Growing A Test Corpus with Bonsai Fuzzing

Vasudev Vikram
University of California, Berkeley
Berkeley, CA, USA
vasumv@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 synthesizing a corpus of concise test inputs. We walk-through 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 autograded by comparing their compilers' output at various stages—parser, type checker, and code generator—with the corresponding output produced by a reference implementation. When starting their project, students are provided with a suite of ChocoPy test programs and the autograder, which together serve as a partial executable specification, simulating *test-driven development*, while also enabling students to continuously get feedback about their progress. For instructors, writing test cases to validate every language feature is a tedious task; we wanted to *automatically* synthesize such a test corpus. This paper describes the technique we developed for this purpose. In particular, 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].

This task presents two conflicting challenges: (1) the generated test suite must be *comprehensive* in covering various semantics of the language, including 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 with larger test cases.

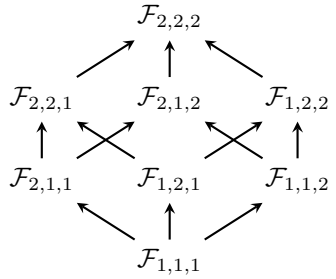

Fig. 1. An example bonsai fuzzing architecture—A lattice of coverage-guided size-bounded grammar-based fuzzers $\mathcal{F}_{m,n,d}$, ordered by three size bounds on the syntax of the test cases they produce: number of unique identifiers m, maximum sequence length n, and maximum nesting depth d. Test cases flow along directed edges: the inputs generated by each fuzzer are used as the seed inputs to its successors. The result of bonsai fuzzing is the corpus produced by the top-most element.

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 [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 *bottom-up*. Bonsai fuzzing generates small inputs *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. 1 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. We have made a replication package publicly available at https://github.com/vasumv/bonsai-fuzzing.

II. Background and Motivation

A. ChocoPy

ChocoPy [1] is a statically typed subset of Python 3.6. It uses Python's type annotation syntax, but enforces static type checking. Figures 2 and 3 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. In this paper, we are interested in specifying and autograding the type-checking component of student-developed ChocoPy compilers: on semantically valid

```
def is_zero(items:[int], idx:int) -> bool:
    val:int = 0
    val = items[idx]
    return val == 0
idx:int = 1
print(is_zero([1, 0, 1, 0, 1], idx))
```

Fig. 2. ChocoPy Program illustrating functions, variables, and static typing. Prints `True` when executed.

```
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)
```

Fig. 3. ChocoPy Program illustrating classes, methods, objects, and conditional statements.

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 aspects of the ChocoPy typing rules.

B. Problem Definition

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

1) *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) *Comprehensiveness*: We want the automatically generated test corpus to have high *code coverage* and *fault-detection ability*. We focus on optimizing for branch coverage in the reference compiler and also measure mutation scores where applicable.
3) *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) *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.

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) *Generalizable*: 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/expression 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. This approach yields minimal programs corresponding to each unique k-length path (from root to leaf) in valid syntax trees. This works really well for generating parser tests; however, it is not ideal for exercising the type checking and semantic analysis logic of a compiler. For example, a minimal well-typed ChocoPy program that contains a valid method-call invocation requires several syntax subtrees to ensure valid class definition, valid method definition inside the class, valid instantiation of an object of that class, and a valid method call on this object. This semantic feature cannot therefore be defined as a linear k-path for any k.

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 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 [26] 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, Pałka 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 $\mathcal{O}(2^n)$ 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 $\mathcal{O}(n^2)$ 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. Similarly, Perses [31] uses a grammar to perform reductions, while guaranteeing that each reduction step also produces a syntactically valid program.

```
while not (0):
    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. [32]–[35] 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 [36]. We call our approach *bonsai fuzzing*.

Figs. 6, 7, and 8 show a total of eleven 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.

```
# (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 $\mathcal{F}_{1,1,1}$.

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

# (Ex. E) Indexing into a string
("a")[0]

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

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

# (Ex. H) Function definition with two arguments
def a(b:str, a:int):
    pass
```

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

1) *idents*: the number of new unique identifiers (variable names, function names, class names) excluding predefined identifiers (e.g. `int`).
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 *idents*=4 (`is_zero, items, idx, val`), *items*=5 (comma-separated list elements on line 6), and *depth*=3 (triply nested expressions on line 6). For the example in Fig. 3, we have *idents*=7 (`A, setx, equals, self, x, y, a`), *items*=4 (top-level statements in the program), and *depth*=2 (doubly nested `if` statements on lines 9–11).

We can bound the input space if we restrict the maximum value of *idents*, *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 $\mathcal{G}$. We consider definitions in an extended Backus–Naur form [37], where $\mathcal{G}$ consists of a set of terminals $\mathcal{T}$, a set of non-terminals $\mathcal{N}$, a start symbol $S \in \mathcal{N}$, and a set of production rules of the form:

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

```
# (Ex. I) Nested list expression
[[1], [None]]

# (Ex. J) Object construction and attribute assignment
class a(object):
    a:int = 1
(a()).a = 1

# (Ex. K) Nested functions
def a():
    def b():
        pass
    return
```

Fig. 8. Example ChocoPy programs saved in the bonsai fuzzing corpus of $\mathcal{F}_{3,3,3}$.

The right-hand side of production rules α are a sequence of zero or more *symbols* which are defined recursively as follows: a *symbol* is either a terminal in $\mathcal{T}$, a non-terminal in $\mathcal{N}$ 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 $\tau \subseteq \mathcal{T}$ whose concrete values are user-defined (e.g. identifiers) instead of predefined (e.g. '+' or '`while`'). In the ChocoPy grammar—included in our online repository (ref. Section I)—we have $\tau = \{ID, IDSTRING\}$.

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

1) *idents*(p): The maximum number of distinct values for any terminal in τ (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 $\mathcal{P}_{m,n,d}$, where

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

For example, the ChocoPy program in Fig. 2 belongs to $\text{ChocoPy}_{4,5,3}$, but the program in Fig. 3 does not. Both of them belong to $\text{ChocoPy}_{7,5,3}$. Neither is in $\text{ChocoPy}_{1,1,1}$.

Algorithm 1 details the procedure we use for sampling programs in $\mathcal{P}_{m,n,d}$. The parameters to function BOUNDED-SAMPLE are a grammar $\mathcal{G}$, 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 BOUNDEDSAMPLE with $a = S$, the start symbol of the grammar, produces a random program in $\mathcal{P}_{m,n,d}$.

Algorithm 1 Bounded grammar sampling algorithm.
$\mathcal{G}$ is a grammar; m, n, and d are positive integers.

```
function BoundedSample(G, symbol a, m, n, d)
    case typeof(a):
      terminal t: return Concretize(t, m)                ▷ See text...
      repetition [b]*: return concatenate(
                    [BoundedSample(b, m, n, d)
                      for i ∈ {0 ... chooseRandom([0 ... n])}])
      nonterminal A: return
                        sampleNonTerminal(G, A, m, n, d)
function sampleNonTerminal(G, nonterminal A, m, n, d)
    if |NT_Expansions(G, A) |== 0 then
        p ← 1                                            ▷ Expand to leaf node
    else if |T_Expansions(G, A) |== 0 then
        p ← 0                                            ▷ Expand to non-leaf node
    else
        Let c ← number of expansions of A from root to here
        p ← (c + 1)/(d + 1)                              ▷ Probability of leaf expansion
    with probability p
        Let A → α = chooseRandom(T_Expansions(A))
    otherwise
        Let A → α = chooseRandom(NT_Expansions(A))
    return concatenate([sample(b, m, n, d) for b in α]
function T_Expansions(G, nonterminal A)
    return all expansions A → α in G where
                    ∀a_i ∈ α, typeof(a_i) == terminal
function NT_Expansions(G, nonterminal A)
    return all expansions A → α in G where
                    ∃a_i ∈ α : typeof(a_i) == nonterminal
```

The sampling algorithm has a similar structure to the PTC1 grammar-sampling procedure described by Luke [38]; the following discussion clarifies specific algorithmic details.

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`, ..., `a_m` is returned uniformly at random). Otherwise, a has exactly one concrete value (e.g. '+' or '`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. The calculation of p differs from that used by Luke in PTC1 [38], since we are interested in bounding the maximum nesting along any given path in a derivation tree, instead of bounding the size of the tree itself.

Preliminary Results with ChocoPy: 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.

For preliminary experiments, 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 all 125 configuration. We noticed that bounds such as $(3, 3, 3)$ were able to achieve high coverage; however, the fraction of valid inputs generated for was 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. [39]), but focuses on generating a comprehensive test-case corpus rather than discovering program crashes[1]. 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 $\mathcal{S}$. The main fuzzing loop continuously cycles through the set $\mathcal{S}$, picking each input in order (sometimes with repetition to increase energy [39]), 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 $\mathcal{S}$ so far), then the mutated input is added to $\mathcal{S}$. The loop ends after a fixed time budget, and the resulting corpus of inputs $\mathcal{S}$ 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

[1]We assume that the reference program being analyzed is not buggy. If we find any crashes, we apply a patch and restart from the beginning.

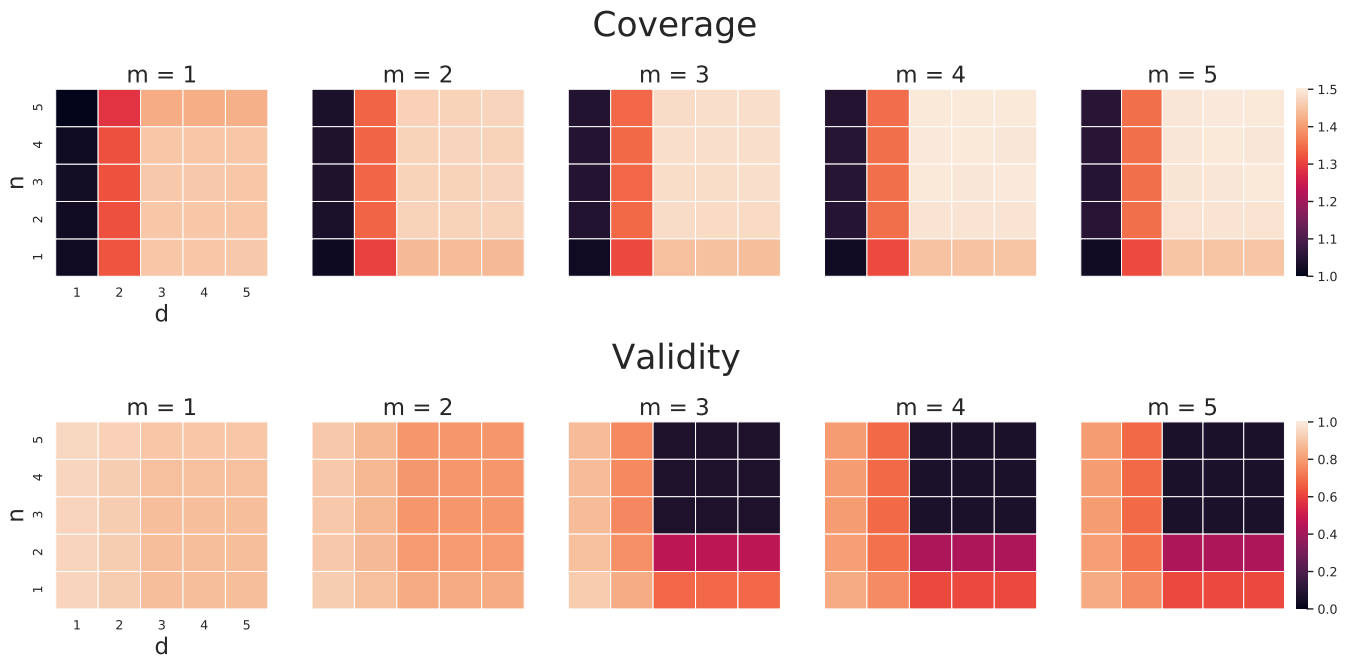

Fig. 9. Validity and coverage of randomly sampled ChocoPy programs

application[2]. 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, which 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. Zest records these choices in a vector, which is associated with the corresponding $input$. The MUTATE function in Algorithm 2 works by performing random point mutations on these recorded pseudo-random choices and then replaying BOUNDEDSAMPLE with the specified choices and with the given bounds to produce $input'$. Essentially, BOUNDEDSAMPLE is implicitly parameterized by the source of pseudo-randomness, which Zest controls—in the implementation, by simply overriding `java.util.Random`. We expect the returned value $input'$ to be a syntactically valid input that is subtly different from—that is, a structural mutation of—the original input $input$ [28]. 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-C.

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 $\mathcal{S}$. Zest thus favors saving semantically valid inputs. Section IV-D describes a tweak to this criterion we make in some scenarios.

The $\mathcal{F}$ notation: We now define some short-hand notation that will be useful when describing our proposed bonsai fuzzing technique. Let $\mathcal{F}^{\mathcal{G},p}_{m,n,d}$ denote a coverage-guided bounded grammar fuzzer (CBGF) parameterized by grammar $\mathcal{G}$, test program p, size bounds m, n, and d. As per Algorithm 2, $\mathcal{F}^{\mathcal{G},p}_{m,n,d}$ 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, $\mathcal{F}_{m,n,d}$ is a CBGF of size bounds (m, n, d).

Preliminary Results with ChocoPy: As described in Section III-C, simply using Zest followed by input minimization on the resulting corpus still lacks conciseness. The program in Fig. 4 was produced using $\mathcal{F}_{3,3,3}$ in seedless mode [40]. 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. A full HDD-reduced corpus of Zest-generated ChocoPy programs can be found in our online repository (ref. Section I).

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

[2]We elide details of all other heuristics in Algorithm 2, since we inherit them from the original Zest implementation [28]. The search heuristics are not important to our proposed technique, which works at a higher level.

Algorithm 2 Coverage-Guided Bounded Grammar Fuzzing

Require: Instrumented program p, Grammar $\mathcal{G}$, Bounds m, n, d

function CBGF(Seed inputs $\mathcal{S}$) ▷ Returns corpus $\supseteq \mathcal{S}$
  repeat
    input ← next($\mathcal{S}$) ▷ Cycle through $\mathcal{S}$
    input' ← MUTATE(*input*, $\mathcal{G}, m, n, d$) ▷ See text...
    feedback ← EXECUTE(p, *input'*) ▷ validity + coverage
    if *feedback* is interesting **then** ▷ new coverage?
      $\mathcal{S} \leftarrow \mathcal{S} \cup$ *input'*
  until time budget expires
  return $\mathcal{S}$

better coverage. By inceasing 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.

Given upper bounds M, N, and D, we can consider the set of CBGFs

$$\mathcal{C}_{M,N,D} = \left\{ \mathcal{F}_{m,n,d} \quad : \quad \begin{matrix} 1 \le m \le M \\ 1 \le n \le N \\ 1 \le d \le D \end{matrix} \right\}$$

With upper bounds $(3, 3, 3)$, we would have 27 different CBGFs in set $\mathcal{C}_{3,3,3}$.

We define a *partial order* $\le$ over $\mathcal{C}_{M,N,D}$ as follows:

$$\mathcal{F}_{m,n,d} \le \mathcal{F}_{m',n',d'} \iff m \le m', n \le n', d \le d'$$

Consequently, $\mathcal{F}_{m,n,d} < \mathcal{F}_{m',n',d'}$ iff $\mathcal{F}_{m,n,d} \le \mathcal{F}_{m',n',d'}$ and $\mathcal{F}_{m,n,d} \ne \mathcal{F}_{m',n',d'}$.

This ordering suggests that $\mathcal{C}_{M,N,D}$ is a lattice with $\mathcal{F}_{1,1,1}$ being the bottom element (denoted $\mathcal{F}_\perp$) and $\mathcal{F}_{M,N,D}$ being the top element (denoted $\mathcal{F}^\top$). Fig. 1 visualizes the lattice for $\mathcal{C}_{2,2,2}$, where the partial order corresponds to graph reachability. In this example, $\mathcal{F}^\top = \mathcal{F}_{2,2,2}$.

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

1) $\mathcal{F}_s$ is a *successor* of $\mathcal{F}$ if $\mathcal{F} < \mathcal{F}_s$ and there exists no CBGF $\mathcal{F}'_s$ such that $\mathcal{F} < \mathcal{F}'_s < \mathcal{F}_s$.
2) $\mathcal{F}_p$ is a *predecessor* of $\mathcal{F}$ if $\mathcal{F}_p < \mathcal{F}$ and there exists no CBGF $\mathcal{F}'_p$ such that $\mathcal{F}_p < \mathcal{F}'_p < \mathcal{F}$.

For example, $\mathcal{F}_{2,1,1}$ is a successor of $\mathcal{F}_{1,1,1}$, whereas $\mathcal{F}_{2,2,1}$ is not. Conversely, $\mathcal{F}_{1,1,1}$ is a predecessor of $\mathcal{F}_{2,1,1}$ but not a predecessor of $\mathcal{F}_{2,2,1}$. In Fig. 1, every node has incoming edges from its predecessors and outgoing edges to its successors. Naturally, $predecessors(\mathcal{F}_\perp) = successors(\mathcal{F}^\top) = \{\}$.

We now formally define *bonsai fuzzing* as a procedure that begins with the smallest configuration $\mathcal{F}_\perp$ and iteratively increases the size until a given upper bound is reached.

Algorithm 3 describes the procedure for bonsai fuzzing. Variable $\mathcal{F}$ is initialized to the smallest CBGF $\mathcal{F}_\perp$. Recall from Algorithm 2 that a CBGF is a function that is given a set of seed inputs and returns a test corpus. Initially, we have

Algorithm 3 Bonsai fuzzing algorithm

procedure BONSAIFUZZING
  $\mathcal{F} \leftarrow \mathcal{F}_\perp$
  $seeds \leftarrow$ [random()] ▷ Single random seed
  $corpus(\mathcal{F}_\perp) = \mathcal{F}(seeds)$ ▷ Run CBGF to generate corpus
  $worklist \leftarrow successors(\mathcal{F})$
  while $worklist$ is not empty **do**
    for each $\mathcal{F}$ in $worklist$ **do** ▷ Parallelizable
      $P \leftarrow predecessors(\mathcal{F})$
      $seeds \leftarrow$ SORTBYSIZE $\left(\bigcup_{\mathcal{F}_p \in P} corpus(\mathcal{F}_p)\right)$
      $corpus(\mathcal{F}) \leftarrow \mathcal{F}(seeds)$ ▷ Run CBGF
    $worklist \leftarrow \bigcup_{\mathcal{F}_s \in worklist} successors(\mathcal{F}_s)$
  return $corpus(\mathcal{F}^\top)$

no seeds. We thus start by running the CBGF $\mathcal{F}_\perp$ with one random seed input (similar to SLF [40]) to produce $corpus(\mathcal{F}_\perp)$. Then, a $worklist$ is populated with the $successors(\mathcal{F})$. For each unprocessed element in the worklist—that is, each unexecuted fuzzer—we prepare its $seeds$ by taking a union of all test cases in the $corpus$ generated by each of its predecessors. The seeds are also sorted by size in ascending order (so that Algorithm 2 encounters smaller inputs to mutate first). We then run $\mathcal{F}$, save its resulting $corpus$, and repeat this process. Eventually, we reach the point where $\mathcal{F} = \mathcal{F}^\top$ and there are no more successors. The final corpus is the result of $\mathcal{F}^\top$.

Consider a sample run of bonsai fuzzing over the set $\mathcal{C}_{3,3,3}$. We start by running the CBGF $\mathcal{F}_\perp = \mathcal{F}_{1,1,1}$ with one randomly generated seed input. Fig. 6 shows three sample test cases saved in the resulting $corpus(\mathcal{F}_{1,1,1})$. We can see that the generated programs are small in size and test simple language features. These inputs will then be used as seeds in successor CBGFs: $\mathcal{F}_{2,1,1}, \mathcal{F}_{1,2,1}$, and $\mathcal{F}_{1,1,2}$. Fig. 7 lists some programs saved in the corresponding corpora of these fuzzers. We can now start to see programs with slightly complex features, such as class attributes, binary expressions, and functions with multiple parameters. We repeat the process until we reach $\mathcal{F}_{3,3,3}$, the top element of the lattice $\mathcal{C}_{3,3,3}$. Fig. 8 shows some example programs saved in the its corpus. More complex features such as nested list expressions and nested function definitions are demonstrated in these generated programs. Note that some of these inputs may have been copied verbatim from its seeds, having been discovered by predecessors. The final corpus necessarily incorporates the corpora generated by all CBGFs in the lattice. The full corpus of ChocoPy programs generated using bonsai fuzzing can be found in our online repository (ref. Section I).

D. Bonsai Fuzzing with Extended Lattice

So far, we have restricted test generation to only those programs that are semantically valid. By and large, we want semantic rules (e.g. well-typed addition) to be exercised in valid representative programs, as opposed to larger invalid programs that contain these as subexpressions. However, in order to have a comprehensive test corpus, we also need some invalid input programs for testing various error paths in the semantic analysis (e.g. duplicate variable definition, non-

boolean condition to `while`, and so on). Ideally, we want these invalid programs to be *concise* as well; that is, they are indicative of the particular error path that is being tested. To achieve this goal, we define two variants of CBGF by tweaking 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 thus add a parameter $v \in \{r, u\}$ to CBGFs, where r denotes restricted and u denotes unrestricted. We use the symbol $\mathcal{F}_{m,n,d,v}$ to denote a CBGF that is parameterized by size bounds as well as the validity restriction (or lack thereof). We can now define a new partial ordering as follows: given two CBGFs $\mathcal{F}_{m,n,d,v}$ and $\mathcal{F}_{m',n',d',v'}$:

$$\mathcal{F}_{m,n,d,v} \leq \mathcal{F}_{m',n',d',v'} \iff \begin{array}{l} \mathcal{F}_{m,n,d} \leq \mathcal{F}_{m',n',d'} \text{ and} \\ (v = v' \text{ or } v' = u) \end{array}$$

The definitions of *successors* and *predecessors* remain the same as before. So we now have $successors(\mathcal{F}_{1,1,1,r}) = \{\mathcal{F}_{2,1,1,r}, \mathcal{F}_{1,2,1,r}, \mathcal{F}_{1,1,2,r}, \mathcal{F}_{1,1,1,u}\}$. Similarly, we now have $predecessors(\mathcal{F}_{1,2,1,u}) = \{\mathcal{F}_{1,1,1,u}, \mathcal{F}_{1,2,1,r}\}$ The key idea of this lattice is that *restricted*-CBGFs are predecessors of *unrestricted*-CBGFs with the same size bounds. In other words, *unrestricted*-CBGFs with size bounds m, n, d will be able to use as seeds all the valid inputs produced by a fuzzer of the same size bounds, as well as both valid and invalid inputs produced by unrestricted fuzzers with smaller size bounds. The hope is that invalid inputs that are generated by mutating valid inputs are more likely to be concise, as they would trigger fewer semantic errors in a single ChocoPy program.

Setting $\mathcal{F}_{\perp} = \mathcal{F}_{1,1,1,r}$ and $\mathcal{F}^{\top} = \mathcal{F}_{M,N,D,u}$, we can run bonsai fuzzing using Algorithm 3 as is.

V. Evaluation

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 [41]: 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 [42]: 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**: Overall, we found the bounds of $(M = N = D = 3)$ to be a good trade-off between conciseness and comprehensiveness. We use these bounds for bonsai fuzzing as well as for the baseline CBGF.
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].
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 [43] and Picireny [44], which are state-of-the-art [32]–[35] implementations of character-level [15] and grammar-based hierarchical [16] delta debugging respectively. An "interestingness" predicate script was required for each of these tools. We provided a predicate that checked whether a candidate 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 generated 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

One of our goals was to generate a high fraction of semantically valid inputs (ref. Section II-B). For each input

[3]We chose these durations because one hour is sufficient time for coverage to stagnate for each CBGF node, and because it helps us make a fair comparison with the baseline by fixing total fuzzing duration to a constant. Bonsai fuzzing can be optimized by stopping each CBGF early by detecting saturation dynamically, but this would make the total fuzzing duration variable. Our evaluation is conservative.

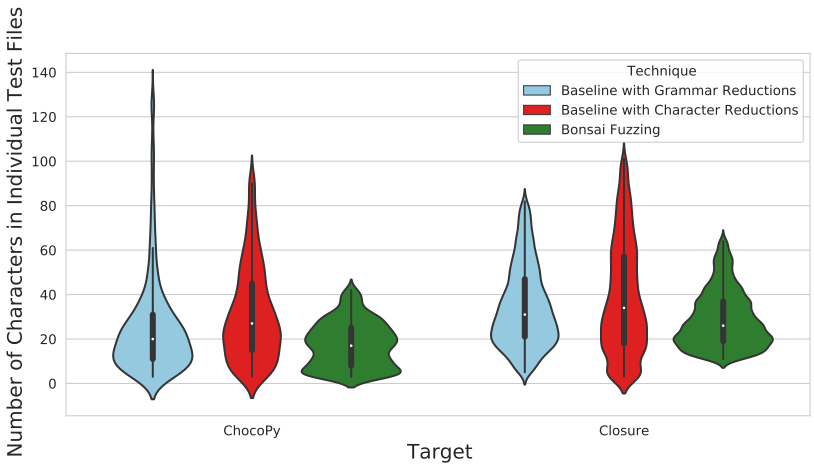

Fig. 10. Distribution of size of individual test files, excluding whitespace characters, in saved test corpora. Lower is better.

TABLE I
TIME TO MINIMIZE ZEST-SAVED TEST INPUTS (MINUTES, AVG ± STDEV).

	ChocoPy	Closure
Picireny Grammar Reductions	56.510 ± 1.887	356.863 ± 37.560
Picire Character Reductions	20.491 ± 3.209	392.777 ± 48.456

TABLE II
NUMBER OF FILES IN TEST CORPUS (AVG ± STDEV). LOWER IS BETTER.

	ChocoPy	Closure
Baseline	186.0 ± 7.528	1507.7 ± 28.351
Bonsai fuzzing	152.9 ± 1.912	1231.2 ± 36.705

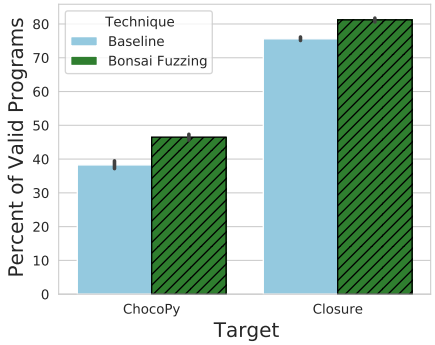

Fig. 11. Fraction of semantically valid programs in test corpora (averages with standard deviation). Higher is better.

program in the saved test corpora, we re-run the ChocoPy compiler to test whether the input is semantically valid or whether the compiler reports any errors.

The average percent of semantically valid programs in the generated 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. Why is this so? In the initial round of bonsai fuzzing, sampling smaller programs leads to a higher likelihood of semantically valid inputs as compared to sampling a larger program from scratch (ref. Fig. 9). In subsequent rounds, it is easier to mutate a small valid program into a slightly larger valid program, as there are less opportunities to introduce errors. We observed that the baseline's seed pool quickly fills up with invalid or large programs early-on in the fuzzing campaign, making it harder to recover in producing diverse valid inputs via random mutations.

We value this improvement in validity resulting from bonsai fuzzing, 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 [45]. 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.

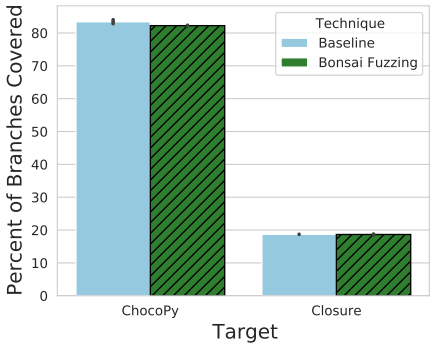

Fig. 12. Branch coverage in semantic analysis stages achieved by saved test corpora (averages with standard deviation). Higher is better.

TABLE III
MUTATION SCORES FOR CHOCOPY TYPECHECKER (AVG ± STDEV)

Baseline	$91.486 \pm 1.012\%$
Bonsai Fuzzing	$90.428 \pm 0.714\%$

D. Fault Detection: Mutation Scores

Finally, we want to ensure that the concise inputs generated by bonsai fuzzing for the ChocoPy target are still useful for catching faults; that is, they can be used for automated grading or providing student feedback. This is essentially a validation of the small scope hypothesis [46]. 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 [47] 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 saved by bonsai fuzzing by itself achieves a mutation-killing score of 81% on average. This is despite the fact that the fuzzing technique and input-saving criteria is related to coverage improvements within the reference compiler only, and is unaware of program mutations or bugs in student implementations. As recently observed by Chen et al. [48], a better 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–that 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 achieve more than 90% mutation-killing score, which we find to be 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 differential testing oracle.

VI. DISCUSSION AND THREATS TO VALIDITY

Although our original motivation for this work was to synthesize concise test inputs for ChocoPy programming assignments, we also evaluated our technique on the Google Closure Compiler. The results of are promising. Bonsai fuzzing can synthesize test inputs that are concise by construction, without sacrificing the quality of test inputs in terms of code coverage or mutation scores as compared to the fuzz-then-reduce approach. Moreover, the test inputs produced using bonsai fuzzing are smaller in size by 16–45%.

However, since the number of target programs we evaluated on is small, we cannot claim that this technique will generalize more broadly. Further, we restricted our evaluation only to compilers, where the input can be represented by a context-free grammar (CFG). We leave the generalization of this technique to other input formats and problem domains as future work.

Further, in our evaluation, we fixed the final size bounds to $(3, 3, 3)$ and fuzzing duration to one hour per CBGF node. Bonsai fuzzing can be improved further by dynamically choosing ideal size bounds and fuzzing duration by monitoring the quality of test inputs saved by each CBGF node.

We were unfortunately unable to test fault detection capabilities of bonsai fuzzing on actual student implementations, due to procedural issues with using student-authored assignments for this research. We used *mutation scores* to estimate the ability of bonsai fuzzing to catch student bugs. Prior empirical studies have shown this metric to be reasonable [49], but we cannot make general claims about the impacts of this research in the classroom.

In this paper, we used the notion of *conciseness* of test inputs as a proxy for readability, based on what we feel are important features of readable test cases (size and semantic validity). Since our evaluation did not comprise of a user study, we cannot make any subjective claims about human-perceived readability. Independently from our work, Roy et al. [50] have recently worked on improving the readability of automatically generated test *code*, addressing issues such as variable names and code comments.

It has not escaped our notice that the bonsai fuzzing technique may also be useful in synthesizing regression tests for fast evolving software. For validating code changes, it is much more efficient to simply run a fixed suite of regression tests than to run a full fuzzing session after every code commit. Concise test inputs, such as those produced using bonsai fuzzing, are more likely to be maintainable.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants CCF-1900968, CCF-1908870, and CNS-1817122, as well as by gifts from Fujitsu and CyLab. The experimental evaluation for this paper was supported by the Amazon AWS Cloud Credits for Research program. We would like to thank Eric Eide for sharing a corpus of C-Reduced [2] programs, which helped us form some of our initial insights about program sizes.

References

[1] R. Padhye, K. Sen, and P. N. Hilfinger, "Chocopy: A programming language for compilers courses," in *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, ser. SPLASH-E 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 41–45. [Online]. Available: https://doi.org/10.1145/3358711.3361627

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, 2011.

[3] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes, "Testing an optimising compiler by generating random lambda terms," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 91–97. [Online]. Available: https://doi.org/10.1145/1982595.1982615

[4] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using clp," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 482–493. [Online]. Available: https://doi.org/10.1109/ASE.2015.65

[5] B. Fetscher, K. Claessen, M. Pałka, J. Hughes, and R. B. Findler, "Making random judgments: Automatically generating well-typed terms from the definition of a type-system," in *European Symposium on Programming Languages and Systems*. Springer, 2015, pp. 383–405.

[6] A. Arcuri, "A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage," *IEEE Transactions on Software Engineering*, vol. 38, no. 3, pp. 497–519, 2012.

[7] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[8] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831728

[9] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[10] Yong Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, 2005, pp. 10 pp.–276.

[11] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 417–420. [Online]. Available: https://doi.org/10.1145/1321631.1321698

[12] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ser. ICSTW '10. USA: IEEE Computer Society, 2010, p. 182–191. [Online]. Available: https://doi.org/10.1109/ICSTW.2010.31

[13] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[14] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[15] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[16] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 142–151. [Online]. Available: https://doi.org/10.1145/1134285.1134307

[17] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[18] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 123–133. [Online]. Available: https://doi.org/10.1145/566172.566191

[19] S. Khurshid and D. Marinov, "Testera: Specification-based testing of java programs using sat," *Automated Software Engineering*, vol. 11, no. 4, pp. 403–434, 2004.

[20] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 185–194. [Online]. Available: https://doi.org/10.1145/1287624.1287651

[21] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 225–234. [Online]. Available: https://doi.org/10.1145/1806799.1806835

[22] N. Havrikov and A. Zeller, "Systematically covering input structure," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 189–199. [Online]. Available: https://doi.org/10.1109/ASE.2019.00027

[23] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF-SE: a symbolic execution extension to Java PathFinder," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[24] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[25] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[26] M. Zalewski, "American fuzzy lop," https://github.com/google/AFL, 2014, accessed August 1, 2020.

[27] LLVM Compiler Infrastructure, "libFuzzer," https://llvm.org/docs/LibFuzzer.html, 2016, accessed August 1, 2020.

[28] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with Zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 329–340. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330576

[29] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for Deep Bugs with Grammars," in *26th Annual Network and Distributed System Security Symposium*, ser. NDSS '19, 2019.

[30] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *41st International Conference on Software Engineering*, ser. ICSE '19, 2019.

[31] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 361–371. [Online]. Available: https://doi.org/10.1145/3180155.3180236

[32] R. Hodován and Á. Kiss, "Practical improvements to the minimizing delta debugging algorithm." in *ICSOFT-EA*, 2016, pp. 241–248.

[33] ——, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, 2016, pp. 31–37.

[34] R. Hodován, Á. Kiss, and T. Gyimóthy, "Coarse hierarchical delta debugging," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 194–203.

[35] ——, "Tree preprocessing and test outcome caching for efficient hierarchical delta debugging," in *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. IEEE, 2017, pp. 23–29.

[36] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[37] R. Scowen, "International Standard ISO/IEC 14977:1996—Extended BNF," 1996.

[38] S. Luke, "Two fast tree-creation algorithms for genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 4, no. 3, pp. 274–283, 2000.
[39] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.
[40] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "SLF: Fuzzing without valid seed inputs," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 712–723. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00080
[41] U. Berkeley, "Chocopy," https://chocopy.org/, 2019.
[42] Google, "Google Closure Compiler," https://github.com/google/closure-compiler.
[43] R. Hodovan, "Picire," https://github.com/renatahodovan/picire.
[44] ——, "Picireny," https://github.com/renatahodovan/picireny.
[45] Eclemma, "Jacoco java code coverage library," https://www.eclemma.org/jacoco.
[46] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.
[47] H. Coles, "Pitest," https://pitest.org/.
[48] Y. Chen, R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just, "Revisiting the relationship between fault detection, test adequacy criteria, and test set size," in *IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2020.
[49] R. Just and F. Schweiggert, "Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 490–507, 2015.
[50] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, "DeepTC-Enhancer: Improving the readability of automatically generated tests," in *IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'20. IEEE, 2020.