bounded Exhaustive Testing: Tools such as Korat [18], TestEra [19], ASTGen [20] and UDITA [21] perform bounded exhaustive testing: inputs of a bounded size are generated systematically, while employing various optimizations. These tools have been effective at generating test suites for data structure libraries, for powering automatic refactoring tools, etc. Unfortunately, the input space of a ChocoPy compiler is too large to be enumerated exhaustively. The number of unique syntactically valid programs, with at most one user-defined identifier, up to two statements per block, and a maximum block/expressive nesting depth of two, is more than the estimated number of atoms in the universe: about $10^{85}$.

2) Input Structure: Since we know the ChocoPy syntax, we can consider systematically enumerating $k$-paths [22] within the ChocoPy grammar. While this approach works well generally for testing parsers (including the ChocoPy parser), it is not well-suited for generating inputs for testing the type checking and semantic analysis logic of a compiler. For example, a minimal well-typed ChocoPy program with a method-call invocation requires several specific $k$-paths to ensure valid class definition, valid method definition, valid object instantiation, and a valid method call. Further, the cost of $k$-path exploration grows exponentially with $k$; Havrikov et al. [22] capped their evaluation of parsers at $k=5$. Minimal valid method-call invocation in ChocoPy requires $k=12$.

3) Symbolic Execution: Instead of enumerating the input space, tools such as JPF-SE [23] systematically explore the space of program paths using symbolic execution [24]. With the use of constraint solvers, one could potentially generate a comprehensive test suite that covers a diverse set of program paths of bounded size (assuming that execution path length correlates with input size). However, the number of program paths to explore grows exponentially with the number of branches encountered during execution [25]. Even on the small ChocoPy program in Fig. 2 the reference compiler executes 12,274 conditional jumps and 5,132 virtual method calls. Exhaustive symbolic execution is therefore not a practical solution even for bounded input sizes.

B. Fuzz Testing

Random test generation is an established technique for sampling complex input spaces with the hope of discovering unexpected corner cases. The term fuzz testing (or simply fuzzing) is generally used for techniques that randomly generate test inputs [13], [14], as opposed to test code [7]–[9]. Fuzzing is mainly used for discovering security vulnerabilities.

There are two main challenges in using fuzz testing tools for test corpus generation. First, generating a comprehensive test corpus for a compiler requires generating a diverse set of inputs satisfying complex constraints (e.g. programs should be well-typed). We therefore consider several variants of fuzzing that address effective state-space exploration. Second, fuzzer-generated inputs are often notoriously large and unreadable. We thus consider some advances in making test inputs concise and semantically valid.

1) Coverage-Guided Fuzzing: One extreme end, coverage-guided fuzzing (CGF) uses no knowledge of the input domain; instead, it instruments programs under test to analyze their run-time behavior. CGF evolves a corpus of test inputs with the goal of maximizing code coverage. The process starts with developer-provided or randomly generated seed inputs. New inputs are created by performing random mutations on seed inputs (e.g. randomly inserting, modifying, or deleting bytes at randomly chosen locations). Inputs that cause the test program to cover previously uncovered code are added to the set of seeds. The process repeats until a time budget expires. AFL [25] and LibFuzzer [27] are popular CGF tools for finding bugs in programs that parse binary data (e.g. media players and network protocol implementations). When applied to the ChocoPy compiler, these tools are useful for generating tests for the frontend; indeed, AFL helped discover some dormant bugs in the reference parser. However, these tools are ineffective at generating comprehensive tests for the type checker. In a preliminary experiment, we found that less than 0.01% of AFL-generated inputs were valid ChocoPy programs. This is unsurprising because random byte-level mutations rarely lead to the generation of inputs that can satisfy syntactic and semantic constraints.

2) Specialized Compiler Fuzzing: On the other extreme end, a highly precise compiler fuzzer can be developed by incorporating the syntax and semantics of the language in the input generation process itself. For example, CSmith [2] generates C programs while avoiding undefined behavior, Palka et al. [3] generate well-typed lambda terms for testing the Glasgow Haskell Compiler, and Dewey et al. [4] use constraint logic programming to test the Rust type-checker. Such specialized compiler fuzzers require quite a bit effort to develop, and do not meet our secondary criteria of being generally applicable to multiple testing targets.

3) Grammar-based Fuzzing: Between these extremes, grammar-based fuzzers offer an acceptable middle ground. Using only a declarative specification of a compiler’s input grammar—which is often readily available—these fuzzers randomly sample syntax trees. Inputs generated in this way are guaranteed to be syntactically valid. By enforcing bounds
on the expansion of recursive production rules and other repeating elements, the size of generated test inputs can also be controlled. In Section [IV-A], we provide an algorithm for sampling size-bounded test inputs from a context-free grammar provided in an extended BNF notation.

Although grammar fuzzing produces syntactically valid test inputs by construction, generating inputs that are semantically valid is challenging. For example, we empirically found that the probability of a randomly sampled ChocoPy program of size 3 (precisely defined in Section [IV-A] being semantically valid is less than 9%.

4) Semantic Fuzzing: Recently developed tools such as Zest [28], Nautilus [29], and Superion [30] combine structure-aware (e.g. grammar-based) input generators with code coverage feedback. The hope is that such feedback will help generate inputs that are not only syntactically valid, but also exercise various code paths in the compiler corresponding to semantic checks. In fact, Zest is specifically designed to generate semantically valid inputs for programs such as compilers. We therefore found Zest a very promising approach for generating a test corpus for ChocoPy.

While Zest-produced test suites were comprehensive—achieving about 95% line coverage on the ChocoPy type-checker—the generated test corpora were not concise. For example, the size-bounded Zest-generated program in Fig. 4 simultaneously achieves novel coverage related to the handling of while loops, for loops, and if-else expressions. However, the program also contains certain redundant features—those that exist in other inputs in the corpus—such as pass statements, assignments, and list indexing. This is sometimes referred to as collateral coverage in the literature [12]. We prefer not to provide such a compound input to undergraduate students developing a compiler, as (1) it does not immediately suggest an implementation goal and (2) it is not ideal for debugging failures.

C. Test-Case Reduction

A natural solution to the conciseness problem presented by Zest-generated inputs is to simply minimize them. In general, finding a minimal input that exhibits a given behavior (e.g. triggers a bug, or exercises certain program features), is an NP-hard problem. Starting with an initial input of size \( n \), there are \( O(2^n) \) possible subsets of the starting input itself, not to mention other small inputs that contain elements not present in the initial input.

Techniques such as Delta Debugging (DD) [15] find locally minimal inputs that are subsets of the initial input in worst-case \( O(n^2) \) steps. One drawback of DD applied on the string representation of inputs is that deleting individual characters and contiguous substrings often results in inputs that have invalid syntax; therefore, most subsets do not exhibit the desired behavior. Hierarchical Delta-Debugging (HDD) [16] solves this problem by applying a DD-like algorithm on a tree representation of parsed inputs. HDD requires knowledge of the input syntax, which is readily available in our application.

```plaintext
while not {}:
    for a in a:
        b and True
    (0..a = (c)[None if c else 1 if a else ""]
    pass

Fig. 4. ChocoPy Program generated using coverage-guided bounded grammar-based fuzzing with size bounds of (3, 3, 3).

while A:
    for A in "":
        A= None if A if None else A else A

Fig. 5. Minimized ChocoPy program achieving the same novel coverage as achieved by the program in Fig. 4
```

We used state-of-the-art implementations of DD and HDD developed by Hodovan et al. [31]–[34] on Zest-generated ChocoPy programs. Fig. 5 depicts a minimized version of the program listed in Fig. 4 where the reduction criterion was that the reduced input achieves at least the unique same coverage as achieved by the original input. The minimization takes about 30 seconds to run, and achieves a 50% reduction in test case size—the redundant pass, assignment, etc. has been removed. However, Fig. 5 still contains multiple loops, branching statements, etc. In the next section, we will describe a novel solution that produces inputs that are much more concise, for free.

IV. BONSAI FUZZING

Our proposed technique leverages the scalability advantages of grammar-based coverage-guided fuzzing while avoiding the constraints of the fuzz-then-reduce approach. The key idea in our approach is to grow a test corpus bottom-up by (1) using coverage-guided bounded grammar fuzzing (CBGF) to generate small inputs by construction and (2) iteratively increasing the input size, inspired by iterative-deepening-based search algorithms [15]. We call our approach bonsai fuzzing.

Figs. 6, 7, and 8 show a total of ten ChocoPy programs saved during various rounds of bonsai fuzzing (comments added manually). These programs are concise and the language features they exercise can be easily discerned. In our opinion, they look almost like hand-written test cases that are precisely designed for testing specific features of the ChocoPy language semantics. However, they were generated completely automatically and without knowledge of any typing rules. We next build a series of concepts leading up to a description of the bonsai fuzzing algorithm.

A. Bounded Grammar Fuzzers

We start by considering an input generator that can randomly sample inputs of a bounded size, where the bounds are based on the definition of an input language’s grammar. We can observe three properties of a ChocoPy program to get an idea of how we might bound the input space.

1) \textit{idents}: the number of new unique identifiers (variable names, function names, class names) excluding predefined identifiers (e.g. \texttt{int}).
# (Ex. A) Single pass statement
pass

# (Ex. B) Simple assignment statement
a:object = 1

# (Ex. C) Function definition with return
def a():
    return

Fig. 6. Three examples of ChocoPy programs saved during bonsai fuzzing, in a corpus produced by $F_{1,1,1}$.

# (Ex. D) Class definition with attribute declaration
class a():
    a:int = 1

# (Ex. E) Less-than comparison on two integers
0 < 0

# (Ex. F) Equality comparison on two strings
"a" == "a"

# (Ex. G) Four examples of ChocoPy programs saved during bonsai fuzzing by def a(b:str, a:int):
# (Ex. G) Function definition with two arguments
def a(b:str, a:int):
    pass

Fig. 7. Four examples of ChocoPy programs saved during bonsai fuzzing by $F_{1,2,1}$, $F_{1,2,1}$, and $F_{1,1,2}$.

2) items: the maximum number of elements in a linear group. This can correspond to the maximum number of statements in a block, arguments in a function definition, arguments in a list expression, etc.
3) depth: the maximum number of times an expression, statement, or function definition is nested.

For the ChocoPy example in Fig. 2 we have $iden = 4$ (is_zero, items, idx, val), $items = 5$ (comma-separated list elements on line 6), $depth = 3$ (triply nested expressions on line 6).

For the ChocoPy example in Fig. 3 we have $ids = 7$ (a, setx, equals, self, x, y, a), $items = 4$ (top-level statements in the program), $depth = 2$ (doubly nested if statements on lines 9–11).

We can bound the input space if we restrict the maximum value of $ids$, $items$, and $depth$ for any generated ChocoPy program. We will now generalize this to any language.

Consider a specification for the syntax of an input language in the form of a context-free grammar $G$. We consider definitions in an extended Backus–Naur form [14], where $G$ consists of a set of terminals $T$, a set of non-terminals $N$, a start symbol $S \in N$, and a set of production rules of the form

$$A \rightarrow \alpha, \text{ where } A \in N \text{ and } \alpha = a_1 a_2 \ldots$$

The right-hand side of production rules $\alpha$ are a sequence of zero or more symbols which are defined recursively as follows: a symbol is either a terminal in $T$, a non-terminal in $N$ or of the form $[b]^*$, where $b$ is a symbol. The Kleene-star in the final form has the usual meaning and enables non-recursive definitions of linear repeating sequences, e.g. list of statements or arguments to a function call. We also consider a special class of terminals $\tau \subseteq T$ whose concrete values are user-defined (e.g. identifiers) instead of predefined (e.g. ‘+’ or ‘while’). We attach the ChocoPy grammar as supplementary material for reference, where $\tau = \{ID, type\}$.

Now consider the set of programs $P = \{p : p \sim G\}$. Each program $p$ has a corresponding derivation tree $t$ from $G$. We are interested in bounding the following properties:

1) $ids(p)$: The maximum number of distinct values for any terminal in $\tau$ (e.g. number of distinct identifiers) observed across the entire tree $t$.
2) $items(p)$: The maximum number of repetitions in any expansion of a kleene-star (e.g. number of statements in a block) when generating $t$.
3) $depth(p)$: The maximum number of expansions of the same non-terminal (e.g. $expr$) in any path from the root to any leaf node in $t$.

We can then define a smaller input space $P_{m,n,d}$, where

$$P_{m,n,d} = \left\{ p \in P : \begin{array}{l}
        \text{ids}(p) \leq m, \\
        \text{items}(p) \leq n, \\
        \text{depth}(p) \leq d
    \end{array} \right\}$$

For example, the ChocoPy program in Fig. 1 belongs to ChocoPy$_{4,5,3}$, but the program in Fig. 3 does not. Both of them belong to ChocoPy$_{7,5,3}$. Neither is in ChocoPy$_{1,1,1}$.

Algorithm 1 details the procedure we use for sampling programs in $P_{m,n,d}$. The parameters to function BOUNDED-SAMPLE are a grammar $G$, a symbol $a$, and bounds $m, n, d$; the function returns a string which is an expansion of symbol $a$ that obeys the provided bounds. A top-level call to BOUNDED-SAMPLE with $a = S$, the start symbol of the grammar, produces a random program in $P_{m,n,d}$.

The sampling algorithm has a similar structure to the PCT1 grammar-sampling procedure described by Luke [37]: the following discussion clarifies specific algorithmic details.
Algorithm 1 Bounded grammar sampling algorithm.
$\mathcal{G}$ is a grammar, $m$, $n$, and $d$ are positive integers.

function BOUNDEDSAMPLE($\mathcal{G}$, symbol $a$, $m$, $n$, $d$) returns

  case typeof($a$):
    terminal $t$: return CONCRETIZE($t$, $m$) // See text...

  repetition $[b^*]$: return concatenate($[
  \text{BOUNDEDSAMPLE}(b, m, n, d) \text{ for } i \in \{0 \ldots \text{chooseRandom}([0 \ldots n])\}]$)

  nonterminal $A$: return SAMPLENONTERMINAL($\mathcal{G}$, $A$, $m$, $n$, $d$)

function SAMPLENONTERMINAL($\mathcal{G}$, nonterminal $A$, $m$, $n$, $d$) returns

  if $|\text{NT_EXPANSIONS}($$\mathcal{G}$, $A$)$| == 0$ then
    $p \leftarrow 1$ // Expand to leaf node
  else if $|\text{T_EXPANSIONS}($$\mathcal{G}$, $A$)$| == 0$ then
    $p \leftarrow 0$ // Expand to non-leaf node
  else
    Let $c \leftarrow \text{number of expansions of } A \text{ from root to here}$
    $p \leftarrow (c + 1)/(d + 1)$ // Probability of leaf expansion

  with probability $p$
    Let $A \rightarrow \alpha = \text{chooseRandom}(\text{T_EXPANSIONS}(A))$
  otherwise
    Let $A \rightarrow \alpha = \text{chooseRandom}(\text{NT_EXPANSIONS}(A))$

  return concatenate($\text{SAMPLE}(b, m, n, d)$ for $b$ in $\alpha$)

function T_EXPANSIONS($\mathcal{G}$, nonterminal $A$) returns

  all expansions $A \rightarrow \alpha$ in $\mathcal{G}$ where
  $\forall a_i \in \alpha$, typeof($a_i$) $== \text{terminal}$

function NT_EXPANSIONS($\mathcal{G}$, nonterminal $A$) returns

  all expansions $A \rightarrow \alpha$ in $\mathcal{G}$ where
  $\exists a_i \in \alpha$, typeof($a_i$) $== \text{nonterminal}$

In general, since $a$ can be any type of symbol—terminal, nonterminal, or a group with kleene-star—BOUNDEDSAMPLE performs different logic depending on the type of $a$.

1) When $a$ is a terminal symbol, it is concretized as follows: If $a \in \tau$, then one of $m$ pre-populated expansions is uniformly chosen at random (e.g. if the terminal represents an identifier, then one of say $a_1$, $a_2$, $\ldots$, $a_m$ is returned uniformly at random). Otherwise, $a$ has exactly one concrete value (e.g. ‘+’ or ‘\texttt{while}’), which is returned directly.

2) If $a$ is a repetition $[b]^*$, we choose a number of expansions $i$ uniformly at random in the range $[0, n]$. Then, we recursively call BOUNDEDSAMPLE with symbol $b$ for $i$ times and the results are concatenated.

3) If $a$ is a nonterminal $A$, then with a calculated probability $p$ we return the output of BOUNDEDSAMPLE on a randomly chosen terminal expansion. Otherwise, we use a randomly chosen nonterminal expansion. The probability $p$ is a function of the number of times $A$ has been expanded from the root and the maximum depth parameter $d$. It ensures that the program cannot have a depth larger than $d$, while favoring nonterminal expansions when the nesting depth is relatively smaller.

### Preliminary Results with ChocoPy:

Now, given that we have a bounded grammar sampling procedure, what bounds $(m, n, d)$ do we choose to achieve our goals? There is a natural dichotomy between conciseness and comprehensiveness. Tiny bounds such as $(1, 1, 1)$ produce very concise inputs, but they do not exercise many language features. Additionally, most randomly sampled inputs of size $(1, 1, 1)$ are well-typed; as the bounds increase, the likelihood of a randomly sampled program being semantically valid diminish.

To find good bounds, we ran small 3-hour fuzzing sessions using bounded grammar sampling for all configurations where $m$, $n$, and $d$ were between 1 and 5 each—a total of 125 configurations. Each experiment was repeated ten times to account for randomness. We then measured (1) branch coverage in the ChocoPy reference typechecker across all the inputs generated during each experiment, and (2) fraction of generated inputs that were semantically valid. Fig. 9 shows averages of the fraction validity and relative branch coverage for a slice of these experiments. Overall, we found the bounds $(3, 3, 3)$ to be a good trade-off between conciseness and comprehensiveness, although the fraction of valid inputs generated was still concerning (only 9%). We next consider a feedback-directed variant of the bounded grammar sampling fuzzer that can produce inputs that are more likely to be semantically valid.

### B. Coverage-Guided Bounded Grammar Fuzzing (CBGF)

In order to incorporate a feedback from test execution, we enhance our bounded grammar sampling technique to a coverage-guided bounded grammar fuzzer (CBGF). Algorithm 2 describes CBGF. It is almost a standard coverage-guided fuzzing loop (e.g. as described by Böhme et al. [38]), but focuses on generating a comprehensive test-case corpus rather than discovering program crashes. The technique expects an instrumented version of the test program, such as the ChocoPy reference compiler; the instrumentation provides a way to receive feedback (e.g. code coverage) from test execution. Test execution on a given input can also return additional feedback such as whether the input was semantically valid or not (e.g. based on whether type-checking succeeded).
or if there were any errors). The function CBGF is given an ordered set of initial seed inputs in $S$. The main fuzzing loop continuously cycles through the set $S$, picking each input in order (sometimes with repetition to increase energy [28]), mutating it, and executing the test program with the mutated input to receive feedback. If the feedback is interesting (e.g., coverage includes a program location that is not exercised by any other input in $S$ so far), then the mutated input is added to $S$. The loop ends after a fixed time budget, and the resulting corpus of inputs $S$ is returned.

The two main unspecified components in this algorithm are how Mutate works (Line 4) and what the interestingness criteria is for saving new inputs (Line 6). We use an off-the-shelf implementation of Zest [28], a structure-aware coverage-guided fuzzer that is well suited for our application. In Zest, all inputs—including the initial seed inputs—are generated using some sampling procedure called a generator; in our case, the generator is simply the bounded grammar sampler (ref. Algorithm 1). Each input is associated with a sequence of pseudo-random choices made during the sampling procedure that uniquely determine the input produced by that procedure. In Algorithm 1 this includes the “random” choices made in expanding production rules and concretizing terminal values. The Mutate function in Algorithm 2 randomly mutates the pseudo-random choices that correspond to input and then replays BoundedSample with the specified choices and with the given bounds. We expect the returned value $input'$ to be a syntactically valid input that is subtly different from—that is, a structural mutation of—original input $input$. Note that if $input$ was a member of the initial set of seeds, then the size bounds $(m, n, d)$ provided to Mutate may be larger than the bounds used to originally generate $input$; we will exploit this fact in the Section IV-D.

The criteria used by Zest to determine whether to save $input'$ (Line 6 in Algorithm 2) is the following: the feedback from execution of $p$ on $input'$ is interesting if (1) there is new code coverage, regardless of the validity of $input'$, or (2) $input'$ is semantically valid and it achieves new coverage when compared to all other semantically valid inputs in $S$. Zest thus favors saving semantically valid inputs. Section IV-D describes a tweak to this criterion we make in some scenarios.

The $F$ notation: We now define some short-hand notation that will be useful when describing our proposed bonsai fuzzing technique. Let $F_{m,n,d}$ denote a coverage-guided bounded grammar fuzzer (CBGF) parameterized by grammar $G$, test program $p$, size bounds $m$, $n$, and $d$. As per Algorithm 2, $F_{m,n,d}$ is a function that accepts an ordered set of inputs and returns a corpus of the same type. Since the grammar and target program are usually fixed in a given application, we will omit the superscripts hereon; therefore, $F_{m,n,d}$ is a CBGF of size bounds $(m, n, d)$.

Preliminary Results with ChocoPy: As described in Section II-C, simply using Zest followed by input minimization on the resulting corpus still lacks conciseness. The program in Fig. 4 was produced using $F_{3,3,3}$ in seedless mode [39]. The program in Fig. 5 is its corresponding reduction after applying hierarchical delta debugging [16]—the invariant being that the reduced input still meets the same interestingness criteria from Algorithm 2. We attach the full corpus of a reduced (using HDD) corpus for ChocoPy in the supplementary material.

C. Bonsai Fuzzing

Our novel solution is to build a concise test corpus from the bottom up by using a set of CBGFs with gradually increasing size bounds. The intuition is that the smaller CBGFs would initially build a corpus of tiny test corpus covering simple features, and larger CBGFs can build on the smaller programs to generate more complex test cases that achieve better coverage. By increasing the size bounds gradually at each step, we expect the complex test cases in later stages to be structural mutations of test cases discovered in earlier stages; thus, we hope to simultaneously achieve validity, conciseness, and comprehensiveness. We now define a way to iteratively increment the size of a CBGF, which allows us to create a formal procedure for this approach.

With upper bounds $(M, N, D)$, we can consider the set of CBGFs

$$C_{M,N,D} = \left\{ F_{m,n,d} : \begin{array}{l} 1 \leq m \leq M \\ 1 \leq n \leq N \\ 1 \leq d \leq D \end{array} \right\}$$

With upper bounds $(3, 3, 3)$, we would have 27 different CBGFs in set $C_{3,3,3}$.

We define a partial order $\leq$ over $C_{M,N,D}$ as follows:

$$F_{m,n,d} \leq F_{m',n',d'} \iff \begin{cases} m = m' & n \leq n', d \leq d' \\ m \leq m' & n = n', d \leq d' \\ m \leq m', n = n', d = d' \end{cases}$$

Consequently, $F_{m,n,d} \leq F_{m',n',d'}$ iff $F_{m,n,d} \leq F_{m',n',d'}$. This ordering suggests that $C_{M,N,D}$ is a lattice with $F_{1,1,1}$ being the bottom element (denoted $F_L$) and $F_{M,N,D}$ being the top element (denoted $F_T$). Fig. 1 visualizes the lattice for $C_{2,2,2}$, where the partial order corresponds to graph reachability. In this example, $F_T = F_{2,2,2}$.

Additionally, we define the terms successor and predecessor with their usual meaning:

1) $F_s$ is a successor of $F$ if $F < F_s$ and there exists no CBGF $F'_s$ such that $F < F'_s < F_s$. 

### Algorithm 2 Coverage-Guided Bounded Grammar Fuzzing

**Require:** Instrumented program $p$, Grammar $G$, Bounds $m$, $n$, $d$

1: function CBGF(Seed inputs $S$) $\triangleright$ Returns corpus $S$ 
2: repeat
3:     input $\leftarrow$ next($S$) $\triangleright$ Cycle through $S$
4:     input $'$ $\leftarrow$ Mutate(input, $G$, $m$, $n$, $d$) $\triangleright$ See text...
5:     feedback $\leftarrow$ Execute($p$, input) $'$ $\triangleright$ validity + coverage
6:     if feedback is interesting then $\triangleright$ new coverage? 
7:         $S$ $\leftarrow$ $S$ $\cup$ input $'$
8: until time budget expires
9: return $S$

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>function CBGF(Seed inputs $S$) $\triangleright$ Returns corpus $S$</td>
</tr>
<tr>
<td>2</td>
<td>repeat</td>
</tr>
<tr>
<td>3</td>
<td>input $\leftarrow$ next($S$) $\triangleright$ Cycle through $S$</td>
</tr>
<tr>
<td>4</td>
<td>input $'$ $\leftarrow$ Mutate(input, $G$, $m$, $n$, $d$) $\triangleright$ See text...</td>
</tr>
<tr>
<td>5</td>
<td>feedback $\leftarrow$ Execute($p$, input) $'$ $\triangleright$ validity + coverage</td>
</tr>
<tr>
<td>6</td>
<td>if feedback is interesting then $\triangleright$ new coverage?</td>
</tr>
<tr>
<td>7</td>
<td>$S$ $\leftarrow$ $S$ $\cup$ input $'$</td>
</tr>
<tr>
<td>8</td>
<td>until time budget expires</td>
</tr>
<tr>
<td>9</td>
<td>return $S$</td>
</tr>
</tbody>
</table>
Algorithm 3 Bonsai fuzzing algorithm

1: procedure BONSAIFUZZING
2: \( \mathcal{F} \leftarrow \mathcal{F}_u \)
3: seeds \( \leftarrow \text{[random()]} \) \quad \triangleright \quad \text{Single random seed}
4: corpus(\( \mathcal{F}_u \)) \( = \) \( \mathcal{F} \)\( (\text{seeds}) \) \quad \triangleright \quad \text{Run CBGF to generate corpus}
5: worklist \( \leftarrow \text{successors}(\mathcal{F}) \)
6: while worklist is not empty do
7: \( \quad \text{for each } \mathcal{F} \text{ in worklist do} \quad \triangleright \quad \text{Parallelizable}
8: \quad P \leftarrow \text{predecessors}(\mathcal{F})
9: \quad seeds \leftarrow \text{SortBySize} \left( \bigcup_{\mathcal{F}_p \in P} \text{corpus}(\mathcal{F}_p) \right) \quad \triangleright \quad \text{Run CBGF}
10: \text{corpus}(\mathcal{F}) \leftarrow \mathcal{F} \)\( (\text{seeds}) \)
11: \text{worklist} \leftarrow \bigcup_{\mathcal{F}_s \in \text{worklist}} \text{successors}(\mathcal{F}_s)
12: \text{return corpus}(\mathcal{F}^\top)

Algorithm 3 describes the procedure for bonsai fuzzing. The hope is that invalid inputs that are generated by mutating the interestingness criterion on Line 6 of Algorithm 2. First, a restricted-CBGF is a CBGF that only saves valid inputs: that is, the feedback is considered interesting on Line 6 if the input was valid and it achieved new code coverage. Second, an unrestricted-CBGF is one that saves both valid and invalid inputs, using the standard interestingness criterion described in Section IV-B.

We evaluate bonsai fuzzing by measuring its ability to generate a test corpus containing test cases that are concise, comprehensive, semantically valid, and (where applicable) able to detect faults. We compare bonsai fuzzing to a baseline of CBGF (that is; Zest [28] with a grammar-based input generator) post-processed with minimization techniques. The baseline is thus the conventional “fuzz-then-reduce” approach.
We run our evaluation on two test targets: our primary application and a secondary target to ensure that our solution is not biased towards a particular implementation or input language.

1) ChocoPy [40]: The test driver reads in a ChocoPy program and runs the semantic analysis / type-checking stage of the ChocoPy reference compiler. For the fault-detection evaluation, we additionally run a differential test on the typed ASTs returned by a reference and buggy compiler (see Section V-D).

2) Google Closure Compiler [41]: The test driver expects a JavaScript program as input and performs source-to-source optimizations. Refer to the benchmark in [28].

**Experimental Setup:**

1) **Bound:** We set \((M = N = D = 3)\) as the bounds for bonsai fuzzing as well as the baseline CBGF bounds.

2) **Duration:** We run each CBGF node in the bonsai fuzzing extended lattice for one hour, which totals 54 hours of CPU time. We allocate the same 54 hours of CPU time for the baseline CBGF to run.

3) **Repetition:** We run each experiment 10 times and report metrics across all repetitions due to the nature of randomness in fuzzing and its effect on results.

**Minimization Techniques:** For the fuzz-then-reduce baseline, we use Picire [42] and Picireny [43], which are state-of-the-art [31–34] implementations of character-level [15] and grammar-based hierarchical [16] delta debugging respectively.

An “interestingness” test that could be used for minimization was required for each of these tools. We chose to use an oracle that tests whether the minimized input program met the same criterion as was used to save the original input during CBGF (ref. Line 6 in Algorithm 2). Table I lists the average CPU-time for each of these reduction tools to minimize an entire corpus.

**A. Conciseness: Test Corpus Size**

We evaluate conciseness by measuring the size of each test file—excluding whitespace characters—in the output corpus. Fig. 10 displays the distribution of test input sizes for the baseline and bonsai fuzzing.

On both targets, we observed that bonsai fuzzing produces test files that are statistically significantly lower in size than those of the baseline. The ChocoPy files are on average 42.22% smaller than the results of grammar-based reduction and 44.51% smaller than the results of character-based reduction. The Closure files are on average 16.49% smaller than the results of grammar-based reduction and 25.56% smaller than the results of character-based reduction. We also see that the variance of the size of files in the violin plot of bonsai fuzzing is much lower than that of the baseline. One clear advantage is that bonsai fuzzing is able to produce these smaller inputs without requiring any additional post-processing time. In contrast, the fuzz-then-reduce approach of the baseline can take up to 6 hours for minimization to run.

As a sanity check, we also report the number of files in the test corpora as shown in Table II. The resulting corpus from bonsai fuzzing contains about 18% fewer files in both targets. This shows that bonsai fuzzing does not compensate for its smaller test inputs by having a large number of tests.

**B. Semantic Validity**

The average percent of semantically valid programs in the output corpora is shown in Fig. 11. Bonsai fuzzing has a statistically significant increase in both targets. On average, it
is able to achieve a 21% improvement in validity in ChocoPy and a 7% improvement in Closure. We value this improvement in validity since it means that more language features are being covered by test cases that are semantically valid, which in our opinion results in more meaningful and readable test cases.

C. Comprehensiveness: Coverage

A key concern when generating small inputs by construction is whether they comprehensively exercise various program behaviors as conventional coverage-guided fuzzing.

We measure coverage using a third-party tool: the widely used JaCoCo library [44]. We report the branch coverage on the semantic analysis classes within each of the benchmarks, similar to approach in [28]. Since many of the branches are unreachable from our test drivers, it is important to focus on the relative difference between the baseline and bonsai fuzzing rather than the raw coverage values.

Fig. [12] shows the branch coverage achieved by the baseline and bonsai fuzzing on each of the targets. We can see that both techniques achieve approximately the same branch coverage. On Closure, the difference is statistically insignificant. On ChocoPy, the difference is significant but its effect is small: bonsai fuzzing loses 1.175% of branch coverage on average. We are not dismayed with this small reduction. In our application, we can easily incorporate the few test cases from conventional fuzzing that cover logic that is not exercised by bonsai fuzzing—in ChocoPy, this is usually just one test case.

D. Fault Detection: Mutation Scores

Finally, we want to ensure that the concise inputs generated by bonsai fuzzing for the ChocoPy target can still catch faults; that is, they can be used for automated grading or providing student feedback. This is essentially the small scope hypothesis [45]. In a classroom setting, we would compare a candidate buggy student implementation with the reference implementation. For our experimental evaluation, we simulate such a buggy candidate by using a mutation testing tool [46] on a copy of the reference compiler. We run the ChocoPy autograder on the reference compiler and its mutation; if the auto-grader detects a failure, then the mutation is killed.

The test corpus produced by bonsai fuzzing achieves a mutation-killing score of 81% on average. This not surprising, since the test corpus is optimized for coverage within the reference compiler, and is not aware of student implementations or mutations. As recently observed by Chen et al. [47], the best technique for increasing fault detection while minimizing test sizes is to first optimize for coverage and then optimize for mutation scores when coverage saturates. We thus use the corpus produced by bonsai fuzzing (and the baseline, for comparison) as seed inputs for a simple grammar-based blackbox fuzzer with the maximum bounds (3, 3, 3) for 30 minutes. We do this for each of the 444 mutated compilers—this is, simulated buggy candidates. If any blackbox -fuzzer-generated input kills the mutation, we say that the corresponding technique kills that mutation.

Table III summarizes these results. Both the baseline and bonsai fuzzing achieves more than 90% mutation-killing score, which is quite acceptable. We therefore conclude that size-bounded fuzzing does not significantly sacrifice fault detection capability on ChocoPy. Unfortunately, we cannot report meaningful mutation scores on Closure since the project does not have a proper testing oracle beyond catching program crashes.

VI. DISCUSSION AND THREATS TO VALIDITY

We chose to run each CBGF node in the bonsai fuzzing lattice for one hour as an estimation for how long it would take for the coverage to saturate. Another idea would be to keep each node running without a time constraint until the coverage has explicitly saturated (i.e. no more new inputs are being saved). An additional area of exploration is choosing the final bound for bonsai fuzzing; our main motivation for choosing the bounds of (M = N = D = 3) was that we felt it achieved a fair compromise between size and coverage. We did not empirically evaluate how the baseline and bonsai fuzzing performed using various bounds, but we feel that the bounds of (3, 3, 3) should be at least sufficient to cover all the ChocoPy features we would like to test.

In this paper, we use the notion of conciseness as a proxy for readability based on what we feel are important features
of readable test cases (size and semantic validity). There are additional ways to evaluate how readable the bonsai fuzzing corpus is, such as surveying and feedback.

We were unfortunately unable to test fault detection capabilities of bonsai fuzzing on actual student implementations, since we do not have access to them. We hope that mutation scores are an indication of the ability of bonsai fuzzing to catch real student bugs in the future.

Overall, bonsai fuzzing highlights the potential to automatically generate concise and comprehensive test cases by growing the test corpus from the bottom-up.

ACKNOWLEDGMENTS

The experimental evaluation for this paper was supported by the Amazon AWS Cloud Credits for Research program.

REFERENCES


