Context-Aware and Data-Driven Feedback Generation for Programming Assignments

Anonymous Author(s)

ABSTRACT

Recently, various techniques have been proposed to automatically provide personalized feedback on programming exercises. The cutting edge of which is the data-driven approaches that leverage a corpus of existing correct programs and repair incorrect submissions by using similar reference programs in the corpus. However, current data-driven techniques work under the strong assumption that the corpus contains a solution program that is close enough to the incorrect submission. In this paper, we present Cafe, a new data-driven approach for feedback generation that overcomes this limitation. Unlike existing approaches, Cafe uses a novel context-aware repair algorithm that can generate feedback even if the incorrect program differs significantly from the reference solutions. We implemented Cafe for OCaml and evaluated it with 4,211 real student programs. The results show that Cafe is able to repair 82% of incorrect submissions, far outperforming existing approaches.

1 INTRODUCTION

In recent years, there has been a surge of interest in automatic feedback generation for programming assignments [1, 5, 11, 15, 16, 18, 29, 31, 32, 35, 36, 39]. As the demand for programming education grows, it is becoming increasingly difficult for an instructor to provide personalized feedback to a large number of students. Simply providing an instructor's solution as feedback is unsatisfactory, as students's attempts typically diverge from the reference solution. The goal of automatic feedback generation technology is to help students to understand what they did wrong and how to fix it without manual effort of instructors.

Among existing techniques, data-driven approaches [11, 30, 38, 39] are arguably the current state-of-the-art. The idea of these techniques is to leverage a corpus of existing correct programs, and repair an incorrect program by using similar reference solutions in the corpus. In contrast to approaches that require intervention of instructors [5, 15, 35], data-driven techniques are fully automatic and yet show impressive performance in repairing introductory programming exercises.

However, existing data-driven techniques have a significant shortcoming. That is, they rely on a strong assumption that the corpus contains a solution program that is *close* to the incorrect program. For example, two notable techniques, Clara [11] and Sarfgen [39], assume a solution exists that is equivalent to the incorrect program modulo control flows. This assumption, however, does not hold always [18], especially when providing feedback beyond introductory-level exercises. In this case, constructing a corpus with the close-program assumption becomes a challenge.

In this paper, we present Cafe, a new data-driven feedback generation technique that overcomes the above limitation. Unlike existing approaches, Cafe can generate feedback even when the incorrect submission is substantially different from reference solutions.

The keystone of Cafe is its context-aware, function-level repair algorithm. Cafe primarily targets sizable programming exercises, where students are freely allowed to define and use their own helper functions. To repair such a program, Cafe does not seek to find a solution program that matches the submission in its entirety; instead, it leverages multiple, partially-matching references. More specifically, Cafe works at the function level, aiming to separately repair each function in the incorrect program by (1) finding a matching function from the corpus, (2) computing their difference, and (3) extracting a patch from the difference. A main challenge with this approach is how to find the matching function that is useful for repair. Our key idea to solve this problem is to infer and compare the *original intent* of the functions by analyzing their calling contexts in the respective programs, which robustly identifies useful references even when functions have different syntax and semantics.

We evaluated Cafe in a real classroom setting. The original motivation of this work was to develop a feedback generation system for our own programming course, where we use OCaml and newcomers to functional programming often have a hard time. Thus, we implemented Cafe for OCaml and evaluated it with 664 incorrect and 3,547 correct student programs collected from the course over the past few years. In total, Cafe successfully repaired 82% (547/664) of incorrect submissions, vastly outperforming FixML [18], a recent feedback generation technique for OCaml, whose fix rate was 35% (234/664). We also confirmed that existing data-driven approaches are ineffective for our dataset; replacing our context-aware approach by the matching algorithm of Sarfgen [39] decreased the fix rate from 82% to 59%. Finally, we conducted a user study, which shows Cafe is actually helpful for students.

We summarize our contributions below:

- We present Cafe, a new context-aware and data-driven feedback generation technique for programming assignments.
- We evaluate Cafe in a realistic setting and make the tool and benchmarks publicly available.[1]

2 OVERVIEW

2.1 Motivating Example

Let us consider a programming exercise asking students to write a function `diff: aexp * string -> aexp`, which takes an arithmetic expression (`aexp`) and a variable name (`string`), and performs symbolic differentiation. Arithmetic expressions are defined in 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 arithmetic expression is either constant integer (`Const`), variable (`Var`), exponentiation (`Power`), sum of arithmetic expressions (`Sum`), or product of arithmetic expressions (`Times`). The function

[1] URL temporarily hidden as our data is not anonymized

```
1  let rec diff (e,x) =
2    let rec timediff (tlst,x) =
3      match tlst with [] -> [] | hd::tl ->
4  (-)  (match hd with Const c -> (Const c)::timediff(tl,x)
5  (-)  | Var v -> if v=x then (Const 1)::timediff(tl,x) else (Const 0)::timediff(tl,x)
6  (-)  | Power (v,c) -> if v=x then (Times[Const c;Power(v,c-1)])::timediff(tl,x) else (Const 0)::timediff(tl,x))
7  (+)  if tl=[] then [diff (hd, x)] else [Times ([diff (hd, x)]@tl)] @ [Times ([hd]@[Sum (timediff (tl, x))])]
8    in let rec sumdiff (slst,x) =
9      match slst with [] -> [] | hd::tl ->
10 (-)  (match hd with Const c -> (Const 0)::(sumdiff (tl,x))
11 (-)  | Var v -> if v=x then (Const 1)::(sumdiff (tl,x)) else (Const 0)::(sumdiff (tl,x))
12 (-)  | Power (v,c) -> if v=x then (Times [Const c;Power(v,c-1)])::(sumdiff (tl,x)) else (Const 0)::(sumdiff (tl,x))
13 (-)  | Sum lst -> (Sum (sumdiff (lst,x)))::(sumdiff (tl,x)) | Times lst -> (Times (timediff (lst,x)))::(sumdiff (tl,x)))
14 (+)  (diff (hd, x))::(sumdiff (tl, x))
15   in let rec differ (e,x) =
16     match e with Const c -> Const 0 | Var v -> if v=x then Const 1 else Const 0
17     | Power (v,c) -> if v=x then Times [Const c;Power(v,c-1)] else Const 0 | Sum lst -> Sum (sumdiff (lst,x))
18 (-) | Times lst -> Times (timediff (lst,x))
19 (+) | Times lst -> Sum (timediff (lst,x))
20   in differ (e, x)
```

Figure 1: A real incorrect student submission and the feedback generated by Cafe

```
let rec check (l, str) = ...
let rec diff (aexp, str) = ...
and gettimes (l, str) =
  match l with | [] -> [] | hd::tl ->
  if tl=[] then [diff (hd, str)] ...
and getsum (l, str) = ...
```

(a) Program used to fix `timediff`

```
let rec diff (e, x) = ...
and diff_sum (lst, key) =
  match lst with hd::tl -> ...
and diff_time (lst, key, x, y) =
  let rec f (lst, key, p, q) = ... in
    if x > y then [] else ...
```

(b) Program used to fix `sumdiff`

```
let rec diff (aexp, str) =
  let rec diffh (aexp, str) = ...
    | Sum l -> Sum (sum (l, str))
  and times (aexp, str, acc) = ...
  and sum (aexp, str) = ...
in diffh (aexp, str)
```

(c) Program used to fix `differ`

Figure 2: Three different solution programs chosen by Cafe to repair the program in Figure 1

`diff` should produce an expression that results from differentiating the given expression with respect to the given variable. For example, `diff (Sum [Power ("x", 2); Const 1], "x")`, i.e., differentiating $x^2 + 1$ w.r.t. x, outputs `Times [Const 2; Var "x"]` denoting $2 * x$.

Figure 1 shows an incorrect program written by a student in our class, which implements `diff` with three helper functions: `timediff`, `sumdiff`, and `differ`. The functions `timediff` and `sumdiff` are intended to compute the derivatives of the product and sum of `aexp` lists, respectively. The function `differ` performs actual differentiation using `timediff` and `sumdiff`, and handles other base cases. Note that the program erroneously handles the case of multiplication (`Times`). For example, `diff (Times [Var "x"; Var "y"], "x")` produces `Times [Const 1; Const 0]`, while the expected answer is `Var "y"`.

Despite its simple manifestation, fixing the bug correctly and providing right feedback is nontrivial even for instructors. For correct repair, we need to change three places. First, the student implemented `timediff` based on a wrong product rule, $(f \cdot g)' = f' \cdot g'$, and therefore the body of `timediff` needs to be rewritten based on the proper rule, i.e., $(f \cdot g)' = f' \cdot g + f \cdot g'$. Second, we need to rewrite `sumdiff` because it depends on the incorrect definition of `timediff`. Finally, the last line of `differ` (line 18) should be changed in accordance with the correct product rule.

Given the buggy program, test cases, and a corpus of 218 solution programs, Cafe repaired the program as shown in Figure 1 in 5 sec. To use the proper product rule, it replaced lines 4–6 of `timediff` by line 7 and line 18 of `differ` by line 19. Then, it modified the body of `sumdiff` to reflect the change. Note that the generated repair is not only correct but also instructive; indeed, it is identical to what we would manually provide to the student. For example, Cafe shows that the redundancy between `sumdiff` and `differ` can be effectively eliminated by making a recursive call to `diff`.

Compared to existing data-driven techniques [11, 30, 38, 39], the most distinguishing feature of Cafe is its ability to generate feedback by collectively using multiple, dissimilar reference solutions. Each of `timediff`, `sumdiff`, and `differ` in Figure 1 was fixed using different solutions; Cafe repaired `timediff` using `gettimes` in Figure 2(a), `sumdiff` using `diff_sum` in Figure 2(b), and `differ` using `diffh` in Figure 2(c). All of these reference programs are substantially different from the program in Figure 1, as none of the 218 solutions in our corpus had a matching control-flow structure, which implies that existing data-driven techniques [11, 30, 38, 39] would fail to generate the desired feedback.

2.2 How Cafe Works

Now we discuss the high-level ideas of our approach on the student attempts to `apply`: "Given a list `l` of integers and a target

```
1  let rec inc_all l =
2    match l with [] -> []
3    | h::t -> (h+1)::(inc_all t)
4
5  let rec dec_all l =
6    match l with [] -> []
7    | h::t -> (h+1)::(dec_all t)
8
9  let apply ((l, o) as i) =
10   match l with [] -> []
11   | hd::tl -> match o with
12     | ADD -> (hd+1)::(inc_all tl)
13     | SUB -> (dec_all tl)
```

(a) Incorrect program (P)

```
1  let rec add_list l1 =
2    match l1 with [] -> []
3    | h1::t1 -> (h1+1)::(add_list t1)
4
5
6
7
8
9  let rec apply1 ((l1, o1) as i) =
10   match l1 with [] -> [] | hd1::tl1 ->
11     let i' = (tl1,o) in match o1 with
12     | ADD -> (hd1+1)::(add_list tl1)
13     | SUB -> (hd1-1)::(apply1 i')
```

(b) Correct program (P_1)

```
1  let id x = x
2
3
4
5  let rec sub_list l2 =
6    match l2 with [] -> []
7    | h2::t2 -> (h2-1)::(sub_list t2)
8
9  let rec apply2 ((l2, o2) as i) =
10   match l2 with [] -> id []
11   | hd2::tl2 -> match o2 with
12     | ADD->(hd2+1)::(apply2 (tl2, o2))
13     | SUB->(hd2-1)::(sub_list tl2)
```

(c) Correct program (P_2)

Figure 3: A running example to illustrate how Cafe works

operation o ∈ {ADD, SUB}, write a function that increments (resp. decrements) each element of l by 1 if the operation is addition, i.e., o = ADD (resp. subtraction, i.e., o = SUB)". Figure 3a–3c show three student submissions of the programming assignment: P is incorrect, while P_1 and P_2 are functionally correct. We differently name the top-level functions that the three submissions implement to distinguish them (P: apply, P_1: apply1, and , P_2: apply2).

Our goal is to automatically generate modifications that make the incorrect submission P correct as a guided feedback by using the existing correct student solutions P_1 and P_2. The program P is functionally wrong in decrementing list elements. We can correct P by (i) modifying the expression h+1 at line 7 to be h-1 and (ii) changing `(dec_all tl)` at line 13 to `(hd-1)::(dec_all tl)`.

Context-Aware Matching. The first step is to find a matching relation between functions in the student submissions. We say that two functions f and g match, written $f \sim g$ if the functions are invoked and call other functions under *compatible contexts*. Here, we mean contexts by conditions over execution paths when function calls occur. We say two path conditions *compatible* if there exists an input that exercises both of the two execution paths.

Based on this notion, Cafe finds the following matching relation: `inc_all` ∼ `add_list`, `dec_all` ∼ `sub_list`, and `apply` ∼ `apply1`. We consider two types of function contexts: (a) incoming contexts describing under what path conditions the function is invoked by other functions, and (b) outgoing contexts describing under what conditions the function invokes other functions. For example, the incoming contexts of `dec_all`, `add_list`, and `sub_list` (denoted $\delta_{\texttt{dec\_all}}$, $\delta_{\texttt{add\_list}}$, and $\delta_{\texttt{sub\_list}}$, respectively) are as follows:

$$\begin{aligned}\delta_{\texttt{dec\_all}} &\equiv \texttt{i} = (\texttt{l}, \texttt{o}) \wedge \texttt{l} = (\texttt{hd} :: \texttt{tl}) \wedge (\texttt{o} = \texttt{SUB}) \\ \delta_{\texttt{add\_list}} &\equiv \texttt{i} = (\texttt{l}_1, \texttt{o}_1) \wedge \texttt{l}_1 = (\texttt{hd}_1 :: \texttt{tl}_1) \wedge (\texttt{o}_1 = \texttt{ADD}) \\ \delta_{\texttt{sub\_list}} &\equiv \texttt{i} = (\texttt{l}_2, \texttt{o}_2) \wedge \texttt{l}_2 = (\texttt{hd}_2 :: \texttt{tl}_2) \wedge (\texttt{o}_1 = \texttt{SUB})\end{aligned}$$

where i is called *input variable* representing the input simultaneously provided to the top-level functions. Over these contexts, Cafe concludes `dec_all` ∼ `sub_list` because the formula $\delta_{\texttt{dec\_all}} \wedge \delta_{\texttt{sub\_list}}$ is satisfiable meaning that there exists a value for i that leads to invoking both functions. On the other hand, Cafe concludes `dec_all` ≁ `add_list` because the formula $\delta_{\texttt{dec\_all}} \wedge \delta_{\texttt{add\_list}}$ is unsatisfiable (∵ SUB ≠ ADD) meaning that there is no value for i that results in invoking both functions. After a similar process, Cafe concludes `inc_all` ∼ `add_list`. Also, based on outgoing contexts, Cafe concludes `apply` ∼ `apply1`.

Note that context-aware matching differs crucially from conventional syntactic or semantic matching used in prior data-driven techniques [11, 30, 38, 39]. For example, existing approaches would match `dec_all` with `add_list` because they are equivalent in syntax and semantics. However, this matching is undoubtedly useless; we need to find a function useful for repair, rather than merely finding similar one.

Extracting Repair Templates. Next, Cafe learns fixes for each function in a buggy submission from its corresponding function in a correct submission. We support insertion/deletion of conditional branches, modifying subexpressions, and adding new function definitions. Especially, changing control flows and defining new functions are beyond the capability of the state-of-the-art feedback generation systems.

We learn such a fix by extracting a *repair template* from the correct submissions and *instantiating* it to be fit into the buggy submission. Cafe learns a repair template by syntactically differencing matched functions. For example, Cafe identifies a discrepancy between the subexpression h+1 in `decl_all` and $\texttt{h}_2$-1 in `sub_list`, and derives the template

$$\mathsf{Modify}(7, \mathsf{h}+1 \longrightarrow \square_{\mathsf{int}}{-}1) \tag{1}$$

which means one way to fix P is to replace the expression h+1 at line 7 by an expression of the form $\square_{\mathsf{int}}$-1 where $\square_{\mathsf{int}}$ can be filled by some integer variable. Similarly, Cafe produces the following template by differencing `apply` and `apply1`:

$$\mathsf{Modify}(13, \texttt{dec\_all tl} \longrightarrow (\square_{\mathsf{int}} - 1) :: (\square_{\mathsf{int\ list} \times \tau \to \mathsf{int\ list}}\ \square_{\mathsf{int\ list} \times \tau})) \tag{2}$$

where τ denotes the algebraic data type for ADD or SUB.

Instantiating Templates. Next Cafe instantiates the templates by filling the holes with proper variables to obtain concrete fixes. Each hole with a type is filled with a variable available at the location with the same type. From the template (1), $\mathsf{Modify}(7, \mathsf{h}+1 \longrightarrow \mathsf{h}-1)$ is generated because h is the only available integer variable at line 7, which is correct. From the template (2), the first hole can be filled with hd because it is the only available integer variable at line 13, and the following partially completed template is obtained.

$$\mathsf{Modify}(13, \texttt{dec\_all tl} \to (\mathsf{hd}-1) :: (\square_{\text{int list}\times\tau \to \text{int list}}\ \square_{\text{int list}\times\tau}))$$

Unfortunately, there is no variable of type int list $\times\ \tau \to$ int list (`apply` is unavailable as it is not recursive). In such a case, we just enumerate all function identifiers as candidates for the hole and obtain the partially completed templates:

$$\mathsf{Modify}(13, \texttt{dec\_all tl} \to (\mathsf{hd}-1) :: (\texttt{dec\_all}\ \square_{\text{int list}}))$$
$$\mathsf{Modify}(13, \texttt{dec\_all tl} \to (\mathsf{hd}-1) :: (\texttt{inc\_all}\ \square_{\text{int list}}))$$

Note that the annotated type of the last hole has been changed in accordance with the type of `dec_all` and `inc_all`. The last hole is filled with `tl`, which is an available variable of type int list. Finally, we obtain the following set $\mathcal{A}$:

$$\left\{\begin{array}{l} \mathsf{Modify}(7, \mathsf{h}+1 \to \mathsf{h}-1), \\ \mathsf{Modify}(13, \texttt{dec\_all tl} \to (\mathsf{hd}-1) :: (\texttt{dec\_all tl})), \\ \mathsf{Modify}(13, \texttt{dec\_all tl} \to (\mathsf{hd}-1) :: (\texttt{inc\_all tl})) \end{array}\right\}$$

Finding a Patch. By trying each subset of $\mathcal{A}$, Cafe finds that applying the first two fixes in sequence corrects the buggy submission. Note that Cafe produced a patch that preserves the original intent of the program as much as possible by using the existing helper function `dec_all` instead of simply removing it and making `apply` recursive.

3 PROBLEM DEFINITION

In this section, we define our problem of data-driven feedback generation for programming assignments. We first define a program model that captures key aspects of Meta Language (ML)-like languages and introduce notations that allow us to formalize our algorithm in the next section.

Language. To formalize our approach, we consider an idealized functional language similar to the core of ML, with the additional property that we *label* all expressions. Our target language features algebraic data types and recursive functions. A program is an expression defined as follows:

$$\begin{array}{llll} e \in Exp & \text{(Expressions)} & x \in VId & \text{(Variables)} \\ f \in FId & \text{(Functions)} & Id = VId \cup FId & \text{(Identifiers)} \\ \ell \in Label & \text{(Labels)} & \tau \in Type & \text{(Types)} \end{array}$$

$$\begin{array}{rcl} e & ::= & n \mid x \mid \lambda x.e \mid e_1 \oplus e_2 \mid e_1\ e_2 \mid \kappa(e_1, \cdots, e_{a(\kappa)}) \\ & \mid & \kappa^{-i}(e) \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2 \\ & \mid & \mathsf{match}\ e\ \mathsf{with}\ \overline{p_i \to e_i}^k \\ p & ::= & \kappa(x_1, \cdots, x_n) \mid \_ \qquad \tau ::= \mathsf{int} \mid T \mid \tau_1 \to \tau_2 \end{array}$$

We assume each expression is associated with a unique label. Expression e associated with a label $\ell \in \mathcal{L}$ is denoted by e^ℓ. For the sake of better readability, we will often elide ℓ when the label is not necessary for discussion. In addition, when we determine equality of two expressions, we do not consider their labels and only check if they are syntactically equivalent.

The syntax of expressions is standard: application is written $e_1\ e_2$, κ ranges over data type constructors, $a(\kappa)$ denotes the arity of κ, κ^{-i} denotes a destructor which extracts the i-th subcomponent of a constructor κ, and let bindings for variables and recursive functions are allowed. For conciseness, we assume that all functions take a single argument and are not mutually recursive (our implementation in Section 5 is not limited by these restrictions though). We use ML-style pattern match expressions in which each pattern p binds subcomponents of a constructor κ, or the underscore (_) called the wildcard pattern. We use $\overline{p_i \to e_i}^k$ to denote $p_1 \to e_1 \mid \cdots \mid p_k \to e_k$. Types include the integer type int, user-defined algebraic data types T, and function types $\tau_1 \to \tau_2$.

We will use some notations regarding expressions throughout the remaining sections. We use $\longrightarrow^*$ to denote the standard multi-step call-by-value operational relation. To denote the set of all subexpressions of expression e, we will use $\mathsf{Sub}(e)$. The size of expression e will be denoted by $|e|$. Identifiers of functions defined in an expression e will be denoted by $\mathsf{functions}(e)$ (i.e., $\mathsf{functions}(e) = \{f \in FId \mid \mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2 \in \mathsf{Sub}(e)\}$). Identifiers used in an expression e will be denoted by $\mathsf{vars}(e)$.

Setting. We assume that each student submission $P \in Exp$ has no type error, and is in the following form: let rec $f(x) = e$ in $f\ x_\iota$ where f is a top-level function that the student is asked to implement, and x_ι is a special variable which we call *input variable*. In Fig. 3a – 3c, `apply`, `apply1`, and `apply2` are the top-level functions, and the variable `i` can be regarded as the input variable. For convinience, we also assume that all labels and identifiers in the pile of entire submissions are unique with exception of the input variable that all the submissions have in common. This assumption enables to use the following global functions: (1) $\mathsf{body} : FId \to Exp$ that returns the body of a function, (2) $\mathsf{param} : FId \to VId$ that returns the parameter variable of a function, and (3) $\mathsf{type} : (Label \cup Id) \to Type$ that returns the type of an expression with a given label or a variable.

Problem. Assuming some (possibly infinite) set Val of values, a set of test cases $\mathcal{T} \subseteq Val \times Val$ is used to determine the correctness of each submission. The submission P is correct (denoted $\mathsf{correct}(P, \mathcal{T})$) iff $\forall (i, o) \in \mathcal{T}.\ (\lambda x_\iota.\ P)\ i \longrightarrow^* o$. Otherwise, the submission is buggy.

Our problem is defined as follows: given a buggy submission $P_b \in Exp$, a corpus of correct submissions $\mathcal{P}_c \subseteq Exp$, and a set of test cases $\mathcal{T}$, derive a correct submission $P \notin \mathcal{P}_c$ from P_b with minimal changes (the notion of the minimality will be detailed in Section 4.2.3).

4 ALGORITHM

In this section, we describe our data-driven feedback generation algorithm. Section 4.1 formalizes calling contexts and how to find a matching relation between functions. Section 4.2 describes the repair algorithm that extracts repair templates from reference functions and uses them to correct a buggy submission.

4.1 Context-Aware Matching

We formally define the notion of context-aware matching. From each function call in all the given submissions, we collect calling contexts. A context $\delta \in Ctx$ is a path condition on the input variable under which a function is invoked. Formally, path conditions are defined below:

$$\delta ::= \mathsf{true} \mid \mathsf{false} \mid e = e \mid \neg\delta \mid \delta \wedge \delta.$$

Calling contexts are defined as follows:

Definition 4.1 (Calling context). A calling context is a triple $\langle f, \delta, g\rangle \in \mathit{FId} \times \mathit{Ctx} \times \mathit{FId}$ where f is a function, g is another function called in the body of f, and δ is a path condition under which the function call happens.

We first perform a *path-sensitive 0-CFA* on all the submissions to obtain calling contexts. Like the standard 0-CFA [28], the analysis information at any given expression is the set of possible evaluation results of the expression. Here, evaluation results are expressions in which the input variable is a free variable. We add path sensitivity by making our analysis track information separately for different execution paths. The analysis computes a dataflow state $\sigma \in (\mathit{Id} \cup \mathit{Label}) \times \mathit{Ctx} \to \mathcal{P}(\mathit{Exp}) \cup \{\top\}$.

Figure 4 depicts a subset of the constraint generation rules; the full set can be found in the supplementary material. The judgement $\delta \vdash [\![e]\!]^\ell \hookrightarrow C$ can be read as "the analysis of expression e with label ℓ generates set constraints C over dataflow state σ under a current path condition δ". While solving the constraints via a least-fix point computation, special constraints of kinds **fn**, **cn**, and **pat** are interpreted by the constraint solver to generate additional concrete constraints by referring to an intermediate analysis result. For example, from a constraint $\mathbf{fn}_\delta\ \ell_1 : \ell_2 \implies \ell$, for every function $\lambda x.e_0^{\ell_0}$ that the analysis (eventually) concludes the expression labeled ℓ_1 may evaluate to, additional constraints are generated to capture value flow from the actual argument expression ℓ_2 to formal function argument x, and from the function result to the calling expression ℓ. To enforce termination, we use a standard widening operator that transforms each collected expression whose size is greater than a threshold into $\top$.

Note that the analysis for collecting calling contexts does not affect the correctness of the overall algorithm but just determines the effectiveness of matching.

We derive calling contexts from a result of the path-sensitive 0-CFA as follows: given submissions $P_1^{\ell_1}, \cdots, P_m^{\ell_m}$, we collect set constraints C_i for each submission such that $\mathsf{true} \vdash [\![P_i]\!]^{\ell_i} \hookrightarrow C_i$, and obtain the least solution σ_i. We collect a set of calling contexts $\Delta = \Delta_1 \cup \cdots \cup \Delta_m$ where each Δ_i is

$$\bigcup_{\substack{f \in \mathsf{functions}(P_i)\\ \delta \in \mathit{Ctx}}} \{\langle f, \delta, g\rangle \mid (e_1^{\ell_1}\ e_2) \in \mathsf{Sub}(\mathsf{body}(f)), g \in \sigma_i(\ell_1, \delta)\}.$$

We also conjoin the analysis results from the submissions and obtain $\sigma = \bigsqcup_{1 \le i \le m} \sigma_i$, which will also be used in Section 4.2.

Example 4.2. Consider the invocation to `sub_list` in the body of the function `apply2` in Figure 3c. We will show how a calling context representing this function invocation is derived. From the parameter definition binding variables $\mathtt{l}_2$ and $\mathtt{o}_2$ in `apply2` where the initial path condition is true, we first generate the following constraints over a dataflow state σ.

$$\mathtt{pair}^{-1}(\mathtt{i}) \in \sigma(\mathtt{l}_2, \mathsf{true}) \tag{3}$$

$$\mathtt{pair}^{-2}(\mathtt{i}) \in \sigma(\mathtt{o}_2, \mathsf{true}) \tag{4}$$

In the outer pattern matching match $\mathtt{l}_2$ with $\mathsf{cons}(\mathsf{hd}_2, \mathsf{tl}_2) \to \cdots$, from (3), we generate the following constraint making $\mathtt{o}_2$ under a new path condition: $\sigma(\mathtt{o}_2, \mathsf{true}) \subseteq \sigma(\mathtt{o}_2, \underbrace{\mathtt{pair}^{-1}(\mathtt{i}) = \mathsf{cons}(\mathsf{hd}_2, \mathsf{tl}_2)}_{\delta})$. Under the current path condition δ, in the inner pattern matching match $\mathtt{o}_2$ with $\mathsf{SUB} \to\ \ldots\ \mathtt{sub\_list}^\ell\ \mathtt{tl}_2$ where we assume `sub_list` is associated with label ℓ, the constraint $\mathtt{sub\_list} \in \sigma(\ell, \delta \wedge \mathtt{pair}^{-2}(\mathtt{i}) = \mathsf{SUB})$ is generated from (4). After computing a least solution satisfying these constraints, we obtain a calling context $\langle \mathtt{apply2}, \delta \wedge \mathtt{pair}^{-2}(\mathtt{i}) = \mathsf{SUB}, \mathtt{sub\_list}\rangle$.

Now we are ready to measure similarity between arbitrary two functions using the calling contexts. Given two functions f and g, we compute a distance between the two functions as follows:

$$\mathsf{dist}(f, g) = w_1 \times |CC_{\mathrm{in}}^{f,g}|^{-1} + w_2 \times |CC_{\mathrm{out}}^{f,g}|^{-1}$$

where $w_{\{1,2\}}$ are coefficients that can be adjusted via statistical learning. $CC_{\mathrm{in}}^{f,g}$ (resp. $CC_{\mathrm{out}}^{f,g}$) is called incoming (resp. outgoing) compatible calling context and defined as follows:

$$CC_{\mathrm{in}}^{f,g} = \{(\delta, \delta') \mid \delta, \delta' \in \mathit{Ctx}, \langle \_, \delta, f\rangle, \langle \_, \delta', g\rangle \in \Delta, \mathit{SAT}(\delta \wedge \delta')\}$$

$$CC_{\mathrm{out}}^{f,g} = \{(\delta, \delta') \mid \delta, \delta' \in \mathit{Ctx}, \langle f, \delta, \_\rangle, \langle g, \delta', \_\rangle \in \Delta, \mathit{SAT}(\delta \wedge \delta')\}$$

If both f and g do not have any callers (resp. callees), we do not consider the term involving $CC_{\mathrm{in}}^{f,g}$ (resp. $CC_{\mathrm{out}}^{f,g}$). Note that the more compatible pairs of calling contexts two functions have, the shorter distance they have between. Based on this notion of distance, we are equipped with the following matching function that takes a function in a buggy submission and returns a function in a correct submission to be referred for correction:

$$\mathcal{M} = \lambda P_b.\ \{f \mapsto \operatorname*{argmin}_{\substack{P_c \in \mathcal{P}_c\\ g \in \mathsf{functions}(P_c)\\ \mathsf{type}(f) = \mathsf{type}(g)}} \mathsf{dist}(f, g) \mid f \in \mathsf{functions}(P_b)\}.$$

Note that we only consider functions of the same type.

Example 4.3. Suppose we want to measure the distance between `dec_all` and `sub_list`, and the distance between `dec_all` and `add_list` in Figure 3. After the 0-CFA analysis, we obtain the following calling contexts.

$$\langle \mathtt{apply}, \underbrace{\mathtt{pair}^{-1}(\mathtt{i}) = \mathsf{cons}(\mathsf{hd}, \mathsf{tl}) \wedge \mathtt{pair}^{-2}(\mathtt{i}) = \mathsf{SUB}}_{\delta_1}, \mathtt{dec\_all}\rangle$$
$$\langle \mathtt{dec\_all}, \underbrace{\delta_1 \wedge \mathsf{cons}^{-2}(\mathtt{pair}^{-1}(\mathtt{i})) = \mathsf{cons}(\mathsf{h}, \mathsf{t})}_{\delta_2}, \mathtt{dec\_all}\rangle$$
$$\langle \mathtt{apply1}, \underbrace{\mathtt{pair}^{-1}(\mathtt{i}) = \mathsf{cons}(\mathsf{hd}_1, \mathsf{tl}_1) \wedge \mathtt{pair}^{-2}(\mathtt{i}) = \mathsf{ADD}}_{\delta_3}, \mathtt{add\_list}\rangle$$
$$\langle \mathtt{add\_list}, \underbrace{\delta_3 \wedge \mathsf{cons}^{-2}(\mathtt{pair}^{-1}(\mathtt{i})) = \mathsf{cons}(\mathsf{h}_1, \mathsf{t}_1)}_{\delta_4}, \mathtt{add\_list}\rangle$$
$$\langle \mathtt{apply2}, \underbrace{\mathtt{pair}^{-1}(\mathtt{i}) = \mathsf{cons}(\mathsf{hd}_2, \mathsf{tl}_2) \wedge \mathtt{pair}^{-2}(\mathtt{i}) = \mathsf{SUB}}_{\delta_5}, \mathtt{sub\_list}\rangle$$
$$\langle \mathtt{sub\_list}, \underbrace{\delta_5 \wedge \mathsf{cons}^{-2}(\mathtt{pair}^{-1}(\mathtt{i})) = \mathsf{cons}(\mathsf{h}_2, \mathsf{t}_2)}_{\delta_6}, \mathtt{sub\_list}\rangle$$
$$\cdots$$

With $w_1 = 1$ and $w_2 = 2$ that we are using in our implementation,

$$\begin{aligned}\mathsf{dist}(\mathtt{dec\_all}, \mathtt{sub\_list}) &= |\{(\delta_1, \delta_5), (\delta_2, \delta_6)\}|^{-1} + 2 \cdot |\{(\delta_2, \delta_6)\}|^{-1}\\ &= 2.5\end{aligned}$$

whereas $\mathsf{dist}(\mathtt{dec\_all}, \mathtt{add\_list}) = \infty$ as the two functions do not share any compatible calling contexts.

$$\frac{}{\delta \vdash [\![n]\!]^{\ell} \hookrightarrow n \in \sigma(\ell, \delta)} \qquad \frac{}{\delta \vdash [\![x]\!]^{\ell} \hookrightarrow \begin{cases} x \in \sigma(x,\delta) \cap \sigma(\ell,\sigma) & (x \in \{x_\iota\} \cup \mathit{FId}) \\ \sigma(x,\delta) \subseteq \sigma(\ell,\sigma) & (\text{o.w.}) \end{cases}} \qquad \frac{[\![e]\!]^{\ell'} \hookrightarrow C}{\delta \vdash [\![\lambda x.e^{\ell'}]\!]^{\ell} \hookrightarrow \lambda x.e \in \sigma(\ell,\delta) \cup C}$$

$$\frac{\delta \vdash [\![e_1]\!]^{\ell_1} \hookrightarrow C_1 \quad \delta \vdash [\![e_2]\!]^{\ell_2} \hookrightarrow C_2}{\delta \vdash [\![e_1\ e_2]\!]^{\ell} \hookrightarrow C_1 \cup C_2 \cup \mathbf{fn}\ \ell_1 : \ell_2 \implies \ell} \qquad \frac{\lambda x.e_0^{\ell_0} \in \sigma(\ell_1,\delta)}{\mathbf{fn}_\delta\ \ell_1 : \ell_2 \implies \ell \hookrightarrow \{\sigma(\ell_2,\delta) \subseteq \sigma(x,\delta), \sigma(\ell_0,\delta) \subseteq \sigma(\ell,\delta)\}}$$

$$\frac{\delta \vdash [\![e]\!]^{\ell'} \hookrightarrow C}{\delta \vdash [\![\kappa^{-i}(e^{\ell'})]\!]^{\ell} \hookrightarrow C \cup \mathbf{cn}_\delta^{\kappa^{-i}}\ \ell' \implies \ell} \qquad \frac{e \in \sigma(\ell',\delta)}{\mathbf{cn}_\delta^{\kappa^{-i}}\ \ell' \Longrightarrow \ell \hookrightarrow \kappa^{-i}(e) \in \sigma(\ell,\delta)} \qquad \frac{\delta \vdash [\![e]\!]^{\ell_0} \hookrightarrow C_0}{\delta \vdash [\![\text{match } e^{\ell_0} \text{ with } \overline{p_i \to e_i}^k]\!]^{\ell} \hookrightarrow C_0 \cup \mathbf{pat}_\delta\ \ell_0 : \overline{p_i \to e_i}^k \Longrightarrow \ell}$$

$$\frac{e_0 \in \sigma(\ell_0,\delta) \quad \begin{matrix} \delta \wedge r_1 \vdash e_1 \hookrightarrow C_1 \\ \delta \wedge r_2 \vdash e_2 \hookrightarrow C_2 \\ \cdots \\ \delta \wedge r_k \hookrightarrow C_k \end{matrix} \quad r_i = \begin{cases} e_0 = p_i & (i=1) \\ e_0 = p_i \wedge \bigwedge_{1 \le j < i} e_0 \ne p_j & (\text{o.w.}) \end{cases} \quad \begin{matrix} C_{FV} = \bigcup_{1 \le i \le k} \{\sigma(x,\delta) \subseteq \sigma(x, \delta \wedge r_i) \mid x \in FV(e_i)\} \\ C_{\sqcup} = \bigcup_{1 \le i \le k} \{\sigma(\ell_i, \delta \wedge r_i) \subseteq \sigma(\ell,\delta)\} \end{matrix}}{\mathbf{pat}_\delta\ \ell_0 : \overline{p_i \to e_i^{\ell_i}}^k \implies \ell \hookrightarrow \bigcup_{1 \le i \le k} C_k \cup C_{FV} \cup C_{\sqcup} \cup \bigcup_{1 \le i \le k, p_i = \kappa(x_1,\cdots,x_n)} \{\kappa^{-j}(e_0) \in \sigma(x_j,\delta) \mid 1 \le j \le n\}}$$

Figure 4: Constraint generation rules for our path-sensitive 0-CFA (selected). The special constraints of kinds fn, cn, and pat are interpreted by the constraint solver to generate additional concrete constraints.

In case of tie, we pick the most syntactically similar function. To measure the syntactic similarity, we use the method of embedding ASTs into numerical vectors, which is called the position-aware characteristic vectors proposed by Wang et al. [39]. We compute Euclidean distances between vectors to obtain the syntactic distances.

4.2 Repair Algorithm

In this subsection, we explain how to extract repair templates from correct submissions and instantiate them to generate patches. In particular, our goal is to obtain a sequence of *edit actions* that transform a given buggy submission into a new correct one. This sequence is called *edit script*. We consider the following edit actions:

- Modify(ℓ, e) replaces the old subexpression at label ℓ by the new expression e.
- Insert$(\ell, p \to e)$ adds a new pattern matching case $p \to e$ into a match expression associated with label ℓ.
- Delete$(\ell, p \to e)$ removes an existing pattern matching case $p \to e$ from a match expression associated with label ℓ.
- Define(f) adds a new definition of function f into the expression. If we apply this action into an expression e, the resulting expression would be let rec $f(\text{param}(f)) = \text{body}(f)$ in e.

4.2.1 Learning Repair Templates. We generate edit scripts by *instantiating* templates (which we call *repair templates*) collected from correct submissions. A repair template is a variant of an edit action where each expression in Modify or Insert action does not have any variables but just *holes*. Each hole is annotated with a type and plays a role as a placeholder that can be replaced with a variable of the type. The set $Exp_\square$ of expressions with holes is similarly defined as Exp in the followings.

$$\begin{array}{rcl} e_\square & \in & Exp_\square \\ e_\square & ::= & \square_\tau \mid n \mid \lambda x.e_\square \mid e_{\square,1} \oplus e_{\square,2} \mid e_{\square,1}\ e_{\square,2} \\ & | & \kappa(e_{\square,1}, \cdots, e_{\square,a(\kappa)}) \mid \kappa^{-i}(e_\square) \\ & | & \text{let } x = e_{\square,1} \text{ in } e_{\square,2} \\ & | & \text{let rec } f(x) = e_{\square,1} \text{ in } e_{\square,2} \\ & | & \text{match } e_\square \text{ with } \overline{p_{\square,i} \to e_{\square,i}}^k \\ p_\square & ::= & \kappa(\square_{\tau_1}, \cdots, \square_{\tau_k}) \mid \_ \end{array}$$

An expression with holes can be considered an abstraction of multiple expressions. The abstraction function $\alpha_e : Exp \to Exp_\square$, which we apply to expressions in correct submissions to extract templates, is defined as follows (to avoid unnecessary clutter, we omit simple inductive cases):

$$\begin{array}{l} \alpha_e(n) = n \qquad \alpha_e(x^\ell) = \square^\ell_{\text{type}(x)} \qquad \alpha_e(\lambda x.e) = \lambda x.\alpha_e(e) \\ \cdots \\ \alpha_e(\text{match } e \text{ with } \overline{p_i \to e_i}^k) \\ \quad = \text{match } \alpha_e(e) \text{ with } \overline{p_i \to \alpha_e(e_i)}^k \\ \alpha_p(\kappa(x_1, \cdots, x_{a(\kappa)})) = \kappa(\alpha_e(x_1), \cdots, \alpha_e(x_{a(\kappa)})) \quad \alpha_p(\_) = \_ \end{array}$$

When abstracting a variable into a hole, we preserve its label. The label is used in various ways, which will be described later in the next subsection.

Now we describe how to generate repair templates. Given a buggy submission P and the matching function $\mathcal{M}$, we obtain a set of templates $\mathcal{T} = \mathcal{T}_D \cup \mathcal{T}_M$ where $\mathcal{T}_D$ is a set of templates of kind Define defined as follows:

$$\mathcal{T}_D = \{\text{Define}(g) \mid f \in \text{functions}(P), g \in \text{callees}(\text{body}(\mathcal{M}(f)))\}.$$

In other words, we collect all the auxiliary functions used in reference solutions. The function callees : $Exp \to \mathcal{P}(\mathit{FId})$ returns all functions that may be invoked in a given expression. The set $\mathcal{T}_M$ includes templates of kinds Modify, Insert, and Delete defined as follows:

$$\mathcal{T}_M = \bigcup \{T \mid f \in \text{functions}(P), [\![\text{body}(f), \text{body}(\mathcal{M}(f))]\!] \rightsquigarrow T\}.$$

The judgement $[\![e, e']\!] \rightsquigarrow T$ can be read as "by differencing a buggy expression e and a reference expression e', we extract a set T of

$$\frac{}{[\![n_1, n_2]\!] \rightsquigarrow \emptyset}\ n_1 = n_2 \qquad \frac{}{[\![x_1^{\ell_1}, x_2^{\ell_2}]\!] \rightsquigarrow \{\mathsf{Modify}(\ell_1, \square^{\ell_2}_{\mathsf{type}(\ell_2)})\}} \qquad \frac{[\![e_1, e_2]\!] \rightsquigarrow T}{([\![\lambda x_1.e_1), (\lambda x_2.e_2)]\!] \rightsquigarrow T} \qquad \frac{[\![e_1, e_1']\!] \rightsquigarrow T_1 \quad [\![e_2, e_2']\!] \rightsquigarrow T_2}{[\![(e_1\ e_2), (e_1'\ e_2')]\!] \rightsquigarrow T_1 \cup T_2}$$

$$\frac{\begin{array}{c} T_M = \bigcup\{T_j \mid 1 \le j \le k, [\![e_j, e_j']\!] \rightsquigarrow T_j, \alpha_p(p_j) = \alpha_p(p_j')\} \\ [\![e_0, e_0']\!] \rightsquigarrow T_0 \qquad T_I = \{\mathsf{Insert}(\ell, p_j \to \alpha_e(e_j)) \mid 1 \le j \le m, \alpha_p(p_j) \in \{\alpha_p(p_i')\}_{i=1}^{n} \backslash \{\alpha_p(p_i)\}_{i=1}^{m}\} \\ T_D = \{\mathsf{Delete}(\ell, p_j \to e_j) \mid 1 \le j \le n, \alpha_p(p_j) \in \{\alpha_p(p_i)\}_{i=1}^{n} \backslash \{\alpha_p(p_i')\}_{i=1}^{m}\} \end{array}}{[\![(\mathsf{match}\ e_0\ \mathsf{with}\ \overline{p_i \to e_i}^n), (\mathsf{match}\ e_0'\ \mathsf{with}\ \overline{p_i' \to e_i'}^m)]\!] \rightsquigarrow T_0 \cup T_M \cup T_I \cup T_D}$$

$$\frac{[\![e_1, e_1']\!] \rightsquigarrow T_1 \quad \cdots \quad [\![e_k, e_n']\!] \rightsquigarrow T_n}{[\![\kappa(e_1, \cdots, e_n), \kappa(e_1', \cdots, e_n')]\!] \rightsquigarrow \bigcup_{1 \le i \le n} T_i} \qquad \frac{}{[\![e_1^{\ell_1}, e_2^{\ell_2}]\!] \rightsquigarrow \{\mathsf{Modify}(\ell_1, \alpha_e(e_2))\}}\quad (e_1 \text{ and } e_2 \text{ are of different kinds.})$$

Figure 5: Inference rules for extracting edit action templates for given two expressions (selected).

edit action templates that can be potentially used to correct e". Figure 5 depicts a subset of inference rules for extracting templates for a given pairs of expressions. The full set is deferred to the supplementary material.

Example 4.4. Suppose we extract a set T of templates from `sub_list` for correcting `dec_all` in Figure 3. We extract templates T such that $[\![e_1, e_2]\!] \rightsquigarrow T$ where e_1 is body(`sub_list`) with labels $\ell_{1..6}$ and e_2 is body(`dec_all`) with labels $\ell'_{1..6}$:

$$e_1 = \begin{array}{l} \mathtt{match}\ \mathtt{l}^{\ell_1}\ \mathtt{with} \\ p_1 \to \mathtt{empty} \\ \mid p_2 \to \mathtt{cons}((\mathtt{h}^{\ell_2} + 1)^{\ell_3}, \mathtt{dec\_all}^{\ell_4}\ \mathtt{t}^{\ell_5})^{\ell_6} \end{array}$$

$$e_2 = \begin{array}{l} \mathtt{match}\ \mathtt{l}_2^{\ell_1'}\ \mathtt{with} \\ p_1' \to \mathtt{empty} \\ \mid p_2' \to \mathtt{cons}((\mathtt{h}_2^{\ell_2'} - 1)^{\ell_3'}, \mathtt{sub\_list}^{\ell_4'}\ \mathtt{t}_2^{\ell_5'})^{\ell_6'} \end{array}$$

and $p_1 = p_1' = \mathtt{empty}$, $p_2 = \mathtt{cons}(\mathtt{h}, \mathtt{t})$, and $p_2' = \mathtt{cons}(\mathtt{h}_2, \mathtt{t}_2)$. By the inference rule for differencing two matching expressions in Figure 5, we first compare $\mathtt{l}^{\ell_1}$ and $\mathtt{l}_2{}^{\ell_1'}$ and derive a template $\mathsf{Modify}(\ell_1, \square_{\mathsf{int\ list}})$. Since both $\{\alpha_p(p_1), \alpha_p(p_2)\}$ and $\{\alpha_p(p_1'), \alpha_p(p_2')\}$ are $\{\mathtt{empty}, \mathtt{cons}(\square_{\mathsf{int}}, \square_{\mathsf{int\ list}})\}$ and the expressions matched for p_1 and p_1' are the same, we compare the matched expressions for p_2 and p_2' labeled ℓ_6 and ℓ_6' respectively. By the rule for differencing two constructors, we additionally derive $\mathsf{Modify}(\ell_3, \alpha_e(\mathtt{h}_2 - 1)) = \mathsf{Modify}(\ell_3, \square^{\ell_2'}_{\mathsf{int}} - 1)$, $\mathsf{Modify}(\ell_4, \square_{\mathsf{int\ list} \to \mathsf{int\ list}})$, and $\mathsf{Modify}(\ell_5, \square_{\mathsf{int\ list}})$.

4.2.2 Generating Edit Scripts. We instantiate the collected templates into edit actions using the following concretization function $\gamma_e^P : Exp_\square \to \mathcal{P}(Exp)$ parametrized by a buggy submission P.

$$\begin{array}{l} \gamma_e^P(n) = \{n\} \qquad \gamma_e^P(\lambda x.e_\square) = \{\lambda x.e \mid e \in \gamma_e^P(e_\square)\} \\ \cdots \\ \gamma_e^P(\mathsf{match}\ e_{\square,0}\ \mathsf{with}\ \overline{p_i \to e_{\square,i}}^k) \\ \quad = \{\mathsf{match}\ e_0\ \mathsf{with}\ \overline{p_i \to e_i}^k \mid e_i \in \gamma_e^P(e_{\square,i})\} \end{array}$$

Most importantly,

$$\gamma_e^P(\square_\tau^\ell) = \begin{cases} \{x \in V \mid \mathsf{type}(x) = \tau\} & (\exists x \in V.\mathsf{type}(x) = \tau) \\ V & (\text{otherwise}) \end{cases}$$

where $V = \mathsf{vars}(P) \cup \{x \in FId \mid \delta \in Ctx.x \in \sigma(\ell, \delta)\}$ is the set of candidate variables for the given hole $\square_\tau^\ell$. Note that the label ℓ always originates from a correct submission. We consider not only variables in P but also function identifiers reachable at ℓ as candidates for the hole. This is because we may copy function definitions in correct submissions (via Define actions) into P and let P invoke the newly added functions. In case of no candidate variable of type τ, we just enumerate all the variables in V.

Over a buggy submission P using the concretization function, we may obtain the following set $\mathcal{A}$ of edit actions.

$$\begin{array}{l} \mathcal{A} = \{\mathsf{Modify}(\ell, e) \mid \mathsf{Modify}(\ell, e_\square) \in \mathcal{T}_M, e \in \gamma_e^P(e_\square)\} \\ \quad \cup \{\mathsf{Insert}(\ell, p \to e) \mid \mathsf{Insert}(\ell, p \to e_\square) \in \mathcal{T}_M, e \in \gamma_e^P(e_\square)\} \\ \quad \cup \{\mathsf{Delete}(\ell, p \to e) \mid \mathsf{Delete}(\ell, p \to e) \in \mathcal{T}_M\} \\ \quad \cup \mathcal{T}_D. \end{array}$$

However, this method is not scalable in practice as the number of concretized edit actions is potentially exponential to the number of holes in the template (despite the type-based pruning). To reduce the number of candidates for the holes, we use the following improved concretization function $\tilde{\gamma}_e^P$, which is similarly defined as γ_e^P except for the following case:

$$\tilde{\gamma}_e^P(\square_\tau^\ell) = \begin{cases} V_\tau\mid_\ell & (V_\tau\mid_\ell \neq \emptyset) \\ \gamma_e^P(\square_\tau^\ell) & \text{otherwise} \end{cases}$$

where $V_\tau\mid_\ell$ is the set of variables of type τ in V that may take the same value reachable at label ℓ. Formally,

$$V_\tau\mid_\ell = \{x \in V \mid \mathsf{type}(x) = \tau, \exists \delta, \delta'.\ \sigma(x, \delta) \cap \sigma(\ell, \delta') \neq \emptyset\}.$$

This heuristic is inspired by the variable-usage based α-conversion [39] but we analyze the usage more accurately using the result of our path-sensitive 0-CFA.

Example 4.5. Recall the template $\mathsf{Modify}(\ell_3, \square^{\ell_2'}_{\mathsf{int}} - 1)$ derived in Example 4.4. When concretizing $\square^{\ell_2'}_{\mathsf{int}}$, we only consider the variable h as a candidate for the hole because

$$\begin{array}{l} \sigma(\mathtt{h}, \delta_1) \ni \mathtt{cons}^{-1}(\mathtt{cons}^{-2}(\mathtt{pair}^{-1}(\mathtt{i}))) \\ \sigma(\ell_2', \delta_5) \ni \mathtt{cons}^{-1}(\mathtt{cons}^{-2}(\mathtt{pair}^{-1}(\mathtt{i}))) \end{array}$$

where δ_1 and δ_5 are the path conditions defined in Example 4.3.

Changing Annotated Types during Instantiation. For ease of presentation, we have presented our instantiation method as if types associated with holes could never change after they were determined. In the actual implementation, we change the types during the course of instantiation. Whenever a hole is filled with a variable, we perform type inference to change types of the other holes

Algorithm 1 The CAFE Algorithm

```
Input: A buggy submission P_b, a set of correct submissions 𝒫_c,
    and a set of test cases 𝒯
Output: A program P_c satisfying all the test cases in 𝒯
1: 𝒜 ← ∅                                          ▷ Set of edit actions
2: 𝒫 ← 𝒫_c ∪ {P_b}
3: σ ← result of the path-sensitive 0CFA on 𝒫
4: Δ ← all calling contexts derivable from σ
5: Derive ℳ from Δ                                 ▷ ℳ : FId → FId
6: 𝒯 ← ExtractTemplates(ℳ, Δ, P_b, 𝒫_c)
7: for T ∈ 𝒯 do
8:     𝒜 ← 𝒜 ∪ InstantiateTemplate(T, P_b, σ)
9: n ← 1
10: repeat              ▷ E: edit script comprising n edit actions
11:     for each permutation E of n elements of 𝒜 do
12:         P ← apply E into P_b
13:         if correct(P, 𝒯) then
14:             return P
15:     n ← n + 1
16: until n ≤ |𝒜|
```

accordingly. An example case is the instantiation of the template (2) described in Section 2.2.

4.2.3 Overall Algorithm. Putting all together, Algo. 1 depicts the CAFE algorithm. We first perform the path-sensitive 0CFA on all the submissions and obtain the result σ (line 3). Then, we derive calling contexts from the analysis result (line 4). From the calling contexts, we obtain the matching function $\mathcal{M}$ that maps each function in the buggy submission P_b to a function in a correct submission that is most likely to be useful for repair (line 5). Using the matching function, we collect repair templates (line 6). By instantiating the templates, we obtain a set of edit actions that can be applicable to P_b (lines 7 – 8). The main loop (lines 10 – 16) applies each possible sequence of the edit actions into P_b in turn. The variable n denotes the number of edit actions that can be used, which is initialized to be 1 (line 9). We apply each edit script into P_b (line 12) and check if the submission has been fixed based on the given test cases (line 13). If we have fixed the submission, we return it as a final result (line 14). Otherwise, we increase n by 1 (line 15) and repeat the main loop.

The algorithm finds a *minimal* edit script that corrects the given buggy submission.

Definition 4.6 (Minimality). Given an incorrect submission P_b and a set of possible edit actions $\mathcal{A}$, an edit script E comprising the edit actions in $\mathcal{A}$ to correct P_b is minimal if there does not exist an edit script E' such that $|E'| < |E|$ and E' fixes P_b.

4.2.4 Optimizations. When generating edit actions of kind Define that add new function definitions into a target buggy submission, we avoid functions that incur a long subsequent call chain to prevent CAFE from generating huge patches (currently, we only consider immediate callees of a reference function). Additionally, when generating edit scripts by permuting edit actions, we avoid generating duplicated edit scripts that lead to the same effect by not respecting orders between edit actions targetting different labels.

5 EVALUATION

We evaluate CAFE to answer the following research questions:

- **Performance of CAFE**: How effectively can CAFE repair incorrect programs? How does it compare to the state-of-the-art for OCaml [18]? (Section 5.1)
- **Comparison with Prior Data-Driven Approaches**: How does our approach compare to the existing data-driven approaches (Section 5.2)
- **Helpfulness**: How helpful is CAFE for students? Is the generated feedback useful for students? (Section 5.3)

We implemented CAFE in about 7,000 lines of OCaml code. Although we formalized our approach for an ideal language, our implementation can handle all student programs in our class without any modification. We used the Z3 SMT solver for checking compatibility of path conditions. All experiments were conducted on an iMac with Intel i5 CPU and 16GB memory.

5.1 Performance of CAFE

Setting. We collected 4,211 OCaml programs from 10 exercises used in our class over the last few years. To distinguish correct and incorrect programs, we used 10–33 test cases per exercise These test cases have been carefully designed to detect various types of errors over the few years. All programs are compilable with no syntax or type errors. The description of the benchmark programs is given in Table 1. We classify the programming exercises into three levels, i.e., introductory (#1-#4), intermediate (#5-#7), and advanced (#8-#10), based on the code size and the ratio of incorrect to correct programs. Although code sizes look rather small, our benchmark set includes a number of notable programs (e.g., with 7 helper functions). Some examples programs are in the supplementary material. We compared CAFE with FIXML [18], the state-of-the-art feedback generation tool for OCaml programs. FIXML repairs incorrect programs using search-based program synthesis. Because FIXML requires test cases and a solution program, we gave FIXML an instructor-provided solution for each exercise and the same set of test cases as ours. We set time budget to 60 seconds per program for both CAFE and FIXML.

Result. Table 1 shows that CAFE is far more effective than FIXML in repairing student submissions. In total, CAFE successfully fixed 82% (547/664) of the buggy submissions, while the fix rate of FIXML was 35% (234/664). Note that CAFE consistently achieves high fix rates for intermediate (82%, 95/116) and advanced problems (79%, 350/443), while FIXML does not perform well for intermediate (47%, 54/116) and advanced problems (25%, 109/443). One key contributor to the high fix rate of CAFE was its capability of modifying multiple expressions with diverse repair strategies (e.g., insertion and deletion of branches, introduction of new functions). By contrast, FIXML is limited to fixing single-location bugs.

We manually validated the correctness of patches, and Table 1 reports correct patches only. Since both CAFE and FIXML use test cases as correctness specifications, they may produce *test-suite-overfitted patches* that satisfy given test cases but still contain errors. Originally, FIXML generated 264 patches, among which 30 were overfitted to test cases. Because those patches are incorrect feedback, we only include 234 correct patches in Table 1. On the other

Table 1: Performance comparison of Cafe and FixML. “#Wrong” and “#Correct” report the numbers of incorrect and correct submissions for each problem, respectively. “#Func”: the average, minimum, and maximum numbers of functions in buggy submissions. “LOC”: the average, minimum, and maximum lines of code of buggy submissions. “Time”: average patch-generation time (in sec). “#Fix (Rate)”: #correct patches generated by each tool and the patch rate.

No	Problem Description	#Wrong	#Correct	#Func avg(min-max)	LOC avg(min-max)	FixML		Cafe	
						Time	#Fix (Rate)	Time	#Fix (Rate)
1	Finding a maximum element in a list	45	171	1.6 (1-3)	5 (1-9)	0.2	40 (89%)	0.9	45 (100%)
2	Checking membership in a binary tree	19	117	1.1 (1-3)	9 (5-13)	3.0	12 (63%)	0.7	19 (100%)
3	Mirroring a binary tree	9	88	1.0 (1-1)	6 (3-9)	0.1	7 (78%)	0.7	9 (100%)
4	Computing $\sum_{i=j}^{k} f(i)$ for j, k, and f	32	704	1.1 (1-2)	4 (2-10)	2.0	12 (38%)	2.0	29 (91%)
5	Composing functions	49	454	1.5 (1-3)	5 (2-11)	13.7	26 (53%)	1.4	42 (86%)
6	Removing redundant elements in a list	32	125	2.3 (1-5)	12 (4-24)	4.1	10 (31%)	3.8	23 (72%)
7	Arithmetic of user-defined natural numbers	35	412	2.2 (1-5)	13 (7-23)	23.0	18 (51%)	3.3	30 (86%)
8	Evaluating a propositional formula	111	597	2.1 (1-8)	29 (13-64)	1.3	44 (40%)	3.2	78 (70%)
9	Checking the validity of a lambda term	141	661	2.7 (1-7)	20 (6-47)	2.2	23 (16%)	3.1	132 (94%)
10	Dfferentiating an algebraic expression	191	218	2.1 (1-9)	29 (7-114)	1.1	42 (22%)	4.7	140 (73%)
	Total / Average	664	3,547	2.0 (1-9)	20 (1-114)	4.2	234 (35%)	3.1	**547 (82%)**

hand, Cafe produced no incorrect patches. This was because Cafe leverages common templates extracted from solutions.

More qualitative analysis on the result is as follows. Notably, Cafe successfully fixed complex programs such as one with 7 functions and the call depth of 4. The generated patches were also non-trivial. Cafe modified 31% expressions of the original programs on average. Furthermore, we found that 25% (135/547) of the fixed errors are patched by repairing at least two functions simultaneously. When we investigated the statistics of edit actions used in patches, the distribution of each four templates (Modify, Insert, Delete, Define) was 79%, 8%, 2%, and 10%, respectively.

The performance of Cafe was not very sensitive to the amount of available data. For example, when we used 50% of the correct programs as a corpus, the fix rate remained almost the same: 82% (542/664, averaged over five random trials). When we used 10% of the correct programs, the fix rate decreased to 78% (517/664).

Limitation. We identified two representative cases that Cafe can fail. Timeout due to the large search space was the most common reason. For example, a buggy submission required Cafe to modify all (eight) if-then-else expressions in the program; Cafe is unlikely to generate such a large fix. Also, Cafe sometimes failed due to the lack of proper patch candidates caused by matching failure.

5.2 Comparison with Prior Techniques

We could not compare Cafe directly with existing data-driven tools as they target different languages [11, 30, 39] and rely on language-specific features [30]. However, to see how much Cafe advances the existing techniques, we implemented two variants, called Prog and Func, of Cafe. Prog and Func are identical to Cafe except that

- Prog uses the program-level matching of Sarfgen [39],
- Func uses it at the function level.

From the corpus of correct programs, Prog selects a program that is most similar to the given incorrect program, where we compute the similarity using the technique of Sarfgen [39], i.e., position-aware characteristic vector embedding. Thus, the performance gap between Prog and Cafe hints at how Cafe performs compared to Sarfgen. Func applies the matching algorithm of Sarfgen at the function level and therefore it partially enjoys the benefit of our approach (i.e., using multiple solutions via function-level matching). Thus, comparing Func and Cafe shows the sole impact of using our context-aware matching against the matching of Sarfgen.

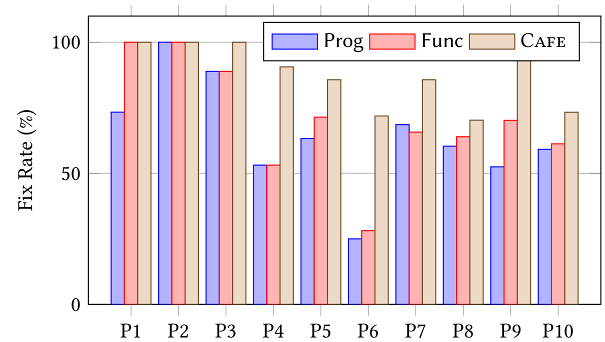

Figure 6: Comparison with existing data-driven techniques.

The result shows that existing data-driven techniques are unlikely to be effective for our dataset. Figure 6 compares the fix rates of Prog, Func, and Cafe on the same dataset as Figure 1. On average, Prog achieved a fix rate of 59%. Simply extending the existing technique at the function level (Func) did not improve its performance significantly (67%). This is because, as illustrated in Section 2.2, simply aiming to find syntactically or semantically similar functions is unlikely to find a useful reference.

5.3 User Study

We recruited 16 undergraduate students from our course. The students were asked to solve the 10 programming exercises in Table 1. Then, we graded their submissions and provided feedback for erroneous ones using Cafe. Finally, students answered the survey questions about the generated feedback: (Q1) Is the feedback correct and easy to understand? (Q2) Does the provided feedback help you understand the mistake? (Q3) Do you think Cafe can be actually useful in our class? For each question, students were asked to choose between 1 (strongly disagree) and 5 (strongly agree). Also,

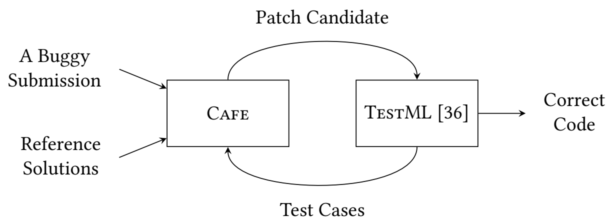

Figure 7: Enhanced Cafe with automatic test generator

Table 2: Cafe with automatic test generation.

No	Cafe w/o TestML			Cafe with TestML		
	#Tests	#Fix	Time	#Tests (min-max)	#Fix	Time
8	18	78	3.2	4 (1-14)	79	124.5
9	33	132	3.1	3 (1-8)	133	123.6
10	28	140	4.7	5 (1-13)	141	124.8

we asked student to leave additional textual comments for each question. For Q1, Q2, and Q3, 14, 13, and all participants agreed (with scores 4 or 5), respectively. The comments the participants left include: "Cafe generates clever patches while keeping my code's structure", "Unlike the TA's solution code, the personalized feedback is easier to understand because it is derived form mine", and "It will help because it not only tells me the error, but also teaches me how to fix it".

5.4 Cafe with Automatic Test Case Generation

A limitation of Cafe is the reliance on manually-provided test cases, which hinders its use in real deployment: the quality of feedback depends on the quality of test cases but coming up with high-quality test cases requires massive human effort. In Table 1, we used manual test cases carefully refined over the few years, but such test cases are not always available. Below, we check if this limitation can be alleviated with the aid of automatic test generation techniques.

As shown in Figure 7, we built an enhanced version of Cafe in combination with TestML [36]. TestML is a recent counter-example generation tool for OCaml programming exercises, which takes two programs and tries to generate a test case on which the two programs behave differently. The enhanced Cafe in Figure 7 combines Cafe and TestML in a loop. Note that it no longer requires manual test cases for patch validation. Instead, it uses TestML as a correctness oracle. When TestML fails to generate a counter-example for the submission and a solution (randomly chosen from the corpus), we regard the current patch candidate as a correct repair. Otherwise, TestML augments the set of test cases, which is initially empty, with the generated counter-example, and Cafe is re-run with new test cases. The loop repeats until one of two components fails. We set the time budget for Cafe and TestML to 60 and 120 seconds, respectively.

Table 2 compares Cafe with and without TestML (results for Problems 8–10 only due to the lack of space). For Cafe without TestML, #Tests reports the number of manual test cases for each exercise. For Cafe with TestML, #Tests shows the average, smallest, and largest number of test cases generated by TestML during the process in Figure 7. The results show that the enhanced system with TestML reproduces the results in Table 1. Indeed, Cafe with TestML generated four new patches and we confirmed they are correct. This was because Cafe now uses a smaller number of test cases and spends less time in patch validation. Despite the overhead, we found that combining Cafe and TestML increases the usability significantly by reducing the instructor's burden of crafting test cases and validating generated patches.

6 RELATED WORK

Automatic feedback generation has received an increasing amount of attention over the last years [1, 2, 5, 6, 9–13, 15, 16, 18, 19, 23, 27, 29, 31, 32, 35, 36, 39]. We discuss closely-related work below.

Our work represents a significant departure from prior data-driven feedback generation techniques. Clara [11] is a clustering-based method that uses control flows and dynamic traces to find a correct solution. Similarly, Sarfgen [39] represetns ASTs as vectors to find similar solution programs. These syntactic approaches could be improved using semantic features [30, 38]; however, those features are designed with imperative languages in mind and not readily applicable to Cafe. All of these data-driven approaches [11, 30, 38, 39] assume that there exists a *close enough* correct program in the corpus. To our knowledge, Cafe is the first to address this limitation by leveraging multiple, partially-matching programs.

Rite [33] and FixML [18] are state-of-the-art techniques for repairing student programs written in OCaml. Unlike ours, however, Rite can fix type errors only. FixML [18] uses program synthesis to repair general errors specified by test cases. However, FixML cannot fix multi-location errors, which consequently leads to a low fix rate. In this paper, we showed that Cafe outperforms FixML.

AutoGrader [35] and CoderAssist [15] are approaches that require manual effort. AutoGrader is a model-based technique to generate feedback by using constraint-based program synthesis. sk_p [31] addresses the limitation of AutoGrader by using a seq2seq neural network but is limited to small Python programs. CoderAssist generates verified feedback for introductory programming assignments but requires instructor-validated submissions.

Our work belongs to program repair techniques that use test cases as correctness criteria. Automatic program repair has a large volume of prior work [8, 26], which broadly classified into techniques for particular error types [4, 7, 14, 17, 22, 34, 37, 40, 43] and general-purpose techniques [3, 20, 21, 24, 25, 41, 42]. In particular, our work is similar to [24] in that both techniques use reference programs but our goal is to provide feedback on student programs.

7 CONCLUSION

We presented a new technique that advances the existing data-driven approaches for automatically generating feedback on programming assignments. Unlike prior approaches, which works under the assumption that close enough reference programs exist in the corpus, Cafe can repair an incorrect submission by using multiple, partially-matching reference programs. To achieve this, we presented a new, context-aware repair algorithm. Evaluation results with real student submissions show that Cafe has a high fix rate and produces quality feedback actually useful for students.

REFERENCES

[1] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani. 2018. Compilation Error Repair: For the Student Programs, From the Student Programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 78–87.

[2] Umair Z. Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the Pedagogical Benefits of Adaptive Feedback for Compilation Errors by Novice Programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training* (Seoul, South Korea) *(ICSE-SEET '20)*. Association for Computing Machinery, New York, NY, USA, 139–150. https://doi.org/10.1145/3377814.3381703

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

[4] Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2017. IntPTI: Automatic Integer Error Repair with Proper-type Inference. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 996–1001. http://dl.acm.org/citation.cfm?id=3155562.3155693

[5] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantiative Objectives. In *27th International Conference on Computer Aided Verification (CAV 2016)* (27th international conference on computer aided verification (cav 2016) ed.).

[6] A. Drummond, Y. Lu, S. Chaudhuri, C. Jermaine, J. Warren, and S. Rixner. 2014. Learning to Grade Student Programs in a Massive Open Online Course. In *2014 IEEE International Conference on Data Mining*. 785–790. https://doi.org/10.1109/ICDM.2014.142

[7] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-leak Fixing for C Programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 459–470. http://dl.acm.org/citation.cfm?id=2818754.2818812

[8] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (Jan 2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[9] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 7 (March 2015), 35 pages. https://doi.org/10.1145/2699751

[10] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2014. Feedback Generation for Performance Problems in Introductory Programming Assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 41–51. https://doi.org/10.1145/2635868.2635912

[11] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 465–480. https://doi.org/10.1145/3192366.3192387

[12] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Neural Attribution for Semantic Bug-Localization in Student Programs. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 11884–11894.

[13] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale* (Cambridge, Massachusetts, USA) *(L@S '17)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/3051457.3051467

[14] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: Scalable, Precise, and Safe Memory-Error Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 271–283. https://doi.org/10.1145/3377811.3380323

[15] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-Supervised Verified Feedback Generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 739–750. https://doi.org/10.1145/2950290.2950363

[16] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I. Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. 2016. Apex: Automatic Programming Assignment Error Explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 311–327. https://doi.org/10.1145/2983990.2984031

[17] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: Static Analysis-based Repair of Memory Deallocation Errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 95–106. https://doi.org/10.1145/3236024.3236079

[18] Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh. 2018. 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

[19] X. Liu, S. Wang, P. Wang, and D. Wu. 2019. Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 126–137. https://doi.org/10.1109/ICSE-SEET.2019.00022

[20] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[21] 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* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[22] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: Automated End-to-end Repair at Scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada). IEEE Press, Piscataway, NJ, USA, 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[23] V. J. Marin, T. Pereira, S. Sridharan, and C. R. Rivero. 2017. Automated Personalized Feedback in Introductory Java Programming MOOCs. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1259–1270. https://doi.org/10.1109/ICDE.2017.169

[24] 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* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 129–139. https://doi.org/10.1145/3180155.3180247

[25] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[26] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (Jan. 2018), 24 pages. https://doi.org/10.1145/3105906

[27] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: Scalable Homework Search for Massive Open Online Programming Courses. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14)*. Association for Computing Machinery, New York, NY, USA, 491–502. https://doi.org/10.1145/2566486.2568023

[28] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg.

[29] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic Grading and Feedback Using Program Repair for Introductory Programming Courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 92–97. https://doi.org/10.1145/3059009.3059026

[30] David M. Perry, Dohyeong Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: Clustering of Imperative Programming Assignments Based on Quantitative Semantic Features. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 860–873. https://doi.org/10.1145/3314221.3314629

[31] 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* (Amsterdam, Netherlands) *(SPLASH Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 39–40. https://doi.org/10.1145/2984043.2989222

[32] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 404–415. https://doi.org/10.1109/ICSE.2017.44

[33] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and*

Implementation (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/3385412.3386005

[34] Alex Shaw, Dusten Doggett, and Munawar Hafiz. 2014. Automatically Fixing C Buffer Overflows Using Program Transformations. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, Washington, DC, USA, 124–135. https://doi.org/10.1109/DSN.2014.25

[35] 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* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195

[36] 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 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360614

[37] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[38] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embeddings for Program Repair. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJuWrGW0Z

[39] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 481–495. https://doi.org/10.1145/3192366.3192384

[40] Westley Weimer. 2006. Patches as better bug reports. In *Proceedings of the 5th international conference on Generative programming and component engineering*. ACM, 181–190.

[41] 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

[42] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 416–426. https://doi.org/10.1109/ICSE.2017.45

[43] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: Value-flow-guided Precise Program Repair for Null Pointer Dereferences. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 512–523. https://doi.org/10.1109/ICSE.2019.00063