Automatic and Scalable Detection of Logical Errors in Functional Programming Assignments

DOWON SONG, Korea University, Republic of Korea
MYUNGHO LEE, Korea University, Republic of Korea
HAKJOO OH[*], Korea University, Republic of Korea

We present a new technique for automatically detecting logical errors in functional programming assignments. Compared to syntax or type errors, detecting logical errors remains largely a manual process that requires hand-made test cases. However, designing proper test cases is nontrivial and involves a lot of human effort. Furthermore, manual test cases are unlikely to catch diverse errors because instructors cannot predict all corner cases of diverse student submissions. We aim to reduce this burden by automatically generating test cases for functional programs. Given a reference program and a student's submission, our technique generates a counter-example that captures the semantic difference of the two programs without any manual effort. The key novelty behind our approach is the counter-example generation algorithm that combines enumerative search and symbolic verification techniques in a synergistic way. The experimental results show that our technique is able to detect 88 more errors not found by mature test cases that have been improved over the past few years, and performs better than the existing property-based testing techniques. We also demonstrate the usefulness of our technique in the context of automated program repair, where it effectively helps to eliminate test-suite-overfitted patches.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Software testing and debugging**.

Additional Key Words and Phrases: Automated Test Case Generation, Program Synthesis, Symbolic Execution

ACM Reference Format:
Dowon Song, Myungho Lee, and Hakjoo Oh. 2019. Automatic and Scalable Detection of Logical Errors in Functional Programming Assignments. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 188 (October 2019), 30 pages. https://doi.org/10.1145/3360614

1 INTRODUCTION

Motivation. In a functional programming course taught by the authors over the past few years, we have repeatedly experienced that detecting logical errors in student submissions is challenging. In a real classroom setting, hundreds of students submit programming assignments on which instructors are required to provide feedback. Logical errors (i.e., errors producing unintended behaviors) are the most difficult type of errors to provide useful feedback compared to syntax or

[*]Corresponding author

Authors' addresses: Dowon Song, dowon_song@korea.ac.kr, Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea; Myungho Lee, myungho_lee@korea.ac.kr, Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea; Hakjoo Oh, hakjoo_oh@korea.ac.kr, Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART188
https://doi.org/10.1145/3360614

type errors. Although a lot of supportive tools are available for syntax or type errors, detecting logical errors remains largely a manual process that requires hand-made test cases.

However, using manual test cases to detect logical errors is hardly effective. Manually generating test cases is a challenging and burdensome task. To be a solid test suite, it must be carefully designed to cover various behaviors of a program, which is practically infeasible for numerous submissions. In other words, the insufficient test suite may miss some erroneous programs when their abnormal behaviors are not covered by the test suite, failing to provide helpful feedback. In our programming course, for instance, we found that a significant number of incorrect submissions received full credit due to the difficulty of designing high-quality test cases (Section 6.1).

Existing techniques for automatic test case generation are also have their problems. The most popular approach for automatically generating test cases for functional programs is property-based testing (e.g., QuickCheck [Claessen and Hughes 2000]). However, property-based testing has two major drawbacks. First, it is essentially random testing and therefore not guaranteed to detect program-specific, corner-case errors. Furthermore, property-based testing requires users to manually provide proper ingredients (e.g., generators and shrinkers [Claessen and Hughes 2000]), which makes the testing process burdensome. To capture the program-specific behaviors, symbolic execution [Cadar et al. 2011; Khurshid et al. 2003; King 1976] can be used, but pure symbolic techniques are not easily applicable in our case because of functional features such as higher-order functions.

Goal and Approach. In this paper, we present a new technique for effectively detecting logical errors in functional programming assignments. Given a reference program and a student program, our technique aims to find a test case that demonstrates the behavioral difference of them. In particular, our technique does so in a fully automatic way and is capable of handling diverse functional programming features such as higher-order functions and algebraic data types.

The key idea of the algorithm is to combine enumerative search and symbolic verification techniques in a synergistic way. In order to generate a test case that shows the behavioral difference of two programs, our algorithm basically performs type-directed enumerative search over the space of test cases. That is, we enumerate inputs in increasing size and check whether each candidate counter-example is able to trigger the behavioral difference or not. This enumerative search is well-known for its effectiveness at synthesizing small code fragments [Feser et al. 2015; Lee et al. 2018a; So and Oh 2017], and therefore we use it to generate test cases that involve function bodies or user-defined data. However, the enumeration-only approach is inappropriate for inferring primitive values such as integers and strings because there are infinitely many values to consider at a single step of enumeration. We overcome this shortcoming by leveraging a symbolic verification technique. That is, we do not directly generate integer and string constants during the enumerative search but represent them as symbols, which produces "symbolic test cases" instead of concrete ones. Checking whether a symbolic test case can trigger the behavioral difference of the two programs is done by first performing symbolic execution on the programs and then solving the resulting verification condition with the SMT solver. We do not use this symbolic technique for non-primitive values such as functions and user-defined data because supporting them in symbolic analysis is heavy, but the enumerative technique can handle them in a relatively simple and effective way. This way, enumerative search and symbolic techniques work together, overcoming the key shortcomings of each other.

Empirical Results. The experimental results show that our approach effectively detects logical errors in real submissions written in OCaml. We evaluated it on 4,060 student submissions collected from our undergraduate functional programming course. Our automated approach successfully found 631 erroneous submissions while manually-designed test cases only detected 538 of them.

This is remarkable because those test cases have been carefully designed, refined, and used in the course over the last three years. Moreover, we found that the test cases generated by our technique are more concise and easier to understand the cause of logical errors than the manual test cases. Our experiments demonstrate that our approach is more effective and efficient than an existing property-based test case generator QCheck, an OCaml version of QuickCheck [Claessen and Hughes 2000], without any human effort. Furthermore, we show that our approach is useful in the context of automated program repair. When we used our counter-example generation algorithm in combination with an existing repair system for functional programs [Lee et al. 2018b], the number of test-suite-overfitted patches reduced significantly.

Contributions. In this paper, we make the following contributions:

- We propose a technique for detecting logical errors in functional programming assignments, which combines enumerative search and symbolic execution in a novel way. Our approach is fully automatic and is able to handle functional features such as higher-order functions effectively.
- We conduct extensive evaluations with real students' submissions. The evaluation results demonstrate that our approach is effective both in error detection of real-world submissions and alleviating test-suite-overfitted patches in automated program repair.
- We provide our counter-example algorithm as a tool, called TestML. Our tool and benchmarks used in the experiments are publicly available.[1]

2 MOTIVATING EXAMPLES

In this section, we motivate our technique with examples. We consider three programming exercises used in our undergraduate course on functional programming.

Example 1. Let us consider a programming exercise, where students are asked to write a function, called `diff`, which symbolically differentiates arithmetic expressions. The arithmetic expressions are defined as an OCaml datatype as follows:

```
type aexp =
  Const of int | Var of string | Power of (string * int) | Sum of aexp list | Times of aexp list
```

An expression (`aexp`) is either constant integer (`Const`), variable (`Var`), exponentiation (`Power`), addition (`Sum`), or multiplication (`Times`). For instance, the expression $x^2 + 2y + 1$ is represented as `Sum [Power ("x", 2); Times [Const 2; Var "y"]; Const 1]`. The function `diff`, whose type is `aexp * string -> aexp`, takes a pair of an expression and a variable name, and differentiates the expression with respect to the given variable. For example, `diff (Sum [Power ("x", 2); Times [Const 2; Var "y"]; Const 1], "x")` should produce `Times [Const 2; Var "x"]`. Fig 1a shows a reference implementation of `diff` provided by the instructor.

Fig 1b shows a student's implementation that has a subtle bug. The function `diff` is defined at line 28 in terms of two helper functions: `do_diff` and `minimize`. The `do_diff` function is responsible for actual differentiation, and `minimize` simplifies the resulting expression. For example, evaluating `diff (Times [Const 0; Var "x"], "x")` first calls `do_diff (Times [Const 0; Var "x"], "x")` to perform differentiation, resulting in a rather verbose expression, `Sum [Times [Const 0; Var "x"]; Times [Const 0; Const 1]]`. Then, `diff` uses `minimize` to simplify the resulting expression and produce `Const 0` as a final output. The program works correctly for most cases, but it rarely causes unexpected results. For example, `diff (Times [Const 1; Var "x"], "x")` produces `Const 0` when the desired output is `Const 1`. The problem is in `minimize`; it incorrectly simplifies the expressions of the form `Times [Const 1; ...; Const 1]` to `Const 0`. Note that the

[1] https://github.com/kupl/TestML

```
let rec diff (e, x) =
  match e with
  | Const n -> Const 0
  | Var y -> if (x <> y) then Const 0 else Const 1
  | Power (y, n) -> if (x <> y) then Const 0 else Times [Const n; Power (y, n-1)]
  | Sum (hd::tl) -> Sum (List.map (fun e -> diff (e, x)) (hd::tl))
  | Times [hd] -> diff (hd, x)
  | Times (hd::tl) -> Sum [Times ((diff (hd, x))::tl); Times [hd; diff (Times tl, x)]]
  | _ -> raise (Failure "Invalid")
```

(a) A reference implementation

```
let rec do_diff (ae, x) =
  match ae with
  | Const i -> Const 0
  | Var v -> if (v = x) then Const 1 else Const 0
  | Power (v, i) -> if (v = x) Times [Const i; Power (v, i-1)] else Const 0
  | Sum (hd::tl) ->
    if (tl = []) then do_diff (hd, x) else Sum [do_diff (hd, x); do_diff (Sum tl, x)]
  | Times (hd::tl) ->
    if (tl = []) then do_diff (hd, x)
    else Sum [Times ((do_diff (hd, x))::tl); Times [hd; (do_diff (Times tl, x))]]

let rec minimize ae =
  let rec minimize_helper ae =
    match ae with
    | Sum lst ->
      if (lst = []) then Const 0
      else if (List.length lst = 1) then List.hd lst
      else Sum (List.map minimize_helper (List.filter (fun ae -> ae <> Const 0) lst))
    | Times lst ->
      if (lst = []) then Const 0
      else if (List.mem (Const 0)) lst then Const 0
      else if (List.length lst = 1) then List.hd lst
      else Times (List.map minimize_helper (List.filter (fun ae -> ae <> Const 1) lst))
    | _ -> ae in
  let ae' = minimize_helper ae in
  if (ae = ae') then ae else minimize ae'

let diff (ae, str) = minimize (do_diff (ae, str))
```

(b) A buggy implementation

Fig. 1. Example 1: `diff`

bug is caused only when `do_diff` yields a sequence of 1s in its output. Otherwise, `diff` behaves correctly, e.g., `diff (Times [Const 2; Var "x"], "x")`. Manually detecting such a corner-case bug is challenging. In fact, the student code in Fig 1b passed all the test cases provided by the instructor and received the full credit.

Our technique can find this bug in 0.2 seconds. It takes the buggy and reference implementations in Fig 1 as inputs and generates a test case `(Times [Const 1; Var "x"], "x")` on which the two

```
1 let rec iter : int * (int -> int) -> int -> int
2 = fun (n, f) x ->
3   if (n < 0) then raise (Failure "Invalid Input")
4   else if (n = 0) then x
5   else f (iter (n-1, f) x)
```

(a) A reference implementation

```
1 let rec iter : int * (int -> int) -> int -> int
2 = fun (n, f) x ->
3   let y = f x in
4   if (n <= 0) then x else iter (n-1, f) y
```

(b) A buggy implementation

Fig. 2. Example 2: `iter`

programs produce different results. Note that our technique is able to infer the specific integer and string constants, i.e., 1 and "x", automatically without requiring any manual effort of the instructor.

Example 2. The second exercise is to write a higher-order function called `iter`. The function, `iter: int * (int -> int) -> int -> int`, takes two arguments. The first is a pair of an integer `n` and an integer-valued function `f`. The second is an integer `x`. Then, `iter(n,f) x` evaluates to the following:

$$\mathtt{iter}(\mathtt{n}, \mathtt{f})\ \mathtt{x} = (\underbrace{\mathtt{f} \circ \cdots \circ \mathtt{f}}_{\mathtt{n}})(\mathtt{x})$$

For instance, `(iter (5, fun x -> 1 + x) 2)` evaluates to 7. When `n` is 0, `iter(n,f)` is defined to be an identity function. Fig 2a shows a reference implementation of `iter`.

Fig 2b shows a program written by a student, which has a tricky bug that is hard to anticipate when manually designing test cases. Note that the student implementation is very similar to the reference implementation. If `n` is no greater than 0, the result is the identity function (line 4). Otherwise, at line 5, it evaluates `iter (n-1, f) y`, where `y` is the result of the single application of `f` to `x`. The overall logic is correct and therefore the program works well in most cases. For example, it correctly evaluates `(iter (5, fun x -> 1 + x) 2)` to 7. However, the program runs into trouble if `n` is 0 and `f` is undefined on `x` because it attempts to evaluate the function application (`f x`) even when `n` is 0 at line 3. For example, evaluating `(iter (0, fun x -> 1 mod x) 0)` causes a division-by-zero error while the reference implementation produces 0 without any runtime errors. We found that this submission also received the full credit as our manually crafted test cases could not check this corner case.

On the other hand, our technique quickly detects the bug in 0.2 seconds. Given the two (correct and incorrect) programs, our technique generates `(0, fun x -> 1 mod x)` for the first argument, i.e., `(n, f)`, and `0` for the second argument, i.e., `x`. Note that our technique is able to generate test cases for high-order functions; that is, it can synthesize the function (`fun x -> 1 mod x`) as the input of `iter`.

Example 3. In this example, we demonstrate another promising application of our technique; it can resolve a common problem in automatic program repair, called *test-suite-overfitted patches* [Smith et al. 2015]. In recent years, several techniques have been proposed for automatic program repair systems that use test cases for checking the correctness of the repaired programs. However, these systems often produce test-suite-overfitted patches which retain some bugs but satisfy the

```
 1 let rec exp_eval e =
 2   match e with
 3   | Num n -> n
 4   | Plus (e1, e2) -> (exp_eval e1) + (exp_eval e2)
 5   | Minus (e1, e2) -> (exp_eval e1) - (exp_eval e2)
 6
 7 let rec eval f =
 8   match f with
 9   | True -> true
10   | False -> false
11   | Not f -> not (eval f)
12   | AndAlso (f1, f2) -> (eval f1) && (eval f2)
13   | OrElse (f1, f2) -> (eval f1) || (eval f2)
14   | Imply (f1, f2) -> not (eval f1) || (eval f2)
15   | Less (e1, e2) -> (exp_eval e1) < (exp_eval e2)
```

(a) A reference implementation

```
 1 let rec eval_expr e =
 2   match e with
 3   | Num x -> x
 4   | Plus (x, y) -> (eval_expr x) + (eval_expr y)
 5   | Minus (x, y) -> (eval_expr x) + (eval_expr y)
 6
 7 let rec eval f =
 8   match f with
 9   | True -> true
10   | False -> false
11   | Not x -> not (eval x)
12   | AndAlso (x, y) -> (eval x) && (eval y)
13   | OrElse (x, y) -> (eval x) || (eval y)
14   | Imply (x, y) -> not (eval x) || (eval y)
15   | Less (x, y) -> (eval_expr x) < (eval_expr y)
```

(b) A buggy implementation

Fig. 3. Example 3: `eval`

given test cases. We show that our technique can enhance the performance of an existing program repair system, FixML [Lee et al. 2018b], for functional programming assignments.

Consider the exercise of writing a function, `eval:formula -> bool`, which evaluates both propositional formula (`formula`) and arithmetic expression (`exp`) defined as following OCaml datatype:

```
type formula =
    True | False | Not of formula | AndAlso of formula * formula
  | OrElse of formula * formula | Imply of formula * formula | Less of exp * exp
and exp = Num of int | Plus of exp * exp | Minus of exp * exp
```

The program in Fig 3b is a buggy version of function `eval`. Much of this program is written correctly, but it has an error at line 5; the function `eval_expr` computes an addition instead of a subtraction when it takes an expression with a pattern `Minus (x, y)`. To correctly fix this error, it must be written as `(eval_expr x) - (eval_expr y)`.

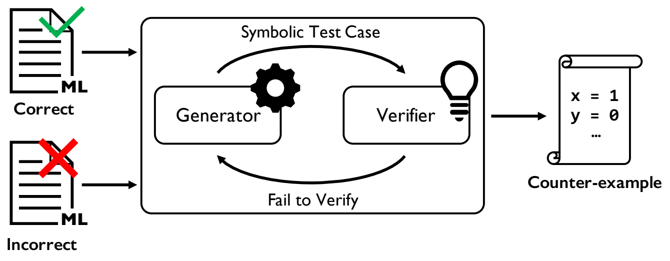

Fig. 4. Overview of the approach

FixML requires users to provide test cases that are able to demonstrate the error and a reference program. We provide a reference program in Fig 3a and 10 nontrivial test cases that we actually have used for grading submissions. Then, FixML generates a patch which replaces the line 4 by `(eval_expr y) + (eval_expr y)`. However, this patch is obviously incorrect and overfitted to the given test cases. Consider the input-output example contained in our test cases:

```
Less (Plus (Minus (Num 4, Num 5), Minus (Num 1, Num (-1))), Plus (Minus (Num 3, Num (-5)), Plus
    (Num 4, Num 5))) -> true
```

where both the expressions `Plus` and `Minus` are calculated abnormally. The generated patch passes this test case by chance; for example, the incorrect patch evaluates true ($0 < 20$) with the test case while the result of the solution is also true ($1 < 17$).

With an aid of our technique, we construct a system so called a counter-example guided program repair system, which will be discussed in Section 6.4. With our technique, FixML no longer requires manual test cases and also resolves the test-suite-overfitted problem by generating the following four test cases automatically during the patch generation process:

```
Less (Num 0, Minus (Num 0, Num (-1))) -> true
False -> false
Less (Num 0, Num 0) -> false
Less (Num 0, Minus (Num (-1), Num (-2))) -> true
```

With these four test cases, FixML successfully created a correct patch; it replaced line 5 in Fig 3b by `(eval_expr x) - (eval_expr y)`.

3 INFORMAL OVERVIEW OF OUR TECHNIQUE

In this section, we informally describe our approach with a simple example in Fig 5, which aims to find the maximum element of a given integer list. While the code on the left is correct, the right is buggy because it assumes that the elements of given list are always bigger than -999. For example, when the input list only contains elements less than -999 (e.g., $[-1000; -1001; \ldots]$), the program incorrectly returns -999 as an output.

Fig 4 overviews our approach. It consists of two key components: symbolic test case generation (Generator) and verification (Verifier). Given a reference implementation and a buggy implementation, our algorithm finds a counter-example on which the two programs produce different outputs. Our algorithm does so by iterating the two phases in an interactive loop.

Symbolic Test Case Generation. To detect a counter-example of two programs, we basically perform the enumerative search which attempts all possible test cases from the smallest one until we find one that causes different outputs of the programs. For example, we initially generate an empty list `[]` for the first trial, but it is not a counter example because the function `max` in a solution

```
let rec max l =
  match l with
  | [] -> raise (Failure "Invalid Input")
  | [e] -> e
  | h::t -> if h > (max t) then h else max t
```

(a) A reference implementation

```
let rec max l =
  match l with
  | [] -> -999
  | h::t -> if h > (max t) then h else max t
```

(b) A buggy implementation

Fig. 5. Example programs to demonstrate our approach

program is designed not to take an empty list as an input. Next, we generate an integer list with one element by producing [□], which denotes a list with one hole, and by completing □ with other integer components. It is important to decide which integer value to use since the behavior of a program is easily influenced by a specific value. For example, if we only use the positive integers, the error in Fig 5b is never detected; thus, it is reasonable to consider all signed integers which can cover negative values. However, enumerating all integers is problematic because it has a huge search space that can generate 2^{32} lists.

To resolve this problem, we introduce symbols to abstract the elements in infinite domains. That is, we now produce a "symbolic test case" instead of a concrete one. Generating symbolic test case has two benefits. First, it reduces the huge search space caused by enumerating infinite primitive values. For instance, we can express all integer lists with one element as a list with one integer symbol $[\alpha^{\text{int}}]$. Furthermore, it helps to automatically determine which constant components to use without user's assumption, which will be discussed in the next section.

Symbolic Verification. The next step is symbolic verification. To check whether a symbolic test case can be a counter-example or not, we leverage symbolic execution [Cadar et al. 2011; Khurshid et al. 2003; King 1976]. Consider a list with an integer symbol $[\alpha^{\text{int}}]$ generated in the previous step. To summarize all behaviors when each program takes $[\alpha^{\text{int}}]$ as an input, our symbolic executor computes the set of possible outputs and their corresponding path conditions. For example, we can obtain a set $\{(\text{true}, \alpha^{\text{int}})\}$ from the reference implementation program because it returns the element of a given list when its length is 1. Similarly, we can abstract the possible execution results of the buggy implementation as $\{(\alpha^{\text{int}} > -999, \alpha^{\text{int}}), (\alpha^{\text{int}} \leq -999, -999)\}$ representing two possible outputs depending on the value of the symbol.

Finally, our algorithm examines the symbolic execution results to decide whether the generated test case is an actual counter-example. We briefly explain how we make a decision on the correctness of a program with respect to a given solution program. Intuitively, we can claim that the submission is correct when it includes all behaviors of the solution program. Suppose the following programs are the two versions of implementations for the function max:

(a)
```
let rec max l =
  match l with
  | [] -> 0
  | [e] -> e
  | h::t -> if h > (max t) then h else max t
```

(b)
```
let rec max l =
  match l with
  | [a; b] -> if a > b then a else b
  | h::t -> if h > (max t) then h else max t
```

The program (a) is implemented to work even when the input is an empty list which is not defined in solution. However, since this program contains all behaviors of the solution program (i.e., it correctly works when the length of input is 1 or more than 2), we can say that this program is correct. On the other hand, the program (b) is incorrect because it causes a pattern-matching failure by an input list with one element (i.e., it does not cover the behavior of the correct one).

Using the symbolic execution results $\{(\text{true}, \alpha^{\text{int}})\}$ and $\{(\alpha^{\text{int}} > -999, \alpha^{\text{int}}), (\alpha^{\text{int}} \leq -999, -999)\}$, the algorithm constructs the following verification condition that checks whether the submission can cover all possible outputs of the solution:

$$\text{true} \Rightarrow ((\alpha^{\text{int}} > -999 \wedge \alpha^{\text{int}} = \alpha^{\text{int}}) \vee (\alpha^{\text{int}} \leq -999 \wedge \alpha^{\text{int}} = -999))$$

It indicates that for any paths in solution (true), there exists a feasible path in the submission ($\alpha^{\text{int}} > -999$ and $\alpha^{\text{int}} \leq -999$) such that the outputs of each program are equivalent ($\alpha^{\text{int}} = \alpha^{\text{int}}$ and $\alpha^{\text{int}} = -999$). If the value of α^{int} is less than -999, such as -1000, the formula evaluates to false. Finally, we can generate a counter example $[-1000]$ by substituting the symbol α^{int} to -1000.

4 PROBLEM DEFINITION

In this section, we formulate the problem this paper aims to solve.

Program. Let us consider a small ML-like programming language. We assume that a program P is a single recursive function definition, which is represented by a triple as follows:

$$P = (f, x, E)$$

where f is the name of the recursive function, x is the formal parameter, and E is the expression denoting the function body. For simplicity, we assume that a program takes a single argument, and the body expression is defined by the following grammar:

$$\begin{array}{rcl} E & ::= & n \mid s \mid x \mid E_1 \oplus E_2 \mid E_1\hat{\ }E_2 \mid E_1\ E_2 \mid c(E_1, E_2) \mid \lambda x.E \\ & \mid & \text{let } x = E_1 \text{ in } E_2 \mid \text{let rec } f(x) = E_1 \text{ in } E_2 \mid \text{match } E_0 \text{ with } \overline{p_i \to E_i}^k \end{array}$$

An expression (E) is either integer constant (n), string constant (s), variable (x), arithmetic operation ($E_1 \oplus E_2$, where $\oplus \in \{+, -, *, /, \text{mod}\}$), string concatenation ($E_1\hat{\ }E_2$), function application ($E_1\ E_2$), user-defined type constructor ($c(E_1, E_2)$, where c is the constructor and for simplicity we assume constructors carry two values), function definition ($\lambda x.E$), let expression ($\text{let } x = E_1 \text{ in } E_2$), recursive function definition ($\text{let rec } f(x) = E_1 \text{ in } E_2$), or pattern matching ($\text{match } E_0 \text{ with } \overline{p_i \to E_i}^k$, where $\overline{p_i \to E_i}^k$ represents $p_1 \to E_1 \mid \cdots \mid p_k \to E_k$). Patterns ($p$) include integer pattern (n), string pattern (s), variable (x), constructor ($c(p_1, p_2)$), and wild card (_). In this language, types (τ) consist of integer type (int), string type (string), user-defined algebraic data types (T), function types ($\tau_1 \to \tau_2$), and type variables for polymorphic type (t). We assume the existence of the table Λ that maps constructors to their type information: for each constructor c, Λ maps it to $(\tau_1 * \tau_2) \to T$, where τ_1, τ_2 are types of the values associated with the constructor and T is the data type of the constructor. Let C be the set of constructors defined in the program.

We assume the standard call-by-value evaluator for expressions, denoted $\mathcal{E}[\![E]\!] : Env \to Val$, which takes an environment and computes the value of the expression E. An environment $\rho \in Env : Id \to Val$ maps variables (Id) to values (Val). The values include integers ($\mathbb{Z}$), strings ($\mathbb{S}$), user-defined constructors ($Cnstr$), functions ($Closure$), and recursive functions ($RecClosure$), and there is a special value ($\perp$) meaning runtime exceptions such as pattern-matching failure, division by zero, or exceeding a predefined time limit:

$$Val = \mathbb{Z} + \mathbb{S} + Cnstr + Closure + RecClosure + \{\perp\},$$

where $Cnstr = Id \times Val^*$ (constructor name and associated values), $Closure = Id \times E \times Env$ (formal parameter name, body, and function-creation environment), and $RecClosure = Id \times Id \times E \times Env$ (function name, formal parameter name, body, and function-creation environment).

Test Case. Next, we define the space of test cases which are input values of the program. We assume that the test cases are defined by the following grammar, which is a subset of our language:

$$I ::= n \mid s \mid c(I_1, I_2) \mid \lambda x.I \mid x \mid I_1 \oplus I_2 \mid I_1\hat{\ }I_2 \tag{1}$$

In our language, test cases can be integers (n), strings (s), constructors ($c(I_1, I_2)$), or functions ($\lambda x.I$). To represent the body of a function, the grammar includes variables (x) and binary operations ($I_1 \oplus I_2, l_1\hat{\ }l_2$) as well. In general, a test case is an expression of the language. However, we do not consider complex expressions such as let expression, recursive function definition, and pattern matching because they are rarely used in test cases. Instead, we focus on a small yet expressive subset of the language, which makes our test-case generation problem more tractable by reducing the search space.

Counter-Example Generation Problem. Let us assume that the two programs P_1 and P_2 are given with the same function name f, which are supposed to implement the same functionality:

$$P_1 = (f, x, E_1), \qquad P_2 = (f, x, E_2).$$

We call P_1 a reference program, which is correct, and P_2 a test program with a potential bug. Our goal is to find a test case (called counter-example) such that evaluating P_1 and P_2 with the same test case produces different results. More precisely, we say a test case $i \in I$ is a counter-example when it satisfies the predicate $\mathsf{CounterExample}(P_1, P_2, i)$ that holds if the two conditions are met:

(1) the reference program does not cause a runtime exception:

$$\mathcal{E}[\![E_1]\!](\rho_1) \neq \bot,$$

(2) and the reference program and the test program disagree on the test case i:

$$\mathcal{E}[\![E_1]\!](\rho_1) \neq \mathcal{E}[\![E_2]\!](\rho_2)$$

where ρ_1 and ρ_2 are initial environments for P_1 and P_2, respectively, which are defined as follows:

$$\rho_1 = [x \mapsto \mathcal{E}[\![i]\!]([]), f \mapsto (f, x, E_1, [])], \quad \rho_2 = [x \mapsto \mathcal{E}[\![i]\!]([]), f \mapsto (f, x, E_2, [])].$$

Note that we can convert test cases (i) to input values ($\mathcal{E}[\![i]\!]([])$) using the evaluator ($\mathcal{E}[\![-]\!]$) for expressions because test cases are defined as a subset of expressions.

5 ALGORITHM

In this section, we present our counter-example generation algorithm.

5.1 Overall Algorithm

Our algorithm basically searches through the space of symbolic test cases. We first define symbolic test cases and the structure of the search algorithm.

Symbolic Test Case. Symbolic test cases are defined by the following grammar:

$$S ::= \alpha^{\mathsf{int}} \mid \alpha^{\mathsf{string}} \mid c(S_1, S_2) \mid \lambda x.S \mid x \mid S_1 \oplus S_2 \mid S_1\hat{\ }S_2 \mid \square^l \tag{2}$$

Unlike concrete test cases defined in (1), symbolic test cases represent primitive values (integers and strings) by symbols, where we distinguish integer-typed (α^{int}) and string-typed (α^{string}) symbols. $\square^l$ is a hole (whose label is l), a placeholder that can be filled with an expression during enumerative search. Note that holes do not appear in the final symbolic test cases; they only appear during the search algorithm. Let Lab be the set of labels possibly associated with holes.

Algorithm 1 Our Counter-Example Generation Algorithm

```
Input: A reference program $P_1 = (f, x, E_1)$ and a test program $P_2 = (f, x, E_2)$
Output: A counter-example $i \in I$
1: $W \leftarrow \{(\square^l, [l \mapsto \tau_i], [], [])\}$
2: repeat
3:     $(s, \Upsilon, \Gamma, \Delta) \leftarrow \mathsf{Choose}(W)$
4:     $W \leftarrow W \setminus \{(s, \Upsilon, \Gamma, \Delta)\}$
5:     if $s$ does not have holes then
6:         $M \leftarrow \mathsf{Verifier}(P_1, P_2, s)$
7:         if $M \neq \emptyset$ then
8:             $i \leftarrow M(s)$
9:             if $\mathsf{CounterExample}(P_1, P_2, i)$ then
10:                return $i$
11:    else
12:        $W \leftarrow W \cup \mathsf{Generator}(s, \Upsilon, \Gamma, \Delta)$
13: until timeout
```

State Space. Our search algorithm is defined over the space of symbolic test cases defined in (2). Let *State* be the state space. A state is a quadruple $(s, \Upsilon, \Gamma, \Delta)$, where $s \in S$ is a symbolic test case that may include holes, and others $(\Upsilon, \Gamma, \Delta)$ are auxiliary information to enable efficient type-directed search. $\Upsilon : Lab \to \tau$ maps labels of holes to their types, $\Gamma : Lab \to (Id \to \tau)$ associates a type environment with each hole, and $\Delta \in Subst : \tau \to \tau$ is a substitution that maps each type variable to its type. We write $\mathsf{dom}(f)$ for the domain of function f. In particular, $\mathsf{dom}(\Gamma(l))$ denotes the set of variables that can be used when we synthesize expressions for the hole whose label is l.

Algorithm Structure. Algorithm 1 describes our counter-example generation algorithm. Given a reference program $P_1 = (f, x, E_1)$ and a test program $P_2 = (f, x, E_2)$, it aims to find a concrete test case $i \in I$ that produces different results on P_1 and P_2. We assume that the type of the counter-example is τ_i, which can be easily obtained by running a standard type inference algorithm on the reference program. The algorithm maintains a workset W. At line 1, it initializes the workset with the initial state $(\square^l, [l \mapsto \tau_i], [], [])$: a symbolic test case is a single hole $\square^l$ with fresh label l, Υ maps the label to the type τ_i of the test case, and Γ and Δ are initially empty. At each iteration of the loop, the algorithm selects a state $(s, \Upsilon, \Gamma, \Delta)$ (line 3). The selection is done based on the size of a symbolic test case:

$$\mathsf{Choose}(W) = \operatorname*{argmin}_{(s,\Upsilon,\Gamma,\Delta)\in W} \mathsf{size}(s),$$

where $\mathsf{size}(s)$ denotes the number of nodes in the syntax tree of s. Our algorithm prefers to find the smallest possible test cases, which is important to generate concise and therefore helpful test cases (see Section 6.1). When the current test case s includes holes (line 11), the algorithm continues searching by updating the workset W with the next states obtained by Generator (line 12). Otherwise, if s is complete, Verifier checks whether s can be a counter-example or not (line 6). When Verifier succeeds to find a counter-example, it computes a model (M) that maps symbols in s to concrete values (line 6). When Verifier fails, the model is empty ($\emptyset$). When a counter-example is found, we obtain the concrete test case i by concretizing the symbolic test case s with the model (line 8). At line 9, we check if i is a genuine counter-example ($\mathsf{CounterExample}(P_1, P_2, i)$) and if so the algorithm terminates with i as an output. The algorithm repeats the procedure described above until it expires the given time budget.

$$\frac{\Delta' = \mathsf{unify}(\Upsilon(l), \mathsf{int}, \Delta) \quad \text{new } \alpha^{\mathsf{int}}}{\langle \square^{l}, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle \alpha^{\mathsf{int}}, \Delta'(\Upsilon), \Delta'(\Gamma), \Delta'\rangle}\ \textsc{E-Num}$$

$$\frac{\Delta' = \mathsf{unify}(\Upsilon(l), \mathsf{string}, \Delta) \quad \text{new } \alpha^{\mathsf{string}}}{\langle \square^{l}, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle \alpha^{\mathsf{string}}, \Delta'(\Upsilon), \Delta'(\Gamma), \Delta'\rangle}\ \textsc{E-Str}$$

$$\frac{c \in C \quad \Lambda(c) = (\tau_1 * \tau_2) \to T \quad \Delta' = \mathsf{unify}(\Upsilon(l), T, \Delta) \quad \text{new } l_1, l_2}{\langle \square^{l}, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle (c(\square^{l_1}, \square^{l_2})), \Delta'(\Upsilon[l_i \mapsto \tau_i]_{i=1}^{2}), \Delta'(\Gamma[l_i \mapsto \Gamma(l)]_{i=1}^{2}), \Delta'\rangle}\ \textsc{E-Cnstr}$$

$$\frac{\Delta' = \mathsf{unify}(\Upsilon(l), t_1 \to t_2, \Delta) \quad \text{new } t_1, t_2, l'}{\langle \square^{l}, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle (\lambda x.\square^{l'}), \Delta'(\Upsilon[l' \mapsto t_2]), \Delta'(\Gamma[l' \mapsto \Gamma(l)[x \mapsto t_1]]), \Delta'\rangle}\ \textsc{E-Fun}$$

$$\frac{x \in \mathsf{dom}(\Gamma(l)) \quad \Delta' = \mathsf{unify}(\Upsilon(l), \Gamma(l)(x), \Delta)}{\langle \square^{l}, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle x, \Delta'(\Upsilon), \Delta'(\Gamma), \Delta'\rangle}\ \textsc{E-Var}$$

$$\frac{\mathsf{dom}(\Gamma(l)) \neq \emptyset \quad \Delta' = \mathsf{unify}(\Upsilon(l), \mathsf{int}, \Delta) \quad \text{new } l_1, l_2}{\langle \square^{l}, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle (\square^{l_1} \oplus \square^{l_2}), \Delta'(\Upsilon[l_1 \mapsto \mathsf{int}, l_2 \mapsto \mathsf{int}]), \Delta'(\Gamma[l_1 \mapsto \Gamma(l), l_2 \mapsto \Gamma(l)]), \Delta'\rangle}\ \textsc{E-Binop}$$

$$\frac{\mathsf{dom}(\Gamma(l)) \neq \emptyset \quad \Delta' = \mathsf{unify}(\Upsilon(l), \mathsf{string}, \Delta) \quad \text{new } l_1, l_2}{\langle \square^{l}, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle (\square^{l_1}\,\hat{}\,\square^{l_2}), \Delta'(\Upsilon[l_1 \mapsto \mathsf{string}, l_2 \mapsto \mathsf{string}]), \Delta'(\Gamma[l_1 \mapsto \Gamma(l), l_2 \mapsto \Gamma(l)]), \Delta'\rangle}\ \textsc{E-Concat}$$

$$\frac{\langle s_1, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle s_1', \Upsilon', \Gamma', \Delta'\rangle}{\langle c(s_1, s_2), \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle c(s_1', s_2), \Upsilon', \Gamma', \Delta'\rangle} \qquad \frac{\langle s_2, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle s_2', \Upsilon', \Gamma', \Delta'\rangle}{\langle c(s_1, s_2), \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle c(s_1, s_2'), \Upsilon', \Gamma', \Delta'\rangle}$$

$$\frac{\langle s, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle s', \Upsilon', \Gamma', \Delta'\rangle}{\langle (\lambda x.s), \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle (\lambda x.s'), \Upsilon', \Gamma', \Delta'\rangle}$$

$$\frac{\langle s_1, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle s_1', \Upsilon', \Gamma', \Delta'\rangle}{\langle (s_1 \oplus s_2), \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle (s_1' \oplus s_2), \Upsilon', \Gamma', \Delta'\rangle} \qquad \frac{\langle s_2, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle s_2', \Upsilon', \Gamma', \Delta'\rangle}{\langle (s_1 \oplus s_2), \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle (s_1 \oplus s_2'), \Upsilon', \Gamma', \Delta'\rangle}$$

$$\frac{\langle s_1, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle s_1', \Upsilon', \Gamma', \Delta'\rangle}{\langle (s_1\,\hat{}\,s_2), \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle (s_1'\,\hat{}\,s_2), \Upsilon', \Gamma', \Delta'\rangle} \qquad \frac{\langle s_2, \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle s_2', \Upsilon', \Gamma', \Delta'\rangle}{\langle (s_1\,\hat{}\,s_2), \Upsilon, \Gamma, \Delta\rangle \rightsquigarrow \langle (s_1\,\hat{}\,s_2'), \Upsilon', \Gamma', \Delta'\rangle}$$

Fig. 6. Transition relation for symbolic test case generation

The key components of the algorithm are Generator and Verifier, which will be described in Sections 5.2 and 5.3, respectively.

5.2 Generator

Generator takes a state $(s, \Upsilon, \Gamma, \Delta)$ and produces a set of next states that immediately follow the given state. To improve the efficiency, we adopt the type-directed search [Feser et al. 2015; Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016], which avoids to explore ill-typed test cases. We define $\mathsf{Generator}(s, \Upsilon, \Gamma, \Delta)$ as follows:

$$\mathsf{Generator}(s, \Upsilon, \Gamma, \Delta) = \{(s', \Upsilon', \Gamma', \Delta') \mid (s, \Upsilon, \Gamma, \Delta) \rightsquigarrow (s', \Upsilon', \Gamma', \Delta')\},$$

where $(\rightsquigarrow) \subseteq \mathit{State} \times \mathit{State}$ is the type-directed transition relation between states.

Figure 6 defines the transition relation $(\rightsquigarrow)$ as a set of inference rules. In the definition, we assume the standard unifier $\mathsf{unify} : \tau \times \tau \times \mathit{Subst} \to \mathit{Subst}$ and write $\Delta(\Upsilon)$ and $\Delta(\Gamma)$ for the results of applying the substitution Δ to the type variables in type environments of Υ and Γ.

The rules are either base cases (named) or inductive cases (unnamed), where the base cases actually describe how holes get replaced in a single-step transition. The rules E-Num and E-Str describe the cases when holes get replaced by symbols. For example, consider the E-Num rule:

$$\frac{\Delta' = \mathsf{unify}(\Upsilon(l), \mathsf{int}, \Delta) \quad \text{new } \alpha^{\mathsf{int}}}{\langle \square^l, \Upsilon, \Gamma, \Delta \rangle \rightsquigarrow \langle \alpha^{\mathsf{int}}, \Delta'(\Upsilon), \Delta'(\Gamma), \Delta' \rangle} \text{ E-NUM}$$

which indicates that we can replace a hole ($\square^l$) by the integer-typed symbol (α^{int}) when the type of the hole ($\Upsilon(l)$) is int (that is, $\Upsilon(l)$ and int can be unified with the current substitution Δ). This way, our search algorithm produces symbolic test cases that involve symbols instead of involving integer constants. Here (and henceforth), we assume that the condition $\Delta' = \mathsf{unify}(\Upsilon(l), \mathsf{int}, \Delta)$ implies that unify does not fail. If it fails, the condition does not hold and therefore the E-Num rule does not apply. The E-Str rule is similar. According to the rules E-Cnstr and E-Fun, a hole may expand into a constructor (E-Cnstr) or a function (E-Fun). In the E-Cnstr rule, recall that C denotes the set of constructors defined in the program and Λ associates constructors with their type information. The E-Var rule shows that a hole may get replaced by variable x if x is available at the current location ($x \in \mathsf{dom}(\Gamma(l))$) and its type can be unified with the hole type. The rules E-Binop and E-Concat are used to expand holes with arithmetic and string operations, respectively. Note that these two rules are only used in function bodies, which is enforced by the condition $\mathsf{dom}(\Gamma(l)) \neq \emptyset$ that some bound variables (formal parameters) must be available at the location l.

5.3 Verifier

Verifier takes a reference program P_1, a test program P_2, and a symbolic test case s (without holes). Then, it checks whether the symbolic test case s can be a counter-example that causes P_1 and P_2 to behave differently. If it succeeds to find such a counter-example, Verifier produces a model, an assignment of symbols in s to concrete values. To do so, it performs bounded symbolic execution and validates the resulting verification condition.

Bounded Symbolic Execution. Verifier first runs $P_1 = (f, x, E_1)$ and $P_2 = (f, x, E_2)$ with the symbolic input s to get the *symbolic summaries* Φ_1 and Φ_2 of P_1 and P_2, respectively:

$$\Phi_1 = \mathsf{SymExec}((f, x, E_1), s), \quad \Phi_2 = \mathsf{SymExec}((f, x, E_2), s),$$

where the function $\mathsf{SymExec}((f, x, E), s)$ computes a symbolic summary obtained by running the program (f, x, E) symbolically with the symbolic test case s.

Let us first define the symbolic summary. A symbolic summary $\Phi \in \mathit{Summary} = \wp(\widehat{\mathit{Path}} \times \widehat{\mathit{Val}})$ is a set of guarded values ($\widehat{\mathit{Path}} \times \widehat{\mathit{Val}}$), where a guarded value is a pair of a path condition ($\widehat{\mathit{Path}}$) and a symbolic value ($\widehat{\mathit{Val}}$). Symbolic values are defined as follows:

$$\widehat{\mathit{Val}} = \mathit{Symbol} + \mathbb{Z} + \mathbb{S} + \widehat{\mathit{Cnstr}} + \widehat{\mathit{Closure}} + \widehat{\mathit{RecClosure}} + (\widehat{\mathit{Val}} \oplus \widehat{\mathit{Val}}) + (\widehat{\mathit{Val}}\,\hat{}\,\widehat{\mathit{Val}}) + \{\bot\}.$$

A symbolic value is either a symbol (Symbol), integer/string constants ($\mathbb{Z}$, $\mathbb{S}$), a symbolic constructor ($\widehat{\mathit{Cnstr}} = \mathit{Id} \times \widehat{\mathit{Val}}^*$), symbolic closures ($\widehat{\mathit{Closure}} = \mathit{Id} \times E \times \widehat{\mathit{Env}}$, $\widehat{\mathit{RecClosure}} = \mathit{Id} \times \mathit{Id} \times E \times \widehat{\mathit{Env}} \times \mathbb{N}$), or symbolic binary operations ($\widehat{\mathit{Val}} \oplus \widehat{\mathit{Val}}$, $\widehat{\mathit{Val}}\,\hat{}\,\widehat{\mathit{Val}}$). A symbolic environment $\widehat{\rho} \in \widehat{\mathit{Env}} = \mathit{Id} \to \wp(\widehat{\mathit{Path}} \times \widehat{\mathit{Val}})$ maps variables to the set of guarded values that the variables may have. Note that the symbolic recursive closure ($\widehat{\mathit{RecClosure}}$) contains an additional non-negative integer component ($\mathbb{N}$) for bounded symbolic execution, which denotes the number of the remaining applications of the recursive function to avoid non-termination. A path condition ($\widehat{\mathit{Path}}$) is a symbolic equality ($\widehat{\mathit{Val}} = \widehat{\mathit{Val}}$), the negation of a path condition ($\neg\widehat{\mathit{Path}}$), or the conjunction of path conditions ($\widehat{\mathit{Path}} \wedge \widehat{\mathit{Path}}$). We write $\top$ for the initial (empty) path condition.

$$\frac{}{\widehat{\rho}, \pi \vdash^k n \Rightarrow \{(\pi, n)\}} \quad \frac{}{\widehat{\rho}, \pi \vdash^k s \Rightarrow \{(\pi, s)\}} \quad \frac{}{\widehat{\rho}, \pi \vdash^k x \Rightarrow \widehat{\rho}(x)}$$

$$\frac{}{\widehat{\rho}, \pi \vdash^k \alpha^{\mathsf{int}} \Rightarrow \{(\pi, \alpha^{\mathsf{int}})\}} \quad \frac{}{\widehat{\rho}, \pi \vdash^k \alpha^{\mathsf{string}} \Rightarrow \{(\pi, \alpha^{\mathsf{string}})\}} \quad \frac{}{\widehat{\rho}, \pi \vdash^k \lambda x.E \Rightarrow \{(\pi, (x, E, \widehat{\rho}))\}}$$

$$\frac{\widehat{\rho}, \pi \vdash^k E_1 \Rightarrow \Phi_1 \quad \widehat{\rho}, \pi \vdash^k E_2 \Rightarrow \Phi_2}{\widehat{\rho}, \pi \vdash^k E_1 \oplus E_2 \Rightarrow \{(\pi_1 \wedge \pi_2, \widehat{v_1} \oplus \widehat{v_2}) \mid (\pi_1, \widehat{v_1}) \in \Phi_1 \wedge (\pi_2, \widehat{v_2}) \in \Phi_2\}}$$

$$\frac{\widehat{\rho}, \pi \vdash^k E_1 \Rightarrow \Phi_1 \quad \widehat{\rho}, \pi \vdash^k E_2 \Rightarrow \Phi_2}{\widehat{\rho}, \pi \vdash^k E_1 \hat{}\, E_2 \Rightarrow \{(\pi_1 \wedge \pi_2, \widehat{v_1} \hat{}\, \widehat{v_2}) \mid (\pi_1, \widehat{v_1}) \in \Phi_1 \wedge (\pi_2, \widehat{v_2}) \in \Phi_2\}}$$

$$\frac{\widehat{\rho}, \pi \vdash^k E_1 \Rightarrow \Phi_1 \quad \widehat{\rho}, \pi \vdash^k E_2 \Rightarrow \Phi_2}{\widehat{\rho}, \pi \vdash^k c(E_1, E_2) \Rightarrow \{(\pi_1 \wedge \pi_2, c(\widehat{v_1}, \widehat{v_2})) \mid (\pi_1, \widehat{v_1}) \in \Phi_1 \wedge (\pi_2, \widehat{v_2}) \in \Phi_2\}}$$

$$\frac{\widehat{\rho}, \pi \vdash^k E_1 \Rightarrow \Phi_1 \quad \widehat{\rho}[x \mapsto \Phi_1], \pi \vdash^k E_2 \Rightarrow \Phi_2}{\widehat{\rho}, \pi \vdash^k \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \Rightarrow \Phi_2} \quad \frac{\widehat{\rho}[f \mapsto \{(\pi, (f, x, E_1, \widehat{\rho}, k))\}], \pi \vdash^k E_2 \Rightarrow \Phi_2}{\widehat{\rho}, \pi \vdash^k \mathsf{let\ rec}\ f(x) = E_1\ \mathsf{in}\ E_2 \Rightarrow \Phi_2}$$

$$\frac{\widehat{\rho}, \pi \vdash^k E_1 \Rightarrow \Phi_1 \quad \widehat{\rho}, \pi \vdash^k E_2 \Rightarrow \Phi_2}{\widehat{\rho}, \pi \vdash^k E_1\ E_2 \Rightarrow \bigcup_{(\pi_1, \widehat{v_1}) \in \Phi_1} \widehat{\mathsf{Call}}((\pi_1, \widehat{v_1}), \Phi_2)}$$

$$\frac{\widehat{\rho}, \pi \vdash^k E_0 \Rightarrow \Phi_0}{\widehat{\rho}, \pi \vdash^k \mathsf{match}\ E_0\ \mathsf{with}\ \overline{p_i \longrightarrow E_i}^n \Rightarrow \bigcup_{(\pi_0, \widehat{v_0}) \in \Phi_0} \widehat{\mathsf{Branch}}(\widehat{\rho}, (\pi_0, \widehat{v_0}), \{(p_i, E_i) \mid \widehat{\mathsf{match}}(p_i, \widehat{v_0}) \wedge i \in [1, n]\})}$$

Fig. 7. Semantics of bounded symbolic execution

Now we define symbolic executor, $\widehat{\mathcal{E}}[\![E]\!]^k : \widehat{Env} \longrightarrow \widehat{Path} \longrightarrow Summary$, which computes the symbolic summary of the input expression (E) by evaluating it under the current environment and path condition. The number k denotes the predetermined loop bound (generated by recursive functions) that is assumed to be given beforehand. Fig 7 shows the evaluation rules for our symbolic execution, where we use the notation $\widehat{\rho}, \pi \vdash^k E \Rightarrow \Phi$ to denote $\widehat{\mathcal{E}}[\![E]\!]^k(\widehat{\rho}, \pi) = \Phi$. For the base cases, it works similar to the standard concrete semantics, except that it returns a singleton set of a guarded value which maintains the current path condition (When $E = x$, the resulting set may not be singleton depending on the environment). Note that the rules consider the cases when expressions are symbols ($\alpha^{\mathsf{int}}, \alpha^{\mathsf{string}}$) even though the expressions defined in Section 4 do not have symbols. This is because the definition of symbolic test cases in (2) has symbols and therefore evaluating a program with a functional input value may involve symbols. When evaluating a binary operation or a constructor, it first computes symbolic summaries of two subexpressions and combines their path conditions and symbolic values to gather all possible outputs that the expression may have. To evaluate a let-expression (let $x = E_1$ in E_2), it evaluates E_2 after extending the given environment for the bound variable x and its values Φ_1. If the symbolic executor encounters a recursive function definition (let rec $f(x) = E_1$ in E_2), it updates the current environment by making f point to a singleton with a guarded value $\{(\pi, (f, x, E_1, \widehat{\rho}, k))\}$ whose symbolic value is a recursive closure with the predefined loop bound k. The rules in Fig 7 do not explicitly describe how our symbolic executor deals with runtime exceptions. If a runtime exception occurs while executing the program E in the path π, the executor no longer continues the current execution and returns $\{(\pi, \bot)\}$, indicating that the path π can reach an error state.

When the symbolic executor encounters a function application ($E_1\ E_2$), it evaluates E_1 to obtain the set Φ_1 of possible functions. Then, it considers each functional value $(\pi_1, \widehat{v_1}) \in \Phi_1$ and calls it with the actual argument (Φ_2). The function call is performed by the following function:

$$\widehat{\mathsf{Call}}((\pi_1, \widehat{v_1}), \Phi_2) =$$
$$\begin{cases} \widehat{\mathcal{E}}[\![E]\!]^k(\widehat{\rho}'[x \mapsto \Phi_2]', \pi_1) & \text{if } \widehat{v_1} = (x, E, \widehat{\rho}') \\ \widehat{\mathcal{E}}[\![E]\!]^k(\widehat{\rho}'[f \mapsto \{(\pi_1, (f, x, E, \widehat{\rho}', k' - 1))\}, x \mapsto \Phi_2], \pi_1) & \text{if } \widehat{v_1} = (f, x, E, \widehat{\rho}', k') \wedge k' > 0 \\ \{(\pi_1, \bot)\} & \text{if } \widehat{v_1} = (f, x, E, \widehat{\rho}', 0). \end{cases}$$

If it calls a non-recursive function ($\widehat{v_1} = (x, E, \widehat{\rho}')$), it executes the body normally with the new environment ($\widehat{\rho}'[x \mapsto \Phi_2]$). If a recursive function that has not yet consumed all budgets on the loop bound is invoked ($\widehat{v_1} = (f, x, E, \widehat{\rho}', k') \wedge k' > 0$), its body is executed under the stored environment $\widehat{\rho}'$ with the argument and function being extended. Note that the loop bound decreases by one whenever the function is called. When a recursive function with no remaining execution count is called ($\widehat{v_1} = (f, x, E, \widehat{\rho}', 0)$), the symbolic executor terminates the current execution by returning $\{(\pi_1, \bot)\}$, which means that this function is no longer callable.

The last rule for the match expressions is most involved. We first evaluate the expression E_0 to get the corresponding summary Φ_0. Then, we consider each $(\pi_0, \widehat{v_0})$ in Φ_0. Because the symbolic value $\widehat{v_0}$ may match multiple patterns, the executor should compute symbolic summaries of all possible branches. Let B be the set of all possible matched branches with the symbolic value $\widehat{v_o}$:

$$B = \{(p_i, E_i) \mid \widehat{\mathsf{match}}(p_i, \widehat{v_0}) \wedge i \in [1, n]\}.$$

where $\widehat{\mathsf{match}}(p, \widehat{v})$ is true if $\widehat{v}$ has a chance of matching p, for which we use a simple analysis based on syntax and types. For example, we can safely conclude that $\widehat{\mathsf{match}}(p, \widehat{v})$ is true if $\widehat{v}$ and p are syntactically equivalent or they have the same type. Next, our symbolic executor joins all possible symbolic summaries from each branch in B with $(\pi_0, \widehat{v_0})$ using the function $\widehat{\mathsf{Branch}}(\widehat{\rho}, (\pi_0, \widehat{v_0}), B)$ defined as follows:

$$\widehat{\mathsf{Branch}}(\widehat{\rho}, (\pi_0, \widehat{v_0}), B) = \begin{cases} \bigcup_{(p_i, E_i) \in B} \widehat{\mathcal{E}}[\![E_i]\!]^k(\mathsf{bind}(\widehat{\rho}, p_i, (\pi_0, \widehat{v_0})), \mathsf{newpc}(p_i, (\pi_0, \widehat{v_0}))) & \text{if } |B| \geq 1 \\ \{(\pi_0, \bot)\} & \text{otherwise.} \end{cases}$$

If there exists at least one matched branch (i.e., $|B| \geq 1$), it combines all symbolic summaries of all branches by executing them under the updated symbolic environment $\mathsf{bind}(\widehat{\rho}, p_i, (\pi_0, \widehat{v_0}))$. The function $\mathsf{bind}(\widehat{\rho}, p, (\pi, \widehat{v})$ updates the symbolic environment $\widehat{\rho}$ with given pattern p and guarded value $(\pi, \widehat{v})$:

$$\mathsf{bind}(\widehat{\rho}, p, (\pi, \widehat{v})) = \begin{cases} \widehat{\rho}[x \mapsto \{(\pi, \widehat{v})\}] & \text{if } p = x \\ \mathsf{bind}(\mathsf{bind}(\widehat{\rho}, p_1, (\pi, \widehat{v_1})), p_2, (\pi, \widehat{v_2})) & \text{if } p = c(p_1, p_2) \wedge \widehat{v} = c(\widehat{v_1}, \widehat{v_2}) \\ \widehat{\rho} & \text{otherwise.} \end{cases}$$

When executing each matched branch, the executor updates the current path condition as follows:

$$\mathsf{newpc}(p_i, (\pi_0, \widehat{v_0})) = \pi_0 \wedge \mathsf{gen\_pc}(p_i, \widehat{v_0}) \wedge (\bigwedge_{j<i} \neg \mathsf{gen\_pc}(p_j, \widehat{v_0}))$$

which indicates that the value $\widehat{v_0}$ is matched with the current pattern ($\mathsf{gen\_pc}(p_i, \widehat{v_0})$) and unmatched with the previous patterns ($\bigwedge_{j<i} \neg \mathsf{gen\_pc}(p_j, \widehat{v_0})$). The function $\mathsf{gen\_pc}(p, \widehat{v})$ that generates a new

path condition is defined as follows:

$$\mathsf{gen\_pc}(p, \widehat{v}) = \begin{cases} n = \widehat{v} & \text{if } p = n \\ s = \widehat{v} & \text{if } p = s \\ \mathsf{gen\_pc}(p_1, \widehat{v_1}) \land \mathsf{gen\_pc}(p_2, \widehat{v_2}) & \text{if } p = c(p_1, p_2) \land \widehat{v} = c(\widehat{v_1}, \widehat{v_2}) \\ \top & \text{if } p = x \lor p = \_. \end{cases}$$

If there are no patterns matched with $\widehat{v_0}$, it returns $\{(\pi_0, \bot)\}$ which means pattern matching failure.

Note that, the rules in Fig 7 do not consider a symbolic function and a symbolic data type as an input because the symbolic test cases generated by Generator include only primitive integer or string type symbols. It enables our symbolic executor to easily handle several features of functional programming language such as higher-order function and inductive data type. Furthermore, the absence of symbolic functions and symbolic data types allows Verifier to generate simple verification conditions that can be easily solved by an off-the-shelf SMT solver.

Finally, we define the function $\mathsf{SymExec}((f, x, E), s)$ as follows:

$$\mathsf{SymExec}((f, x, E), s) = \widehat{\mathcal{E}}[\![E]\!]^k(\widehat{\rho_0}, \top),$$

where the initial environment $\widehat{\rho_0}$ is defined as follows:

$$\widehat{\rho_0} = [x \mapsto \widehat{\mathcal{E}}[\![s]\!]^k([], \top), f \mapsto \{(\top, (f, x, E, [], k))\}].$$

On top of the symbolic execution described so far, we apply an optimization technique. Recall that the semantics in Fig 7 always decreases the execution count of a recursive function whenever it is invoked. However, we found that doing so sometimes loses too much information about the program behavior. Thus, we use a simple optimization technique that does not decrease the count in a "deterministic" context. The intuition is that some paths can be uniquely determined during the symbolic execution as the symbolic inputs contain "concrete" parts.

Suppose that the symbolic executor runs the following program with a symbolic input `Add (Add (Int` α_1^{int}`, Int` α_2^{int}`), Add (Int` α_3^{int}`, Int` α_4^{int}`))`:

```
type exp = Int of int | Add of exp * exp | Sub of exp * exp
let rec eval e =
  match e with
  | Int n -> n
  | Add (e1, e2) -> (eval e1) + (eval e2)
  | Sub (e1, e2) -> (eval e1) - (eval e2)
```

In this example, even if the input contains symbols (i.e., α_1^{int}), the input matches only with the first branch (`Int n`) or the second branch (`Add (e1, e2)`) at each iteration. For this reason, the symbolic executor can conclude that the path is "deterministic", and therefore it does not decrease the execution count for the recursive function call (i.e., `eval e1` and `eval e2`). When aggressively reducing the execution count, the symbolic executor fails to run the above program if the execution count is set to less than two, while the optimization technique allows the symbolic executor to succeed in the same environment. Thus, we introduce this optimization technique to help the symbolic executor gather more program behaviors without increasing the loop bound.

Validation. Using the symbolic summaries Φ_1 and Φ_2 computed by the symbolic executor, we generate a counter-example by checking the validity of the formula ϕ:

$$\phi \equiv \bigwedge_{(\pi_1, \widehat{v_1}) \in \Phi_1} \Big(\pi_1 \implies \big(\bigvee_{(\pi_2, \widehat{v_2}) \in \Phi_2} \pi_2 \land (\widehat{v_1} = \widehat{v_2})\big)\Big).$$

The formula ϕ holds if and only if for every feasible path π_1 of the reference program, there exists a feasible path π_2 in the test program such that the outcomes of the two programs are equivalent ($\widehat{v_1} = \widehat{v_2}$). Intuitively, this means that we regard the test program as to be correct if it covers all the possible behaviors of the reference program. We can check the validity of ϕ by checking the satisfiability of $\neg\phi$ with an off-the-shelf SMT solver. When $\neg\phi$ is unsatisfiable (i.e., ϕ is valid), we conclude that the current symbolic test case cannot be a counter-example. Otherwise (i.e., ϕ is invalid), we conclude that the symbolic test case can be a counter-example, and Verifier returns a model M of $\neg\phi$. Algorithm 1 uses the model M to convert the symbolic test case into a concrete one.

Note that the decision made above can be unreliable beyond the given loop bound, which is why we verify the generated test case with CounterExample in Algorithm 1 (line 9). Suppose ϕ is invalid and we have a model M of $\neg\phi$. The concretized test case ($M(s)$) may not be an actual counter-example in cases when the symbolic executor fails to collect the relevant execution paths beyond the loop bound in the test program. Further, the validity of ϕ does not always imply the correctness of the test program because the symbolic executor may fail to collect some behaviors of the reference program beyond the given loop bound. To minimize these undesired situations in practice, we apply the optimization technique mentioned above (retaining the execution count of function in deterministic contexts).

In addition, we observed that off-the-shelf SMT solvers are not very efficient in practice and often require domain-specific engineering for better performance. To alleviate the overhead of the SMT solver, we apply several preprocessing optimizations to identify the obviously false formulas beforehand. For example, a formula that contains a conjunction of two contradictory clauses (e.g., $(\alpha_1 = \alpha_2) \land \neg(\alpha_1 = \alpha_2)$) can be easily identified as false, and we statically identify such formulas as false without passing to the SMT solver.

Running Example. Let us finish this section with a running example describing how Verifier works for a program taking a function and a user-defined data type as input. Suppose that we have a buggy (left) and a correct (right) implementations of a function `map` which applies a given function to all elements of a user-defined data type `lst` which consists of an integer (`Int`) and concatenation of two lists (`App`):

```
let rec map f l =
  match l with
  | Int n -> if n > 0 then Int (f n) else Int n
  | App (a, b) -> App (map f a, map f b)
```

```
let rec map f l =
  match l with
  | Int n -> Int (f n)
  | App (a, b) -> App (map f a, map f b)
```

Assume that Generator generates (`fun x -> x +` α_1^{int}) and (`Int` α_2^{int}) for each argument of the function `map`. Verifier first runs two programs with these symbolic test cases. When the symbolic executor runs the buggy program, it encounters a match expression and executes the first branch. When executing the first branch, two paths are considered: $(\alpha_2^{\text{int}} > 0)$ and $(\alpha_2^{\text{int}} \leq 0)$. We get $\texttt{Int}(\alpha_2^{\text{int}} + \alpha_1^{\text{int}})$ for the former by applying the function (`fun x -> x +` α_1^{int}) to α_2^{int}, and `Int` α_2^{int} for the latter. As a result, we get a symbolic summary with two guarded values, $\{(\alpha^{\text{int}} > 0, \texttt{Int}(\alpha_2^{\text{int}} + \alpha_1^{\text{int}})), (\alpha^{\text{int}} \leq 0, \texttt{Int}\ \alpha_2^{\text{int}})\}$, for the buggy program. Similarly, we obtain a symbolic summary $\{(\top, \texttt{Int}(\alpha_2^{\text{int}} + \alpha_1^{\text{int}}))\}$ for the solution program.

With these two symbolic summaries, Verifier constructs the following verification condition:

$$\begin{aligned}\top \Rightarrow \quad & ((\alpha_2^{\text{int}} > 0) \land (\texttt{Int}(\alpha_2^{\text{int}} + \alpha_1^{\text{int}}) = \texttt{Int}(\alpha_2^{\text{int}} + \alpha_1^{\text{int}}))) \lor \\ & ((\alpha_2^{\text{int}} \leq 0) \land (\texttt{Int}(\alpha_2^{\text{int}} + \alpha_1^{\text{int}}) = \texttt{Int}\ \alpha_2^{\text{int}})).\end{aligned}$$

Using an off-the-shelf SMT solver, Verifier easily computes a model which falsifies the verification condition. Suppose that a model $[\alpha_1^{\text{int}} \mapsto 1, \alpha_2^{\text{int}} \mapsto 0]$ is obtained, then we get concrete test cases `(fun x -> x + 1)` and `(Int 0)` by substituting each symbol in the symbolic test cases `(fun x -> x +` α_1^{int}`)` and `(Int` α_2^{int}`)`.

6 EVALUATION

In this section, we experimentally evaluate our technique. We aim to answer the following research questions:

- **Effectiveness**: How effectively can our counter-example generation algorithm detect erroneous submissions? (Section 6.1)
- **Comparison with property-based testing**: Can our technique find counter-examples more effectively than property-based testing? (Section 6.2)
- **Comparison with simpler approaches**: Is our hybrid approach more effective than simpler approaches such as pure enumerative or symbolic approaches? (Section 6.3)
- **Usefulness in automatic program repair**: Can our technique enhance automatic program repair systems by preventing them from generating test-suite-overfitted patches? (Section 6.4)

We implemented our approach in a tool, TestML, with about 6500 lines of OCaml code. We set the execution count (k) to 6, which is used in bounded symbolic execution to ensure termination (Section 5.3). We used Z3 [De Moura and Bjørner 2008] to check the validity of the formulas that result from symbolic execution (Section 5.3), and set the timeout for each Z3 invocation to 50 milliseconds. All the experiments were conducted on an iMac with Intel i5 CPU and 16GB memory.

6.1 Effectiveness

In this section, we demonstrate that our technique is superior to manually-designed test cases when detecting logical errors in real submissions.

Experimental Setting. We collected 4,060 compilable submissions without syntax and type errors from 10 exercises used in our functional programming course. The exercises range from introductory problems to more advanced ones requiring various user-defined constructors or functions as inputs (Table 1).

The manually-designed test suite consists of 10 input-output test cases for each problem. All test cases have been carefully designed to cover diverse behaviors of programs; they have been continually refined in order to better grade student submissions over the last three years. As an example, Appendix A shows the test cases used for grading submissions for Problem 10. To verify the quality of the test suite, we measured the expression coverage of all submissions and found that our test suite achieved more than 90% coverage for most of the submissions (3689/4060). It indicates that our test cases are sufficiently well-designed to cover various behaviors of a wide variety of programs.

We set the timeout for TestML to 60 seconds per program. If it fails to generate a counter-example within the time limit, it judges that the given program is error free. According to our experience, the 60-second time limit is sufficient enough for finding a single counter-example.

Result. Table 1 shows the evaluation result on our data set. For each problem, the table shows the number of erroneous submissions found in various experimental settings. The first sub-column of '# Error Programs' column indicates the number of buggy programs detected by both manually-designed test cases and TestML. The second one shows the number of buggy programs detected only by TestML, and the third is the opposite.

The result demonstrates that TestML is far more effective than human-made test cases in logical error detection. While 543 erroneous programs were detected by both TestML and the human-made test cases, TestML found 88 more errors as shown in the second sub-column of '# Error programs'

Table 1. Comparison with the instructor-generated test cases. 'TestML ✓', 'Manual ✓', 'TestML ✗', and 'Manual ✗' indicate whether an erroneous program is found or not by TestML or the manual test cases, respectively.

No	Problem Description	# Error Programs			
		TestML ✓ Manual ✓	TestML ✓ Manual ✗	TestML ✗ Manual ✓	Total
1	Finding a maximum element in a list	35	10	0	45
2	Filtering a list	5	4	0	9
3	Mirroring a binary tree	9	0	0	9
4	Checking membership in a binary tree	19	0	0	19
5	Computing $\sum_{i=j}^{k} f(i)$ for j, k, and f	32	0	0	32
6	Composing functions	46	3	0	49
7	Adding numbers in user-defined number system	14	4	0	18
8	Evaluating expressions and propositional formulas	105	7	0	112
9	Deciding lambda terms are well-formed or not	116	25	0	141
10	Differentiating algebraic expressions	162	35	0	197
	Total	543	88	0	631

column. Furthermore, there are no errors which are detected only by the human-provided test cases but missed by TestML (the third sub-column of '# Error Programs'). This is remarkable because the test cases are not a strawman; we have refined them several times over the past three years. Despite of this effort, the latest version of our test cases for Problem 10 found only 7 more error programs (162) compared to the oldest one (155). TestML, however, found 42 more programs (197) than the first version of test suite. In conclusion, TestML found 631 erroneous submissions in total, yet the manually-designed test cases only found 543.

The effectiveness of our technique comes from the ability to automatically examine numerous submissions one by one. Because the instructor cannot predict the behaviors of divergent implementations or examine a huge set of programs individually, it is impossible to manually construct a test set which includes all corner cases. We found that the students' submissions are usually very complex to understand and relatively sizable compared to instructor's solution, which makes the manual investigation more difficult. However, as our technique is able to automatically generate a counter-example of each submission without any manual effort, it can detect errors more precisely than manually-designed test cases. We manually confirmed that all erroneous programs newly detected by our technique have actual errors.

Importance of Concise Test Cases. We also analyzed the experimental results qualitatively as well as quantitatively. An interesting observation is that the counter-examples generated by our technique are significantly more concise than the manually-designed test cases. Consider the following erroneous implementation of Problem 10.

```
let rec diff (e, var) =
  match e with
    | Times [hd] -> diff (hd, var)
    | Times (hd::tl) ->
      (match hd with
      | Const a -> Times (hd::[diff (Times tl, var)])
      | Var a -> if (a = var) then Sum (hd::(diff (Times tl, var)::[diff (Times tl, var)]))
                 else Times (hd::[diff (Times tl, var)])
      | _ -> Sum [Times ((diff (hd, var))::tl); Times (hd::[diff (Times tl, var)])]) | ...
```

In this example, the program has an error at line 7. When the head element `hd` in multiplication is the same variable with the given variable `var`, it incorrectly differentiates the multiplication by `Sum (hd::(diff (Times tl, var)::[diff (Times tl, var)]))` (i.e., $(f * g)' = f + g' + g'$). Given this program, TestML generates a test case

`(Times [Var ”x”; Const 0], ”x”)`

as a counter-example to detect such an error within 0.3 seconds. It triggers the error because it is a `Times` list whose head is the same with the given variable. In contrast, the manually-designed test case which detected the error was far more complicated as shown below:

`(Sum [Times [Sum [Var ”x”; Var ”y”]; Times [Var ”x”; Var ”y”]]; Power (”x”, 2)], ”x”).`

This test case contains a lot of uninformative noises that are unnecessary for error detection (e.g., `Sum` or `Power` are not directly related to the error), which makes it hard to understand how the test case causes the error.

As test cases should be helpful to understand how the corner cases are occurred, it is important to make them simple. In other words, an intricate test case hinders users in understanding the behaviors of a program. Typically, human-made test cases tend to be complicated by the desire to cover as many corner cases as possible, and these complex test cases are not ideal for investigating a main cause of a corner case. Thus, capability of writing concise test cases is important for better understandings on the programs, and we believe that our technique is helpful in this area.

6.2 Comparison with Property-Based Testing

In this section, we compare our technique with property-based testing, a well-known approach for testing functional programs.

Experimental Setting. To compare with property-based testing, we used QCheck[2], an OCaml version of QuickCheck [Claessen and Hughes 2000]. QCheck provides various built-in test generators of primitive data types (e.g., integer, boolean, list, etc.) for user convenience, but it requires users to manually build a test generator for other data types. We carefully built them for each problem to use QCheck properly because the inputs of our problems vary in a range from primitive values to diverse user-defined constructors and functions. In addition, we also designed shrinkers which simplifies the results of QCheck since it basically performs the random testing and often generates counter-examples that are too complex to understand. An example of the generator and the shrinker we used for Problem 10 is given in Appendix B.

The benchmark set is similar to the one used in Table 1, yet we excluded problems that require to write higher-order functions (Problem 2, Problem 5, and Problem 6) because testing higher-order functions with QCheck is relatively challenging (see the “Difficulty of Testing Higher-order Functions” paragraph below).

Result. Table 2 shows that TestML outperforms the property-based testing tool in error detection. The columns ‘QCheck1’, ‘QCheck2’, and ‘TestML’ indicate the performance of QCheck without and with test case shrinking, and our tool, respectively. To demonstrate the performance of each technique, we measured the number of detected erroneous programs (#E) and the total amount of time to produce counter-examples (Time) for each problem. Our technique successfully proved that 541 submissions have errors by discovering their counter-examples. On the other hand, QCheck without shrinkers detected only 528 erroneous programs, and 508 otherwise.

[2] https://github.com/c-cube/qcheck

Table 2. Comparison with QCheck. '#E' reports the number of detected erroneous programs and 'Time' reports the total amount of time to produce all the counter-examples.

No	Problem Description	QCheck1		QCheck2		TestML	
		#E	Time	# E	Time	#E	Time
1	Finding a maximum element in a list	45	86.0	38	72.6	45	0.5
3	Mirroring a binary tree	9	0.0	9	0.0	9	0.3
4	Checking membership in a binary tree	19	0.0	19	0.0	19	0.5
7	Adding numbers in user-defined number system	18	0.8	18	0.8	18	0.3
8	Evaluating expressions and propositional formulas	112	3.7	112	10.5	112	6.5
9	Deciding lambda terms are well-formed or not	139	110.4	130	555.8	141	10.4
10	Differentiating algebraic expressions	186	390.1	182	318.6	197	86.6
	Total	528	592.0	508	958.4	541	105.1

The result also shows that TestML is far more efficient than QCheck in time cost. Because QCheck basically performs random testing, it has some difficulties in finding a specific counter-example within a short time. Even it successfully generates a counter-example, QCheck often produces multiple lines of long test cases which are hard to understand. As we mentioned in the paragraph "Importance of Concise Test Cases" in Section 6.1, generating concise test cases is important in logical error detection; thus, it is very natural for programmers to build an additional input shrinker to simplify the test cases generated by QCheck. However, we found that the input shrinker often degrades the performance as it spends more time shortening the test cases. The evaluation results show that QCheck without any shrinkers took about 592.0 seconds in total for generating 528 counter-examples, and 958.4 seconds for 508 counter-examples otherwise. In contrast, TestML generated 541 concise test cases with no needs for shrinking algorithm, and it only took 105.1 seconds in total.

QCheck Requires Significant Manual Effort. The most important point is that our technique outperforms the property-based test generator without any manual effort such as an implementation of a test generator or a shrinker. We found that the performance of QCheck heavily depends on the given generator. For a proper experiment, we have built the generators and shrinkers as well-designed as possible to generate counter-examples in reasonable time. For instance, if a program does not need a large integer value, we designed a generator to only produce the small unsigned integers (e.g. integers between 0 and 100). Without this optimization, we observed that the performance of QuickCheck is seriously degraded. In an extreme case, it failed to detect a single counter-example for some problems even with a 20-minute time budget. Thus, developers should carefully design generators and shrinkers to use QCheck effectively; it, however, requires a lot of effort and intuition to do so. Indeed, to achieve the result of the columns 'QCheck1' and 'QCheck2' in Table 2, we tried several experiments with a number of generators and shrinkers (more than 5 for each problem) and reported the best performance among the several trials. Unlike QCheck, our technique does not require any such human effort for effective testing.

Difficulty of Testing Higher-order Functions. Since QCheck does not support a function generator, we were not able to evaluate it on the three higher-order problems which take a function as an input. To design a test generator and an input shrinker for functional test cases manually is challenging as they require programmers to design grammar for an input function. Some researches focus on addressing the problem of testing higher-order function in property-based testing (e.g., [Koopman and Plasmeijer 2006]).

Unlike QCheck, Haskell QuickCheck can generate random functions by using provided function generators, yet it also has some limitations; the generated function only returns random values without performing any computation. In addition, the generated function is not expressed as a format of arguments and function body, but a mapping relation. For example, if QuickCheck produces a function that returns 1 when it takes 1 as an input and returns 0 otherwise, it represents the function by a mapping relation $[1 \mapsto 1, \_ \mapsto 0]$. To use this mapping as a function, users manually convert it to a proper function format (e.g., a function whose body is a conditional expression). However, our technique is able to produce a function by synthesizing its body automatically. We believe that the capability of function type input generation is an indispensable feature in terms of automatic test case generation for functional programs.

6.3 Comparison with Pure Enumerative and Symbolic Approaches

The key novelty of our approach is that it combines enumerative search and symbolic verification in a novel way that enjoys the benefit of both approaches. In this section, we briefly describe the limitations when each of them is used solely.

While our combined approach does not require the users to provide any testing components, pure enumerative search needs testing components because enumerating all integers and strings is impossible. To make the enumerative search more tractable, we provided limited integer and string components (we used only 1, 2, and 3 as integer components and "x", "y", and "z" as string components). However, despite of these restrictions, the pure enumeration failed to find a counter-example for one of the submission for Problem 10 after 20 minutes. Compared to this result, TestML successfully generated the following counter-example within 58 seconds:

```
(Times [Sum [Var "x"; Const 1; Var "x"]; Var "x"], "x").
```

It shows that our symbolic search algorithm can prune out the large search space of test cases significantly, which is essential to generate a sizable test case.

The pure symbolic approach also has limitations. First, in functional programs, there exist several features like higher-order functions and algebraic data types which are hard to be symbolized and verified by off-the-shelf SMT solvers. Another problem we have experienced is that the extensive use of pattern-matching and recursion in functional programs makes the number of paths explode exponentially. For this reason, the symbolic execution step was not even finished on the most of the submissions. This problem becomes more serious in an introductory programming course. Because many students are not familiar with functional programming, they tend to implement programs with a large number of infeasible paths, which is a significant burden for symbolic execution. To resolve these problems, we do not apply symbolic technique for the non-primitive values that are difficult to symbolize, but easy to be handled by enumerative search. Furthermore, since the symbolic test case includes some concrete parts, it alleviates the path explosion problem by making the symbolic executor only consider the paths which are related with given input instead of executing all paths symbolically.

6.4 Enhancing Automatic Program Repair with Automatic Test Generation

Finally, we show the usefulness of our technique to solve the *test-suite-overfitted patch* problem [Smith et al. 2015] in test-case-based program repair systems.

Counter-Example Guided Repair System. First, we propose a new type of program repair system called *Counter-Example Guided Repair System* which is an enhanced test-case-based repair system using our counter-example generation algorithm. Fig 8 shows the overall workflow of the counter-example guided repair system. Given an incorrect program and a correct program, it

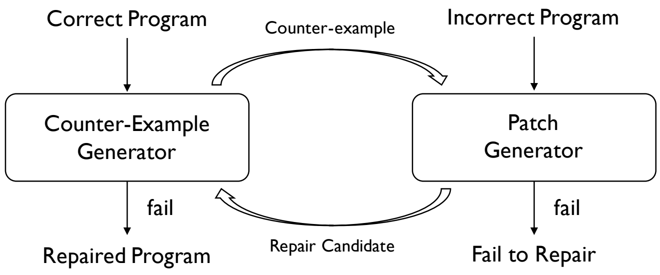

Fig. 8. Overall workflow of Counter-Example Guided Repair System

generates a patch by correcting the error in the given incorrect program and validates the patch by counter-example generation. If the counter-example generator produces a counter-example of the generated repair candidate, the system enriches the test suite by adding the newly discovered counter-example and tries to repair the original error program again. It repeats this procedure until there is no counter-example found by the counter-example generator. In this case, it concludes that the patch is correct and returns it. If the patch generator fails to generate a repair candidate, the system reports that generating a patch is failed.

Experimental Setting. For evaluation, we implemented a counter-example guided repair system based on FixML [Lee et al. 2018b], a state-of-the-art feedback generator for functional programming assignments. As FixML requires a reference implementation to effectively repair a buggy program, we provided a solution program implemented by an instructor as a reference code. The benchmark set is the same with the one used in the Table 1. To run FixML, we provided 10 manually-designed test cases used in Experiment 1 for each problem while the counter-example guided repair system did not take any test cases beforehand. We set the timeouts for both the patch generation and counter-example generation to 60 seconds. To identify the test-suite-overfitted patch problems generated by FixML, we manually confirmed the correctness of all generated patches one by one thoroughly. We classified a patch as correct one when its semantics is the same with the one of the solution code (i.e., it always returns the same output as the correct answer), and did as a test-suite-overfitted patch otherwise.

Result. In Table 3, we present the number of detected error programs (#E), the number of correct patches generated by FixML (#P), the number of test-suite-overfitted patches (#O), and the ratio of successfully repaired programs with respect to the entire programs (Rate). We compare the performance of FixML when using manual-test-cases (Manual Test Suite) with the one enhanced by our counter-example generation algorithm (Our Technique). The result in Table 3 shows that the counter-example guided approach dramatically improved the performance of FixML. The number of test-suite-overfitted patches notably decreased from 58 to 1 as the counter-example guided repair system verifies the correctness of patch candidates by generating counter-examples. As a result, the patch rate eventually increased from 29% (156/543) to 38% (237/631) as well as the total number of patches.

Even if our counter-example guided repair approach significantly reduces the number of incorrect patches, there was still an incorrect patch in Problem 5. Our technique failed to generate a counter-example because its performance depends on SMT solver in symbolic verification step and the given solving time (50 milliseconds) was not enough to solve it. Because the insufficient time is the only reason for failure, this problem can be easily addressed by giving the solver more time. We

Table 3. Enhancement in automatic program repair. The column '#E' indicates the number of detected error programs, '#P' reports the number of correct patches , '#O' shows the number of test-suite-overfitted patches, 'Rate' represents the ratio of correctly patched programs to the erroneous programs.

No	Problem Description	Manual Test Suite				Our Technique			
		#E	#P	#O	Rate	#E	#P	#O	Rate
1	Finding a maximum element in a list	35	32	0	90%	45	42	0	93%
2	Filtering a list	5	3	0	60%	9	6	0	67%
3	Mirroring a binary tree	9	7	1	78%	9	8	0	89%
4	Checking membership in a binary tree	19	11	1	58%	19	12	0	63%
5	Computing $\sum_{i=j}^{k} f(i)$ for j, k, and f	32	11	6	34%	32	16	1	50%
6	Composing functions	46	17	0	37%	49	20	0	41%
7	Adding numbers in user-defined number system	14	4	2	29%	18	9	0	50%
8	Evaluating expressions and propositional formulas	105	29	12	28%	112	45	0	40%
9	Deciding lambda terms are well-formed or not	116	16	29	14%	141	33	0	23%
10	Differentiating algebraic expressions	162	26	7	16%	197	46	0	23%
	Total/Average	543	156	58	29%	631	237	1	38%

observed that this incorrect patch also can be fixed when changing the timeout for solver from 50 milliseconds to 2 seconds.

7 RELATED WORK

In this section, we discuss the researches closely related to our work. The existing works are classified into five categories according to their research field.

Property-based Testing. The *property-based testing* is a well-known approach for testing functional programs. It aims to find test cases which fail to satisfy the predefined property, and QuickCheck [Claessen and Hughes 2000] is the most famous framework for the property-based testing. It, however, has a difficulty in generating function type test cases and requires users to build test generators manually. To address these limitations, several researches have been proposed [Koopman and Plasmeijer 2006; Lampropoulos et al. 2017; Löscher and Sagonas 2017]. Koopman and Plasmeijer [2006] improved property-based testing to test higher-order function more easily by representing functions' AST as a data type format and generating instances of this data type using property-based testing tool. As constructing a test case generator is a burdensome task for users, Lampropoulos et al. [2017] demonstrated *Luck*, a language which allows users to easily build, read, and maintain the property-based test generators. Löscher and Sagonas [2017] proposed an enhanced property-based testing called *targeted property-based testing* and implemented TARGET which uses a search strategy based guidance rather than completely random testing to generate test cases more effectively. These recent works, however, still have the same limitation addressed above; they basically require manual human effort, which is particularly undesirable when testing numerous programs.

Symbolic Execution. Symbolic execution [Cadar et al. 2011; Khurshid et al. 2003; King 1976] is another approach which is widely used in program testing. For example, it is used to check the correctness of students' program for providing an appropriate feedback [Phothilimthana and Sridhara 2017; Singh et al. 2013]. It, however, has a well-known problem called path explosion as it basically collects all execution paths of a program. Despite the modern high-performance SAT and SMT solvers make symbolic execution practical by eliminating infeasible paths [Cadar et al. 2008a,b], it is still hard to symbolically execute functional programs. In Section 6.3, we briefly mentioned that it is not appropriate to use only the symbolic execution for testing numerous students' submissions as real submissions involve a lot of features which burden the symbolic execution.

Symbolic execution of higher-order program is especially challenging. Several works adopted higher-order behavioral contracts [Findler and Felleisen 2002] to specify the behaviors of the higher-order functions [Nguyen et al. 2014; Tobin-Hochstadt and Van Horn 2012]. Nguyen and Van Horn [2015] presented sound and relatively complete semantics for constructing a counter-example of a higher-order program. When it encounters the application of a symbolic function, without any user's assumptions, it gradually refines the unknown values of the symbolic function and maintains a complete path condition as a form of first-order formula. When it reaches an error state, it constructs a counter-example by solving the path condition. While these works focused on symbolically executing unknown functions, we mitigate the burden of symbolic function in a relatively simple way; we use symbolic technique only for the limited domain (i.e., integer and string). For the rest, we perform enumerative search (i.e., there are no symbolic functions or symbolic data types during symbolic execution). Our evaluation results show that this approach is simple yet effective to detect logical errors in a number of higher-order programs.

Higher-order Function Testing. Several researches have been presented to test higher-order function [Heidegger and Thiemann 2010; Klein et al. 2010; Koopman and Plasmeijer 2006; Selakovic et al. 2018]. Klein et al. [2010] and Heidegger and Thiemann [2010] extended random testing to work on higher-order program. Their key ideas are to use developer-provided contracts which guide random testing. *LambdaTester* [Selakovic et al. 2018] is a test case generator for higher-order functions in JavaScript. It automatically determines which parameters are expected as functions and creates a sequence of method calls from a given setup code. However, these works still require manual human effort like constructing well-tuned test generator, providing behavioral contracts as specifications, or building an appropriate setup code to create a set of initial values for testing.

As common compilers require a program or a function as their input, we also categorize compiler testing [Pałka et al. 2011; Sun et al. 2016; Yang et al. 2011] as a part of higher-order program testing. While existing works on compiler testing targeted to generate relatively sizable yet expressive programs to make divergent behaviors of several compilers, our goal is to detect a concise counter-example which causes a behavioral difference between two programs. To easily find the specific values making two program work differently, we use symbolic verification which is not used in the those compiler testing.

Automatic Program Repair. Recently, automatic program repair technique has been widely used to fix bugs in general programs [Forrest et al. 2009; Kim et al. 2013; Le Goues et al. 2012; Long and Rinard 2016; Nguyen et al. 2013; Weimer et al. 2009] or students' implementations [Bhatia et al. 2018; D'Antoni et al. 2016; Gupta et al. 2017; Lee et al. 2018b; Pu et al. 2016; Singh et al. 2013]. Most of these works use a test suite as a specification of the generated patches. However, the test-case-based approach has a fundamental limitation; it often generates test-suite-overfitted patches which only satisfy the given test suite, and the potential bugs still remain [Smith et al. 2015].

Many researches have been proposed to resolve this problem by supplementing the original test suite with automatic test case generation technique [Mechtaev et al. 2018; Xin and Reiss 2017; Yang et al. 2017]. DiffTGen [Xin and Reiss 2017] generates a new test case based on the syntactic differences between a generated patch and a provided oracle program. *Opad* [Yang et al. 2017] leverages fuzz testing to generate random inputs from original test suite. On the other hand, our approach systematically generates a counter-example based on symbolic verification rather than depending on syntactic differences or random fuzzing.

Mechtaev et al. [2018] proposed *counter-example-guided inductive repair* of which approach is similar to ours. Given a reference program, it automatically infers an intended specification as a formula by symbolically executing the reference program. It guarantees that the generated patches are conditionally equivalent to the specification. However, it is difficult to apply this technique directly to functional programs as they have some features which are hard to manipulate with the pure symbolic execution.

Program Synthesis. Program synthesis technique has been widely used in various domains such as generation of complex APIs [Feng et al. 2017] or SQL queries [Wang et al. 2017; Yaghmazadeh et al. 2017], and programming education [D'Antoni et al. 2016; Lee et al. 2018b; Pu et al. 2016; Singh et al. 2013; So and Oh 2017, 2018]. As the program synthesis gains the popularity recently, a lot of researches are proposed to improve the performance of it.

In functional program synthesis, there is a well-known technique called type-directed search [Feser et al. 2015; Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016] which significantly reduces the search space by pruning out ill-typed programs. SAT or SMT solver is also popular as an aid of program synthesis to reduce the search space [Albarghouthi et al. 2013; Gulwani et al. 2011; Kneuss et al. 2013]. Machine learning technique such as deep learning [Balog et al. 2017] or probabilistic model [Lee et al. 2018a] were used to enhance program synthesis by giving priority to candidates which are likely to be solutions. Applying static analysis or dynamic symbolic execution is another promising approach for pruning out the redundant states during program synthesis [Lee et al. 2018b; So and Oh 2017, 2018]. To generate a well-typed test case and to prove whether it is an actual counter-example, we adopted two techniques above, type-directed search and SMT solving based on symbolic execution. We believe that the other techniques can also improve our technique.

Existing program synthesizers [Albarghouthi et al. 2013; So and Oh 2017; Wang et al. 2017] require users to provide appropriate components for synthesis. Unlike them, our technique uses symbolic execution to automatically deduce the values which cause differences in the behavior of two programs without requiring any synthesis components. So and Oh [2018] also use constraint solving to infer the integer components for synthesis. However, the constraints are generated from the given input-output examples, not symbolic execution.

8 CONCLUSION

In this paper, we presented a novel technique to automatically detect a logical error in functional programming assignments. The novelty of our technique is that it combines enumerative search and symbolic execution to achieve the benefits from both. By this key approach, our technique can effectively detect logical errors from massive student submissions as well as efficiently. We conducted the experiments with 4060 student submissions from functional programming course. Throughout the evaluation, we observed that our technique detected 88 more erroneous programs than human-made test cases which have been actually used for grading. Moreover, the results demonstrated that our technique outperformed the existing property-based testing, which requires human effort, and enhanced an existing functional program repair system.

A TEST CASES FOR PROBLEM 10

The 10 test cases we have used for grading submissions for Problem 10 as follows:

```
(Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1], "x") [("x", 2)] => 6
(Sum [Power ("x", 2); Power ("x", 2); Const 1], "x") [("x",3)] => 12
(Sum [Power ("x", 2); Power ("x", 2); Const 1], "y") [("x",1)] => 0
(Times [Power ("x", 3); Power ("y", 2)], "x") [("x", 10); ("y", 5)] => 7500
```

```
(Sum [Times [Sum [Var "x"; Var "y"]; Times [Var "x"; Var "y"]]; Power ("x", 2)], "x") [("x",
    3); ("y", 4)] => 46
(Times [Times [Sum [Var "x"; Var "y"]; Var "x"]; Var "x"], "x") [("x", 2); ("y", 5)] => 32
(Times [Power ("x",2); Var "y"], "x") [("x", 3); ("y", 4)] => 24
(Times [Const 2; Sum [Var "x"; Var "y"]; Power ("x", 3)], "x") [("x", 2); ("y", 1)] => 88
(Times [Sum [Var "x"; Var "y"; Var "z"]; Power ("x", 2); Sum[Times [Const 3; Var "x"]; Var
    "z"]], "x") [("x", 2); ("y", 1); ("z", 1)] => 188
(Times [Sum [Var "x"; Var "y"; Var "z"]; Power ("x", 2); Sum[Times [Const 3; Var "x"]; Var
    "z"]], "y") [("x", 1); ("y", 1); ("z", 1)] => 4
```

Each of them means an input and output relation. To compare various expressions with the same semantics, we provide not only an arithmetic expressions and a variable name as input but also an environment revealing a variable-value mapping. For example, the first test case `(Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1], "x") [("x", 2)] => 6` represents that differentiating an expression `(Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1]` by x and assigning 2 to x is expected to have 6 as an output.

B QCHECK GENERATOR AND SHRINKER FOR PROBLEM 10

```
let shrink ((ae, x), env) =
  let open QCheck.Iter in
  let rec shrink_ae ae = match ae with
    | Const n -> map (fun n' -> Const n') (QCheck.Shrink.int n)
    | Var x -> map (fun x' -> Var x') (QCheck.Shrink.int x)
    | Power (x, n) ->
      map (fun x' -> Power (x', n)) (QCheck.Shrink.int x)
      <+> map (fun n' -> Power (x, n')) (QCheck.Shrink.int n)
    | Times aes -> map (fun aes' -> Times aes') (QCheck.Shrink.list ~shrink:shrink_ae aes)
    | Sum aes -> map (fun aes' -> Sum aes') (QCheck.Shrink.list ~shrink:shrink_ae aes)
  in
  let rec shrink_env e =
     QCheck.Shrink.list ~shrink:(QCheck.Shrink.pair (QCheck.Shrink.int) (QCheck.Shrink.int)) e
  in pair (pair (shrink_ae ae) (QCheck.Shrink.int x)) (shrink_env env)
let gen =
  let open QCheck.Gen in
  let gen_ae = sized (fix (fun recgen n -> match n with
    | 0 -> oneof [map (fun n -> Const n) small_int;
                  map (fun x -> Var n) nat;
                  map2 (fun x n -> Power (x, n)) nat small_int]
    | _ -> frequency [1, map (fun n -> Const n) small_int;
                      1, map (fun x -> Var n) nat;
                      1, map2 (fun x n -> Power (x, n)) nat small_int;
                      1, map (fun aes -> Times aes) (list_size (int_range 0 5) (recgen (n/4)));
                      1, map (fun aes -> Sum aes) (list_size (int_range 0 5) (recgen (n/4)));]))
  in pair (pair gen_ae nat) (list_repeat 5 (pair nat small_int))
```

ACKNOWLEDGMENTS

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (2017M3C4A7068175). This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09.

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. Springer-Verlag, Berlin, Heidelberg, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *ICLR*.

Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 60–70. https://doi.org/10.1145/3180155.3180219

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008a. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008b. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (Dec. 2008), 38 pages. https://doi.org/10.1145/1455518.1455522

C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*. 1066–1071. https://doi.org/10.1145/1985793.1985995

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantiative Objectives. https://www.microsoft.com/en-us/research/publication/qlose-program-repair-with-quantiative-objectives/

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 599–612. https://doi.org/10.1145/3009837.3009851

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/581478.581484

Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09)*. ACM, New York, NY, USA, 947–954. https://doi.org/10.1145/1569901.1570031

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 802–815. https://doi.org/10.1145/2837614.2837629

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*. AAAI Press, 1345–1351. http://dl.acm.org/citation.cfm?id=3298239.3298436

Phillip Heidegger and Peter Thiemann. 2010. Contract-Driven Testing of JavaScript Code. In *Objects, Models, Components, Patterns*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–172.

Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. Springer-Verlag, Berlin, Heidelberg, 553–568. http://dl.acm.org/citation.cfm?id=1765871.1765924

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 802–811. http://dl.acm.org/citation.cfm?id=2486788.2486893

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

Casey Klein, Matthew Flatt, and Robert Bruce Findler. 2010. Random Testing for Higher-order, Stateful Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 555–566. https://doi.org/10.1145/1869459.1869505

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications (OOPSLA '13)*. ACM, New York, NY, USA, 407–426. https://doi.org/10.1145/2509136.2509555

Pieter Koopman and Rinus Plasmeijer. 2006. Automatic Testing of Higher Order Functions. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 148–164.

Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 114–129. https://doi.org/10.1145/3009837.3009868

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 3–13. http://dl.acm.org/citation.cfm?id=2337223.2337225

Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh. 2018b. Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 158 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276528

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018a. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 46–56. https://doi.org/10.1145/3092703.3092711

Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic Program Repair Using a Reference Implementation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 129–139. https://doi.org/10.1145/3180155.3180247

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890

Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 139–152. https://doi.org/10.1145/2628136.2628156

Phúc C. Nguyen and David Van Horn. 2015. Relatively Complete Counterexamples for Higher-order Programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 446–456. https://doi.org/10.1145/2737924.2737971

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

Phitchaya Mangpo Phothilimthana and Sumukh Sridhara. 2017. High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students?. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, New York, NY, USA, 182–187. https://doi.org/10.1145/3059009.3059058

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_P: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2016)*. ACM, New York, NY, USA, 39–40. https://doi.org/10.1145/2984043.2989222

Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. 2018. Test Generation for Higher-order Functions in Dynamic Languages. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 161 (Oct. 2018), 27 pages. https://doi.org/10.1145/

3276531

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195

Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 532–543. https://doi.org/10.1145/2786805.2786825

Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 364–381.

Sunbeom So and Hakjoo Oh. 2018. Synthesizing Pattern Programs from Examples. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18)*. AAAI Press, 1618–1624. http://dl.acm.org/citation.cfm?id=3304415.3304645

Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 849–863. https://doi.org/10.1145/2983990.2984038

Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-order Symbolic Execution via Contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 537–554. https://doi.org/10.1145/2384616.2384655

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

Qi Xin and Steven P. Reiss. 2017. Identifying Test-suite-overfitted Patches Through Test Case Generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 226–236. https://doi.org/10.1145/3092703.3092718

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 63 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133887

Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 831–841. https://doi.org/10.1145/3106237.3106274

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532