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

Dowon Song  
Korea University  
Republic of Korea  
dowon_song@korea.ac.kr

Woosuk Lee  
Hanyang University  
Republic of Korea  
woosuk@hanyang.ac.kr

Hakjoo Oh∗  
Korea University  
Republic of Korea  
hakjoo_oh@korea.ac.kr

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 83% of incorrect submissions, far outperforming existing approaches.

CCS CONCEPTS
• Software and its engineering → Automatic programming.

KEYWORDS
Program Repair, Program Synthesis

ACM Reference Format:

1 INTRODUCTION
In recent years, there has been a surge of interest in automatic feedback generation for programming assignments [1, 5, 12, 19, 20, 22, 33, 35, 36, 39, 40, 43]. 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’ 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.

Data-Driven Feedback Generation. Among prior techniques, data-driven approaches [12, 17, 34, 42, 43] are arguably the current state-of-the-art. The main 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, 19, 39], 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 [12] and Sarfgen [43], assume a solution exists that is equivalent to the incorrect program modulo control flows. This assumption, however, does not hold always [22], especially when providing feedback beyond introductory-level exercises. In this case, constructing a corpus with the close-program assumption becomes a challenge.

Our Approach. 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 83% (548/664) of incorrect submissions, vastly outperforming FrxML [22], 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 SAGEREN [43] decreased the fix rate from 83% to 59%. Finally, we conducted a user study, which shows CAFE is actually helpful for students.

Contributions. 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 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 \(2x\).

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; Var "y"], 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 differ.

Compared to existing data-driven techniques [12, 34, 42, 43], 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 gettmes 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 [12, 34, 42, 43] 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 operation \(o \in \{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 (\(P: apply1, P_1: apply1,\) 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 (\(inc\_\_all\ tl\) 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: \(inc\_\_all\sim add\_\_list, dec\_\_all\sim sub\_\_list,\) and apply\_\_all = 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 \(\delta_{dec\_\_all}, \delta_{add\_\_list},\) and \(\delta_{sub\_\_list}\), respectively) are as follows:

\[
\begin{align*}
\delta_{dec\_\_all} & = \delta_{add\_\_list} = \delta_{sub\_\_list} = (l, 0) \land l = (hd :: tl) \land (o = SUB) \\
\delta_{add\_\_list} & = (l_1, o_1) \land l_1 = (hd \_\_1 :: tl) \land (o_1 = ADD) \\
\delta_{sub\_\_list} & = (l_2, o_2) \land l_2 = (hd_2 :: tl_2) \land (o_1 = SUB)
\end{align*}
\]
Context-Aware and Data-Driven Feedback Generation for Programming Assignments ESEC/FSE ’21, August 23–28, 2021, Athens, Greece

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

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

Figure 3: A running example to illustrate how CAFE works

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 δ_{dec_all} ∧ δ_{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 δ_{dec_all} ∧ δ_{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 [12, 34, 42, 43]. 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;
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 $\text{dec\_all}$ 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:

$$
e \in \text{Exp} \quad \langle \text{Expressions} \rangle \quad x \in \text{VId} \quad \langle \text{Variables} \rangle$$

$$f \in \text{Fld} \quad \langle \text{Functions} \rangle \quad \text{Id} = \text{VId} \cup \text{Fld} \quad \langle \text{Identifiers} \rangle$$

$$\ell \in \text{Label} \quad \langle \text{Labels} \rangle \quad \tau \in \text{Type} \quad \langle \text{Types} \rangle$$

$$e \ ::= \ n \mid x \mid \lambda x.e \mid e_1 \mathbf{\oplus} e_2 \mid \text{rec } f(x) = e_1 \text{ in } e_2$$

$$\mathbf{\oplus} \quad \langle \text{Binary Operations} \rangle$$

We assume each expression is associated with a unique label. Expression $e$ associated with a label $\ell \in L$ is denoted by $e^\ell$. For the sake of better readability, we will often elide $\ell$ 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 $e_1 e_2$, $x$ ranges over data type constructors, $a(\kappa)$ denotes the arity of $\kappa$, $\kappa^{-1}$ denotes a destructor which extracts the $i$-th subcomponent of a constructor $\kappa$, 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 $\kappa$, or the underscore (_) called the wildcard pattern. We use $p_1 \mathbf{\rightarrow} p_2^k$ to denote $p_1 \rightarrow e_1 \mid \cdots \mid p_k \rightarrow e_k$. Types include the integer type int, user-defined algebraic data types $T$, and function types $\tau_1 \rightarrow \tau_2$.

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

Setting. We assume that each student submission $P \in \text{Exp}$ has no type error, and is in the following form: let rec $f(x) = e$ in $f \ x_i$ where $f$ is a top-level function that the student is asked to implement, and $x_i$ 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 convenience, 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) body : Fld → Exp that returns the body of a function, (2) param : Fld → Vld that returns the parameter variable of a function, and (3) type : (Label ∪ Id) → 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 $T \subseteq \text{Val} \times \text{Val}$ is used to determine the correctness of each submission. The submission $P$ is correct (denoted $\text{correct}(P; T)$) iff $\forall (i, o) \in T, (\lambda x_i. P) i \rightarrow^* o$. Otherwise, the submission is buggy.

Our problem is defined as follows: given a buggy submission $P_b \in \text{Exp}$, a corpus of correct submissions $P_c \subseteq \text{Exp}$, and a set of test cases $T$, derive a correct submission $P \notin P_c$ 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 $\delta \in \text{Ctx}$ is a path condition on the input variable under which a function is invoked. Formally, path conditions are defined below:

$$\delta := \text{true} \mid \text{false} \mid e = e \mid \neg \delta \mid \delta \land \delta.$$  

Calling contexts are defined as follows:

**Definition 4.1 (Calling context).** A calling context is a triple $(f, \delta, g) \in \text{Fld} \times \text{Ctx} \times \text{Fld}$ where $f$ is a function, $g$ is another function called in the body of $f$, and $\delta$ 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 [32], 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 $\sigma \in (\text{Id} \cup \text{Label}) \times \text{Ctx} \rightarrow \mathcal{P}(\text{Exp}) \cup \{T\}$.

Figure 4 depicts a subset of the constraint generation rules; the full set can be found in the supplementary material. The judgement $\delta \vdash [e]^l \leftarrow C$ can be read as "the analysis of expression $e$ with label $l$ generates set constraints $C$ over dataflow state $\sigma$ under a current path condition $\delta". While solving the constraints via a least-fix point computation, special constraints of kinds $\text{fn}$, $\text{cn}$, and $\text{pat}$ are interpreted by the constraint solver to generate additional concrete constraints by referring to an intermediate analysis result. For example, from a constraint $\text{fn}_1 \cdot \ell_1 \rangle \ell_2 \rightarrow \ell$, for every function $\lambda x_i e_i^l$ that the analysis (eventually) concludes the expression labeled $\ell_1$ may evaluate to, additional constraints are generated to capture value flow from the actual argument expression $\ell_2$ to formal function argument $x$, and from the function result to the calling expression $\ell$. 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 true $\vdash \bigcap_i P_i^{\ell_i} \leftarrow C_i$, and obtain the least solution $\sigma_i$. We collect a set of calling contexts $\Delta = \Delta_1 \cup \cdots \Delta_m$ where each $\Delta_i$ is

$$\bigcup_{f \in \text{functions}(P_i)} \{(f, \delta, g) \mid (e_1^l, e_2) \in \text{Sub(body}(f)), g \in \sigma_i(\ell_1, \delta)\}.$$  

We also conjoin the analysis results from the submissions and obtain $\sigma \subseteq \bigcup_{1 \leq i \leq m} \sigma_i$, 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 $l_2$ and $o_2$ in apply2 where the initial path condition is true, we first generate the following constraints over a dataflow state $\sigma$.

$$\text{pair}^{-1}(i) \in \sigma(o_2, \text{true}) \quad (3)$$

$$\text{pair}^{-2}(i) \in \sigma(o_2, \text{true}) \quad (4)$$

In the outer pattern matching match $l_2$ with cons($\text{hd}_2, \text{tl}_2$) → ..., from (3), we generate the following constraint making $o_2$ under a path condition: $\sigma(o_2, \text{true}) \subseteq \sigma(o_2, \text{pair}^{-1}(i) = \text{cons(hd}_2, \text{tl}_2)).$

Under the current path condition $\delta$, in the inner pattern matching match $o_2$ with $\text{SUB} \rightarrow \cdots$ sub_list $\text{tl}_2$ where we assume sub_list is associated with label $l$, the constraint sub_list $\in \sigma(l, \delta \land \text{pair}^{-2}(i) = \text{SUB})$ is generated from (4). After computing a least solution satisfying these constraints, we obtain a calling context (apply2, $\delta \land \text{pair}^{-2}(i) = \text{SUB}, \text{sub_list}$).

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:

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

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

$$C_{\text{in}}^{f,g} = \{(\delta, \delta') \mid \delta, \delta' \in \text{Ctx}, (., \delta, f), (., \delta', g) \in \Delta, \text{SAT}(\delta \land \delta')\}$$

$$C_{\text{out}}^{f,g} = \{(\delta, \delta') \mid \delta, \delta' \in \text{Ctx}, (f, \delta), (g, \delta', .) \in \Delta, \text{SAT}(\delta \land \delta')\}$$

If both $f$ and $g$ do not have any callers (resp. callee), we do not consider the term involving $C_{\text{in}}^{f,g}$ (resp. $C_{\text{out}}^{f,g}$). 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
With distance ASTs into numerical vectors, which is called the position-aware characteristic vectors proposed by Wang et al. [43]. We compute the syntactic similarity, we use the method of embedding.

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.

function in a buggy submission and returns a function in a correct submission to be referred for correction:

\[ M = \lambda P_f. \{ f \mapsto \text{argmin } \text{dist}(f, g) \mid f \in \text{functions}(P_f) \}. \]

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.

\[
\begin{align*}
&\text{apply1}(i) = \text{cons}(\text{hd}, \text{tl}) \land \text{pair2}(i) = \text{SUB, dec_all}) \\
&(\text{dec_all}, d_1 \land \text{cons}(\text{pair2}(i)) = \text{cons(h, t, dec_all)}) \\
&(\text{apply1}, \text{pair2}(i) = \text{cons(h, t1, 1) \land \text{pair2}(i) = \text{ADD, add_list})} \\
&(\text{add_list}, d_0 \land \text{cons}(\text{pair2}(i)) = \text{cons(h, t1, add_list)}) \\
&(\text{apply2}, \text{pair2}(i) = \text{cons(h, t2, 1) \land \text{pair2}(i) = \text{SUB, sub_list})} \\
&(\text{sub_list}, d_0 \land \text{cons}(\text{pair2}(i)) = \text{cons(h, t2, sub_list)}) \\
&\ldots
\end{align*}
\]

With \(w_1 = 1\) and \(w_2 = 2\) that we are using in our implementation,

\[
\text{dist}(\text{dec_all, sub_list}) = |(d_1, d_2, d_3)|^{-1} + 2 \cdot |(d_4, d_5)|^{-1}
\]

whereas \(\text{dist}(\text{dec_all, add_list}) = \infty\) as the two functions do not share any compatible calling contexts.

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. [43]. 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** \((f, e)\) replaces the old subexpression at label \(f\) by the new expression \(e\).
- **Insert** \((f, p \rightarrow e)\) adds a new pattern matching case \(p \rightarrow e\) into a match expression associated with label \(f\).
- **Delete** \((f, p \rightarrow e)\) removes an existing pattern matching case \(p \rightarrow e\) from a match expression associated with label \(f\).
- **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 \(f(\text{param}()) = \text{body}(f)\) 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 \(\text{Exp}_h\) of expressions with holes is similarly defined as \(\text{Exp}\) in the following:

\[
\begin{align*}
\epsilon_1 &\in \text{Exp}_h \\
\epsilon_2 &::= n \mid \lambda x. \epsilon_3 \mid \epsilon_1 \oplus \epsilon_2 \mid \epsilon_2 \oplus \epsilon_1 \\
&\mid \kappa(\epsilon_1, \ldots, \epsilon_{\text{arity}}) \mid \kappa^{-1}(\epsilon) \\
&\mid \text{let } x = \epsilon_1 \text{ in } \epsilon_2 \\
&\mid \text{let rec } f(x) = \epsilon_1 \text{ in } \epsilon_2 \\
&\mid \text{match } \epsilon_2 \text{ with } \epsilon_3 \rightarrow \epsilon_4 \\
\rho &::= \kappa(\epsilon_1, \ldots, \epsilon_n) \mid _-
\end{align*}
\]

An expression with holes can be considered an abstraction of multiple expressions. The abstraction function \(\alpha_e : \text{Exp} \rightarrow \text{Exp}_h\), which we apply to expressions in correct submissions to extract templates, is defined as follows (to avoid unnecessary clutter, we omit simple
4.2.2 Generating Edit Scripts.

We instantiate the collected templates into edit actions using the following concretization function $\gamma^P_e : \mathcal{P}(\text{Exp}) \rightarrow \mathcal{P}(\text{Exp})$ parametrized by a buggy submission $P$.

$$\gamma^P_e(n) = \{n\} \quad \gamma^P_e(\lambda x.e) = \{\lambda x.e \mid e \in \gamma^P_e(e_0)\}$$

... 

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 $\mathcal{T} = T_D \cup T_M$ where $T_D$ is a set of templates of kind Define defined as follows:

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

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

$$T_M = \bigcup \{T \mid f \in \text{functions}(P), \llbracket \text{body}(f), \text{body(M(f))} \rrbracket \sim T\}.$$ 

The judgement $\llbracket e, e' \rrbracket \sim T$ can be read as "by differing a buggy expression $e$ and a reference expression $e'$, we extract a set $T$ of 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 $\text{sub_list}$ for correcting $\text{dec_all}$ in Figure 3. We extract templates $T$ such that $\llbracket e_1, e_2 \rrbracket \sim T$ where $e_1$ is $\text{body(\text{sub_list})}$ with labels $\ell_1, \delta_1$ and $e_2$ is $\text{body(\text{dec_all})}$ with labels $\ell'_1, \delta_1$:

- $e_1 = p_1 \rightarrow \text{empty}$
  - $p_2 \rightarrow \text{cons}(h^t + 1, \ell_1, \text{dec_all} \ell_1, t^t)\delta_1$
- $e_2 = p'_1 \rightarrow \text{empty}$
  - $p_2' \rightarrow \text{cons}(h^t, \ell_1, \text{sub_list} \ell_1, t^t)\delta_1$

and $p_1 = p'_1 = \text{empty}$, $p_2 = \text{cons}(h, t)$, and $p_2' = \text{cons}(h_2, t_2)$.

By the inference rule for differing two matching expressions in Figure 5, we first compare $1^t$ and $2^t$ and derive a template $\text{Modify}(\ell_1, \ell_2 \text{list})$. Since both $\{a_\ell(p_1), a_\ell(p_2)\}$ and $\{a_\ell(p'_1), a_\ell(p'_2)\}$ are $\{\text{empty, cons(\ell_1, \text{list\_int\_list})}\}$ the expressions matched for $p_1$ and $p'_1$ are the same, we compare the matched expressions for $p_2$ and $p_2'$ labeled $\delta_1$ and $\delta'_1$ respectively. By the rule for differing two constructors, we additionally derive $\text{Modify}(h, \ell_1, \ell_2 - 1) = \text{Modify}(\ell_2, \text{list\_int\_int\_list})$, $\text{Modify}(\ell_1, \text{list\_int\_list\_int\_list})$, and $\text{Modify}(\ell_2, \text{list\_int\_list\_int\_list})$.

4.2.2 Generating Edit Scripts. We instantiate the collected templates into edit actions using the following concretization function

$$y^P_e \{ V_e \mid \gamma^P_e(\ell_1) \} \quad (V_e \mid \ell \neq 0)$$

where $V_e \mid \ell$ is the set of variables of type $\ell$ in $V$ that may take the same value reachable at label $\ell$. Formally,

$$V_e \mid \ell = \{ x \in V \mid \text{type}(x) = \ell, \exists \delta, \delta', \sigma(x, \delta) \cap \sigma(\ell, \delta') \neq \emptyset \}.$$ 

This heuristic is inspired by the variable-usage based $\alpha$-conversion of SARGEN [43] but we analyze the usage more accurately using the result of our path-sensitive 0-CFA.

**Example 4.5.** Recall the template $\text{Modify}(\ell_1, \ell_2 \text{list} - 1)$ derived in Example 4.4. When concretizing $\ell_2 \text{list}$, we only consider the variable $h$ as a candidate for the hole because

$$\sigma(h, \delta_1) \equiv \text{cons}^{-1}(\text{cons}^{-2}(\text{pair}^{-1}(1)))$$

and $\delta_1$ and $\delta_2$ 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 accordingly. An example case is the instantiation of the template (2) described in Section 2.2.
\[
\begin{align*}
\text{Input:} & \quad A \text{ buggy submission } P_b, \text{ a set of correct submissions } \mathcal{P}_c, \text{ and a set of test cases } \mathcal{T} \\
\text{Output:} & \quad \text{A program } \mathcal{P}_c \text{ satisfying all the test cases in } \mathcal{T} \\
& \quad \begin{array}{l}
\mathcal{A} \leftarrow \emptyset \quad \triangleright \text{Set of edit actions} \\
\mathcal{P} \leftarrow \mathcal{P}_c \cup \{P_b\} \\
\sigma \leftarrow \text{result of the path-sensitive OCAFA on } \mathcal{P} \\
\Delta \leftarrow \text{all calling contexts derivable from } \sigma \\
\mathcal{M} \leftarrow \Delta \\
\mathcal{T} \leftarrow \text{ExtractTemplates}(\mathcal{M}, \Delta, P_b, \mathcal{P}_c) \\
\text{for } T \in \mathcal{T} \text{ do} \\
\mathcal{A} \leftarrow \mathcal{A} \cup \text{InstantiateTemplate}(T, P_b, \sigma) \\
n \leftarrow 1 \\
\text{repeat} \quad \triangleright E: \text{edit script comprising } n \text{ edit actions} \\
\text{for each permutation } E \text{ of } n \text{ elements of } \mathcal{A} \text{ do} \\
P \leftarrow \text{apply } E \text{ into } P_b \\
\text{if correct}(P, T) \text{ then} \\
\quad \text{return } P \\
n \leftarrow n + 1 \\
\text{until } n \leq |\mathcal{A}| 
\end{array}
\end{align*}
\]

Algorithm 1 The CAFE Algorithm

- **Input:** A buggy submission \(P_b\), a set of correct submissions \(\mathcal{P}_c\), and a set of test cases \(\mathcal{T}\)
- **Output:** A program \(\mathcal{P}_c\) satisfying all the test cases in \(\mathcal{T}\)

4.2.3 Overall Algorithm. Putting all together, Alg. 1 depicts the CAFE algorithm. We first perform the path-sensitive OCAFA on all the submissions and obtain the result \(\sigma\) (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 enumerates edit scripts in increasing size, guaranteeing to find a minimal edit script in the following sense.

**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 targeting different labels.

4.3 Discussion

**Limitation of Context-Aware Matching.** Our context-aware matching may produce imprecise results for functions called with trivial contexts. For example, suppose that there are two functions called under empty contexts (i.e., no path conditions are accumulated in their callers), and they should not be matched. In this case, context-aware matching would match them because their contexts are equivalent as empty contexts. We mitigated this shortcoming by applying the idea of context tunneling [18] and updating contexts at call-sites only when they are non-empty. For example, suppose we measure the similarity between two functions that have empty incoming contexts. In this case, we apply context tunneling, so the callee inherits the callers’ incoming contexts rather than producing empty contexts. We can further mitigate the issue of trivial contexts by falling back to existing syntactic matching: when the contexts are trivial (even after context tunneling), we can use the syntactic matching of Sarfgen rather than our context-aware matching.

**Applicability to Other Languages.** Although we formalized our approach for a functional language, the ideas of CAFE may work for other languages (e.g., Python) as well. First, the core idea can be applicable to any programs that consist of multiple functions, regardless of whether the language is functional or imperative. Second, the ideas of extracting repair templates from syntactic discrepancies and instantiating obtained templates can also be easily adapted for other languages whose syntax is defined inductively.

One feature of OCaml that CAFE particularly assumes is that it is a statically typed language. CAFE uses static type information to
prune out unnecessary function/variable mappings (i.e., excluding type-inconsistent pairs). This type-based optimization can be readily used for imperative languages with static typing such as C and Java. For dynamic languages like Python, CAFE is still applicable simply without the optimization (which might degrade the performance of CAFE slightly) or with extra type information computed by a static analysis.

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 [22]? (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, which include most of the benchmarks used in our prior work [22].

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 [22], the state-of-the-art feedback generation tool for OCaml programs.

**Result.** Table 1 shows that CAFE is far more effective than FixML in repairing student submissions. In total, CAFE successfully fixed 83% (548/664) 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%, 351/443), 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 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% (135/548) 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 representative cases that CAFE can fail to produce patches. 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. In our experiments, potential imprecise matching discussed in Section 4.3 rarely happened (after applying context tunneling). For example, in problem 10, we observed that only five failures (9.8%, 5/51) were due to imprecise matching.

5.2 Comparison with Prior Techniques

We could not compare CAFE directly with existing data-driven tools as they target different languages [12, 34, 43] and rely on language-specific features [34]. 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 [43],
- 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 [43], 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).

\[^{2}\text{We excluded four exercises from [22] because they do not contain sufficient number of programs, or they require program-specific testing drivers that make the specification of problem unclear.}\]
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.

<table>
<thead>
<tr>
<th>No</th>
<th>Problem Description</th>
<th>#Wrong</th>
<th>#Correct</th>
<th>#Func avg(min-max)</th>
<th>LOC avg(min-max)</th>
<th>FixML Time</th>
<th>#Fix (Rate)</th>
<th>CAFE Time</th>
<th>#Fix (Rate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Finding a maximum element in a list</td>
<td>45</td>
<td>171</td>
<td>1.6 (1-3)</td>
<td>5 (1-9)</td>
<td>0.3</td>
<td>40 (89%)</td>
<td>0.0</td>
<td>45 (100%)</td>
</tr>
<tr>
<td>2</td>
<td>Checking membership in a binary tree</td>
<td>19</td>
<td>117</td>
<td>1.1 (1-3)</td>
<td>9 (5-15)</td>
<td>3.1</td>
<td>12 (63%)</td>
<td>0.0</td>
<td>19 (100%)</td>
</tr>
<tr>
<td>3</td>
<td>Mirroring a binary tree</td>
<td>9</td>
<td>88</td>
<td>1.0 (1-1)</td>
<td>6 (3-9)</td>
<td>0.1</td>
<td>7 (78%)</td>
<td>0.0</td>
<td>9 (100%)</td>
</tr>
<tr>
<td>4</td>
<td>Computing (\sum_{k=1}^{n} f(i)) for (j, k,) and (f)</td>
<td>32</td>
<td>704</td>
<td>1.1 (1-2)</td>
<td>4 (2-10)</td>
<td>1.6</td>
<td>12 (38%)</td>
<td>2.1</td>
<td>29 (91%)</td>
</tr>
<tr>
<td>5</td>
<td>Composing functions</td>
<td>49</td>
<td>454</td>
<td>1.5 (1-3)</td>
<td>5 (2-11)</td>
<td>11.8</td>
<td>26 (53%)</td>
<td>1.1</td>
<td>42 (86%)</td>
</tr>
<tr>
<td>6</td>
<td>Removing redundant elements in a list</td>
<td>32</td>
<td>125</td>
<td>2.3 (1-3)</td>
<td>12 (5-24)</td>
<td>4.1</td>
<td>10 (31%)</td>
<td>3.0</td>
<td>23 (72%)</td>
</tr>
<tr>
<td>7</td>
<td>Arithmetic of user-defined natural numbers</td>
<td>35</td>
<td>412</td>
<td>2.2 (1-5)</td>
<td>13 (7-25)</td>
<td>23.3</td>
<td>18 (51%)</td>
<td>1.0</td>
<td>30 (86%)</td>
</tr>
<tr>
<td>8</td>
<td>Evaluating a propositional formula</td>
<td>111</td>
<td>597</td>
<td>2.1 (1-8)</td>
<td>29 (13-64)</td>
<td>1.5</td>
<td>44 (40%)</td>
<td>0.3</td>
<td>78 (70%)</td>
</tr>
<tr>
<td>9</td>
<td>Checking the validity of a lambda term</td>
<td>141</td>
<td>661</td>
<td>2.7 (1-7)</td>
<td>20 (6-47)</td>
<td>1.5</td>
<td>23 (16%)</td>
<td>1.8</td>
<td>133 (94%)</td>
</tr>
<tr>
<td>10</td>
<td>Differentiating an algebraic expression</td>
<td>191</td>
<td>218</td>
<td>2.1 (1-9)</td>
<td>29 (7-114)</td>
<td>1.1</td>
<td>42 (22%)</td>
<td>3.0</td>
<td>140 (73%)</td>
</tr>
<tr>
<td>Total / Average</td>
<td>664</td>
<td>3,547</td>
<td>2.0 (1-9)</td>
<td>20 (1-114)</td>
<td>3.9</td>
<td>234 (35%)</td>
<td>1.6</td>
<td>548 (83%)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Comparison with existing data-driven techniques.

Thus, comparing Func and CAFE shows the sole impact of using our context-aware matching against the matching of SARFGEN.

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. CAFE was slightly superior to the existing syntactic approach in terms of efficiency and patch quality. When we compared the CAFE to the variant of SARFGEN (Prog), CAFE was faster than Prog (1.6 seconds vs. 1.9 seconds), which implies that context-aware matching can be as efficient as the embedding method of SARFGEN. In addition, we found that CAFE modified less expressions than Prog (31% and 33% respectively).

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, 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 [40]. 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 three 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, 7, 10–14, 19, 20, 22, 23, 27, 31, 33, 35, 36, 39, 40, 43]. We discuss closely-related work below.

Our work builds upon but represents a significant departure from prior data-driven feedback generation techniques [12, 17, 34, 42, 43]. Clara [12] is a clustering-based method that uses control flows and dynamic traces to find a correct solution. Similarly, Sarfgen [43] represents ASTs as vectors to find similar solution programs. These syntactic approaches could be improved using semantic features [34, 42]; however, those features are designed with imperative languages in mind and not readily applicable to CAFE. These data-driven approaches assume that there exists a close enough correct program in the corpus and CAFE aims to address this limitation by leveraging multiple, partially-matching programs.

Refactory [17] resolves the strict control flow matching problem of existing works [12, 43] by generating semantic-preserving references through refactoring. Note, however, that Refactory still relies on the control flow structures of refactored reference programs. On the other hand, our key observation in this paper is that conventional syntactic/semantic matching is not suitable for finding useful references when target programs consist of multiple sub-functions. In this setting, “similar” reference functions are unlikely to be useful for repair (Section 2.2). To address this issue, we present a new method, context-aware matching, which matches functions by analyzing how they are used in programs (rather than comparing syntactic similarity).

Rite [37] and FixML [22] are state-of-the-art techniques for repairing student programs written in OCaml. Unlike ours, however, Rite can fix type errors only. FixML [22] 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 [39] and CoderAssist [19] are approaches that require manual effort. AutoGrader is a model-based technique to generate feedback by using constraint-based program synthesis. sk_p [35] 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 [9, 30], which broadly classified into techniques for particular error types [4, 8, 16, 21, 26, 38, 41, 44, 47] and general-purpose techniques [3, 24, 25, 28, 29, 45, 46]. In particular, our work is similar to [28] in that both techniques use reference programs but our goal is to provide feedback on student programs.

Our context-aware matching is also related to existing program embedding techniques [6, 15]. For example, Func2vec [6] is used to identify similar functions based on call relationships of functions. By contrast, our technique finds similar functions based on function contexts.

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.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair) and Samsung Research Fund & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-51. This research was partly supported by the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program(IITP-2021-2020-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation) and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2020R1C1C1014518, 2021R1A5A1021944).


