Type Error Feedback via Analytic Program Repair

Georgios Sakkas
Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
gsakkas@eng.ucsd.edu

Madeline Endres
Computer Science & Engineering
University of Michigan
Ann Arbor, MI, USA
endremad@umich.edu

Benjamin Cosman
Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
blcosman@eng.ucsd.edu

Westley Weimer
Computer Science & Engineering
University of Michigan
Ann Arbor, MI, USA
weimerw@umich.edu

Ranjit Jhala
Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
jhala@cs.ucsd.edu

Abstract

We introduce Analytic Program Repair, a data-driven strategy for providing feedback for type-errors via repairs for the erroneous program. Our strategy is based on insight that similar errors have similar repairs. Thus, we show how to use a training dataset of pairs of ill-typed programs and their fixed versions to: (1) *learn* a collection of candidate repair templates by abstracting and partitioning the edits made in the training set into a representative set of templates; (2) *predict* the appropriate template from a given error, by training multi-class classifiers on the repair templates used in the training set; (3) *synthesize* a concrete repair from the template by enumerating and ranking correct (*e.g.* well-typed) terms matching the predicted template. We have implemented our approach in Rite: a type error reporting tool for OCaml programs. We present an evaluation of the *accuracy* and *efficiency* of Rite on a corpus of 4,500 ill-typed OCaml programs drawn from two instances of an introductory programming course, and a user-study of the *quality* of the generated error messages that shows the locations and final repair quality to be better than the state-of-the-art tool in a statistically-significant manner.

CCS Concepts: • **Software and its engineering** → **General programming languages**; **Automatic programming**; • **Computing methodologies** → *Machine learning*; • **Theory of computation** → *Abstraction.*

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386005

Keywords: Type Error Feedback, Program Synthesis, Program Repair, Machine Learning

ACM Reference Format:
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 International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3386005

1 Introduction

Languages with Hindley-Milner style, unification-based inference offer the benefits of static typing with minimal annotation overhead. The catch, however, is that programmers must first ascend the steep learning curve associated with understanding the *error messages* produced by the compiler. While *experts* can, usually, readily decipher the errors, and view them as invaluable aids to program development and refactoring, *novices* are typically left quite befuddled and frustrated, without a clear idea of *what* the problem is [41]. Owing to the importance of the problem, several authors have proposed methods to help debug type errors, typically, by *slicing* down the program to the problematic locations [12, 31], by *enumerating* possible causes [6, 21], or by *ranking* the possible locations using MAX-SAT [30], Bayesian [43] or statistical analysis [37]. While valuable, these approaches at best help localize the problem but students are still left in the dark about how to *fix* their code.

Repairs as Feedback. Several recent papers have proposed an inspiring new line of attack on the feedback problem: using techniques from synthesis to provide feedback in the form of *repairs* that students can apply to improve their code. These repairs can be found by symbolically searching a space of candidate programs circumscribed by an expert-defined repair model [14, 38]. However, for type errors, the space of candidate repairs is massive. It is quite unclear whether a small set of repair models *exists* or even if it does, what it *looks like*. More importantly, to scale, it is essential

that we remove the requirement that an expert carefully curate some set of candidate repairs.

Alternately, we can generate repairs via the observation that *similar programs* have similar repairs, *i.e.* by calculating "diffs" from the student's solution to the "closest" *correct* program [11, 42]. However, this approach requires a corpus of similar programs, whose syntax trees or execution traces can be used to match each incorrect program with a "correct" version that is used to provide feedback. Programs with static type errors have no execution traces. More importantly, we desire a means to generate feedback for *new* programs that novices write, and hence cannot rely on matching against some (existing) correct program.

Analytic Program Repair. In this work, we present a novel error repair strategy called *Analytic Program Repair* that uses supervised learning instead of manually crafted repair models or matching against a corpus of correct code. Our strategy is based on the key insight that *similar errors* have similar repairs and realizes this insight by using a training dataset of pairs of ill-typed programs and their fixed versions to: (1) *learn* a collection of candidate repair templates by abstracting and partitioning the edits made in the training set into a representative set of templates; (2) *predict* the appropriate template from a given error, by training multi-class classifiers on the repair templates used in the training set; (3) *synthesize* a concrete repair from the template by enumerating and ranking correct (*e.g.* well-typed) terms matching the predicted template, thereby, generating a fix for a candidate program. Critically, we show how to perform the crucial abstraction from a particular *program* to an abstract *error* by representing programs via *bag-of-abstracted-terms* (BOAT) *i.e.* as numeric vectors of syntactic and semantic features [35]. This abstraction lets us train predictors over high-level code features, *i.e.* to learn correlations between features that cause errors and their corresponding repairs, allowing the analytic approach to generalize beyond matching against existing programs.

Rite. We have implemented our approach in Rite: a type error reporting tool for OCaml programs. We train (and evaluate) Rite on a set of over 4,500 ill-typed OCaml programs drawn from two years of an introductory programming course. Given a new ill-typed program, Rite generates a list of potential solutions ranked by likelihood and an *edit-distance* metric. We evaluate Rite in several ways. First, we measure its *accuracy*: we show that Rite correctly predicts the right repair template 69% of the time when considering the top three templates and surpasses 80% when we consider the top six. Second, we measure its *efficiency*: we show that Rite is able to synthesize a concrete repair within 20 seconds 70% of the time. Finally, we measure the *quality* of the generated messages via a user study with 29 participants and show that humans perceive both Rite's edit locations and final repair quality to be better than those produced by Seminal, a state-of-the-art OCaml repair tool [21] in a statistically-significant manner.

2 Overview

We begin with an overview of our approach to suggesting fixes for faulty programs by learning from the processes novice programmers follow to fix errors in their programs.

```
1  let rec mulByDigit i l =
2    match l with
3    | []     -> []
4    | hd::tl -> (hd * i) @ mulByDigit i tl
```

```
1  let rec mulByDigit i l =
2    match l with
3    | []     -> []
4    | hd::tl -> [hd * i] @ mulByDigit i tl
```

Figure 1. (top) An ill-typed OCaml program that should multiply each element of a list by an integer. (bottom) The fixed version by the student.

The Problem. Consider the program `mulByDigit` shown at the top of Figure 1, written by a student in an undergraduate Programming course. The program is meant to multiply all the numbers in a list with an integer digit. The student accidentally misuses the list append operator (`@`), applying it to a number and a list rather than two lists. Novice students who are still building a mental model of how the type checker works are often perplexed by the compiler's error message [26]. Hence a novice will often take a long time to arrive at a suitable fix, such as the one shown at the bottom of Figure 1, where `@` is used with a singleton list containing the multiplication of the head `hd` and `i`. Our goal is to use historical data of how programmers have fixed similar errors in their programs to automatically and rapidly guide novices to come up with candidate solutions like the one above.

Solution: Analytic Program Repair. One approach is to view the search for candidate repairs as a synthesis problem: synthesize a (small) set of edits to the program that yields a good (*e.g.* type-correct) one. The key challenge is to ensure that synthesis is *tractable* by restricting the repairs to an efficiently searchable space, and yet *precise* so the search does not miss the right fixes for an erroneous program. In this work, we present a novel strategy called *Analytic Program Repair* which enables tractable and precise search by decomposing the problem into three steps: First, *learn* a set of widely used *fix templates*. Second, *predict*, for each erroneous program, the correct fix template to apply. Third, *synthesize* candidate repairs from the predicted template. In the remainder of this section, we give a high-level overview of our approach by describing how to:

1. Represent fixes abstractly via *fix templates* (§ 2.1),

2. Acquire a *training set* of labeled ill-typed programs and fixes (§ 2.2),
3. Learn a small set of candidate fix templates by *partitioning* the training set (§ 2.3),
4. Predict the appropriate template to apply by training a *multi-class classifier* from the training set (§ 2.4), and
5. Synthesize fixes by enumerating and checking terms from the predicted templates to give the programmer localized feedback (§ 2.5).

2.1 Representing Fixes

Our notion of a fix is defined as a *replacement* of an existing expression with a new *candidate* expression at a specific program location. For example, the `mulByDigit` program is fixed by replacing `(hd * i)` with the expression `[hd * i]` on line 4. We focus on AST-level replacements as they are compact yet expressive enough to represent fixes.

Generic Abstract Syntax Trees. We represent the different possible candidate expressions via abstract fix templates called *Generic Abstract Syntax Trees* (GAST) which each correspond to many possible expressions. GASTs are obtained from concrete ASTs in two steps. First, we abstract concrete variable, function, and operator names. Next, we prune GASTs at a certain depth d to keep only the top-level changes of the fix. Pruned sub-trees are replaced with *holes*, which can represent *any* possible expression in our language.

Together, these steps ensure that GASTs only contain information about a fix's *structure* rather than the specific changes in variables and functions. For example, the fix `[hd * i]` in the `mulByDigit` example is represented by the GAST of the expression `[_ ⊕ _]`, where variables `hd` and `i` are abstracted into holes (*e.g.* by pruning the GAST at a depth $d = 2$) and `*` is represented by an abstract binary operator $\oplus$. Our approach is similar to that of Lerner *et al.* [21], where AST-level modifications are used, however, our proposed GASTs represent more abstract fix schemas.

2.2 Acquiring a Fix-Labeled Training Set

Previous work has used experts to create a set of ill-typed programs and their fixed versions [21, 22], or to manually create *fix templates* [16] that can yield *repair patches* [24, 25]. These approaches are hard to scale up to yield datasets suitable for machine learning. Also, they do not discover the *frequency* in practice of particular classes of novice mistakes and their fixes. In contrast, we show that such fix templates can be *learned* from a large, automatically constructed training set of ill-typed programs labeled with their repairs. Fixes in our dataset are represented as the ASTs of the expressions that students changed in the ill-typed program to transform it into the correct solution.

Interaction Traces. Following [37], we extract a labeled dataset of erroneous programs and their fixed versions from *interaction traces*. Usually students write several versions of their programs until they reach the correct solution for a programming assignment. An instrumented compiler is used to capture such sequences (or *traces*) of student programs. The first type-correct solution in this sequence of attempts is considered to be the fixed version of all the previous ones and thus a pair for each of them is added to the dataset. For each program pair, we then produce a *diff* of their abstract syntax trees (ASTs), and assign as the dataset's fix labels the *smallest* sub-tree that changed between the correct and ill-typed attempt of the program.

2.3 Learning Candidate Fix Templates

Each labeled program in our dataset contains a fix, which we abstract to a fix template. For example, for the `mulByDigit` program in Figure 1 we get the candidate fix `[hd * i]` and hence the fix template `[_ ⊕ _]`. However, a large dataset of fix-labeled programs, which may include many diverse solutions, can introduce a huge set of fix templates, which can be inappropriate for predicting the correct one to be used for the final program repair.

Therefore, the next step in our approach is to learn a set of fix templates that is *small enough* to automatically predict which template to apply to a given erroneous program, but nevertheless *covers* most of the fixes that arise in practice.

Partitioning the Fixes. We learn a suitable small set of fix templates by *partitioning* all the templates obtained from our dataset, and then selecting a single GAST to represent the fix templates from each fix template set. The partitioning serves two purposes. First, it identifies a small set of the most common fix templates which then enables the use of discrete classification algorithms to predict which template to apply to a new program. Second, it allows for the principled removal of outliers that arise because student submissions often contain non-standard or idiosyncratic solutions that we do not wish to use for suggesting fixes.

Unlike previous repair approaches that have used clustering to group together similar programs (e.g., [11, 42]), we partition our set of fix templates into their *equivalence classes* based on a fix similarity relation.

2.4 Predicting Templates via Multi-classification

Next, we train models that can correctly predict error locations and fix templates for a given ill-typed program. We use these models to generate candidate expressions as possible program fixes. To reduce the complexity of predicting the correct fix templates and error locations, we separate these problems and encode them into two distinct *supervised classification* problems.

Supervised Multi-Class Classification. We propose using a *supervised multi-class classification* problem for predicting fix templates. A *supervised* learning problem is one where, given a labeled training set, the task is to learn a function

that accurately maps the inputs to output labels and generalizes to future inputs. In a *classification* problem, the function we are trying to learn maps inputs to a discrete set of two or more output labels, called *classes*. Therefore, we encode the task of learning a function that will map subexpressions of ill-typed programs to a small set of candidate fix templates as a *multi-class* classification (MCC) problem.

Feature Extraction. The machine learning models that we will train to solve our MCC problem expect datasets of labeled *fixed-length vectors* as inputs. Therefore, we define a transformation of fix-labeled programs to fixed-length vectors. Similarly to Seidel *et al.* [37], we define a set of feature extraction functions $f_1, \ldots, f_n$, that map program subexpressions to a numeric value (or just $\{0, 1\}$ to encode a boolean property). Given a set of feature extraction functions, we can represent a single program's AST as a set of fixed-length vectors by decomposing the AST e into a set of its constituent subexpressions $\{e_1, \ldots, e_m\}$ and then representing each e_i with the n-dimensional vector $[f_1(e_i), \ldots, f_n(e_i)]$. This method is known as a *bag-of-abstracted-terms* (BOAT) representation in previous work [37].

Predicting Templates via MCC. Our fix-labeled dataset can be updated so the labels represent the corresponding template that fixes each location, drawn from the minimal set of fix templates that were acquired through partitioning. We then train a *Deep Neural Network (DNN)* classifier on the updated template-labeled data set.

Neural networks have the advantage of associating each class with a *confidence score* that can be interpreted as the model's probability of each class being correct for a given input according to the model's estimated distribution. Therefore, confidence scores can be used to rank fix template predictions for new programs and use them in descending order when synthesizing repairs. Exploiting recent advances in machine learning, we use deep and dense architectures [34] for more accurate fix template predictions.

Error Localization. We view the problem of finding error locations in a new program as a *binary* classification problem. In contrast with the template prediction problem, we want to learn a function that maps a program's subexpressions to a binary output representing the presence of an error or not. Therefore, this problem is equivalent to MCC with only two classes and thus, we use similar deep architectures of neural networks. For each expression in a given program, the learned model outputs a confidence score representing how likely it is an error location that needs to be fixed. We exploit those scores to synthesize candidate expressions for each location in descending order of confidence.

2.5 Synthesizing Feedback from Templates

Next, we use classic program *synthesis* techniques to synthesize candidate expressions that will be used to provide feedback to users. Additionally, synthesis is guided by predicted fix templates and a set of possible error locations, and returns a ranked list of *minimal* repairs to users as feedback.

Program Synthesis. Given a set of locations and candidate templates for those locations, we are trying to solve a problem of *program synthesis*. For each program location, we search over all possible expressions in the language's grammar for a small set of candidate expressions that match the fix template and make the program type-check. Expressions from the ill-typed program are also used during synthesis to prune the search space of candidate expressions.

Synthesis for Multiple Locations. It is often the case that more than one location needs to be fixed. Therefore, we do not only consider the ordered set of single error locations for synthesis, but rather its power set. For simplicity, we consider fixing different program locations as independent; the probability we assign that a set of locations needs to be fixed is thus the product of their individual confidence scores. This is unlike recent approaches to multi-hunk program repair [33] where modifications depend on each other.

Ranking Fixes. Finally, we rank each solution by two metrics, the *tree-edit distance* and the *string-edit* distance. Previous work [11, 21, 42] has used such metrics to consider minimal changes, *i.e.* changes that are as close as possible to the original programs, so novice programmers are presented with more coherent feedback.

```
1    let rec mulByDigit i l =
2      match l with
3      | []     -> []
4      | hd::tl -> [v1 * v2] @ mulByDigit i tl
```

Figure 2. A candidate repair for the `mulByDigit` program.

Example. We see in Figure 2 a minimal repair that our method could return (`[v1 * v2]` in line 4) using the template discussed in § 2.3 to synthesize it. While this solution is not the highest-ranked that our implementation returns (which would be identical to the human solution), it demonstrates relevant aspects of the synthesizer. In particular, this solution has some abstracted variables, v_1 and v_2. Our algorithm suggests to the user that they can replace the two variables with two distinct variables and insert the whole expression into a list, in order to obtain the correct program. We hypothesize that such solutions produced by our algorithm can provide valuable feedback to novices, and we investigate that claim empirically in § 6.3.

3 Learning Fix Templates

We start by introducing our approach for extracting useful *fix templates* from a training dataset comprised of paired erroneous and fixed programs. We express those templates

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.e \mid e\ \bar{e} \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& \mid & n \mid b \mid e + e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \langle e, e \rangle \mid \texttt{match}\ e\ \texttt{with}\ \langle x, x \rangle \rightarrow e \\
& \mid & [] \mid e :: e \mid \texttt{match}\ e\ \texttt{with} \begin{cases} [] \rightarrow e \\ x :: x \rightarrow e \end{cases} \\
n & ::= & 0, 1, -1, \ldots \\
b & ::= & \texttt{true} \mid \texttt{false} \\
t & ::= & \alpha \mid \texttt{bool} \mid \texttt{int} \mid t \rightarrow t \mid t \times t \mid [t]
\end{array}
$$

Figure 3. Syntax of λ^{ML}

$$
\begin{array}{rcl}
e & ::= & \_ \mid \hat{x} \mid \lambda \hat{x}.e \mid \hat{x}\ \bar{e} \mid \texttt{let}\ \hat{x} = e\ \texttt{in}\ e \\
& \mid & \hat{n} \mid e \oplus e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \langle e, e \rangle \mid \texttt{match}\ e\ \texttt{with}\ \langle \hat{x}, \hat{x} \rangle \rightarrow e \\
& \mid & [] \mid e :: e \mid \texttt{match}\ e\ \texttt{with} \begin{cases} [] \rightarrow e \\ \hat{x} :: \hat{x} \rightarrow e \end{cases}
\end{array}
$$

Figure 4. Syntax of λ^{RTL}

in terms of a language that allows us to succinctly represent fixes in a way that captures the essential structure of various fix patterns that novices use in practice. However, extracting a single fix template for *each* fix in the program pair dataset yields too many templates to perform accurate predictions. Hence, we define a *similarity* relation between templates, which we use to *partition* the extracted templates into a small but representative set, that will make it easier to train precise models to predict fixes.

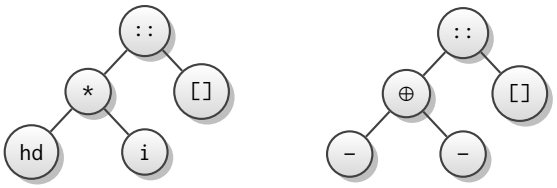
3.1 Representing User Fixes

Repair Template Language. Figure 4 describes our Repair Template Language, λ^{RTL}, which is a lambda calculus with integers, booleans, pairs, and lists, that extends our core ML language λ^{ML} (Figure 3) with syntactic abstraction forms:

1. *Abstract variable* names $\hat{x}$ are used to denote variable occurrences for functions, variables and binders, *i.e.* $\hat{x}$ denotes an unknown variable name in λ^{RTL};
2. *Abstract literal* values $\hat{n}$ can represent *any* integer, float, boolean, character, or string;
3. *Abstract operators* $\oplus$ similarly denote unknown unary or binary operators;
4. *Wildcard* expressions _ are used to represent *any* expression in λ^{RTL}, *i.e.* a program *hole*.

Recall from § 2.1 that we define fixes as replacements of expressions with new candidate expressions at specific program locations. Therefore, we use candidate expressions over λ^{RTL} to represent fix templates.

Generalizing ASTs. A *Generic Abstract Syntax Tree* (GAST) is a term from λ^{RTL} that represents many possible expressions from λ^{ML}. GASTs are abstracted from standard ASTs over the core language λ^{ML} using the **abstract** function that



(a) Fix AST **(b)** Template GAST

Figure 5. (left) The fix from example Figure 1 and (right) a possible template for that fix.

takes as input an expression e^{ML} over λ^{ML} and a depth d and returns an expression e^{RTL} over λ^{RTL}, *i.e.* a GAST with all variables, literals and operators of e^{ML} abstracted and all subexpressions starting at depth greater than d pruned and replaced with holes _.

Example. Recall our example program `mulByDigit` in Figure 1. The expression `[hd * i]` replaces `(hd * i)` in line 4, and hence, is the user's *fix*, whose AST is given in Figure 5a. The output of **abstract**, given this AST and a depth $d = 2$ as input, would be the GAST in Figure 5b, where the operator `*` has been replaced with an abstract operator $\oplus$, and the sub-terms `hd` and `i` at depth 2 have been abstracted to wildcard expressions _. Hence, the λ^{RTL} term `[_ ⊕ _]` represents a potential fix template for `mulByDigit`.

3.2 Extracting Fix Templates from a Dataset

Our approach fully automates the extraction of fixes by harvesting a set of fix templates from a training set of program pairs. Given a program pair (p_{err}, p_{fix}) from the dataset, we extract a unique fix for each location in p_{err} that changed in p_{fix}. We do so with an expression-level **diff** [20] function. Recall that our fixes are replacements of expressions, so we abstract these extracted changes as our fix templates.

Contextual Repairs. Following Felleisen *et al.* [8], let $\mathbf{C}$ be the *context* in which an expression e appears in a program p, *i.e.* the program p with e replaced by a hole _. We write that $p = \mathbf{C}[e]$, meaning that if we fill the hole with the original expression e we obtain the original program p. In this fashion, **diff** finds a *minimal* (in number of nodes) expression replacement e_{fix} for an expression e_{err} in p_{err}, such that $p_{err} = \mathbf{C}_{p_{err}}[e_{err}]$ and $\mathbf{C}_{p_{err}}[e_{fix}] = p_{fix}$. There may be several such expressions, and **diff** returns all such changes.

Examples. If $f\ x$ is rewritten to $g\ x$, the context is $\mathbf{C} = \_\ x$ and the fix is g, since $\mathbf{C}[g] = g\ x$. If $f\ x$ is rewritten to $(f\ x) + 1$, the context is $\mathbf{C} = \_$, and the fix is the whole expression $(f\ x) + 1$, thus $\mathbf{C}[(f\ x) + 1] = (f\ x) + 1$. (Even though $f\ x$ appears in both the original and fixed programs, we consider the application expression $f\ x$ — but not f or x — to be replaced with the + operator.)

3.3 Partitioning the Templates

Programs over λ^{ML} force similar fixes, such as changes to variable names, to have identical GASTs. Our next step is to define a notion of program fix *similarity*. Our definition supports the formation of a small but widely-applicable set of fix templates. This small set is used to train a repair predictor.

GAST Similarity. Two GASTs are *similar* when the root nodes are the same and their child subtrees (if any) can be ordered such that they are pairwise similar. For example, $x+3$ and $7-y$ yield the *similar* GASTs $\hat{x} \oplus \hat{n}$ and $\hat{n} \oplus \hat{x}$, where the root nodes are both abstract binary operators, one child is an abstract literal, and one child is an abstract variable.

Partitioning. GAST similarity defines a relation which is reflexive, symmetric, and transitive and thus an *equivalence* relation. We can now define *partitioning* as the computation of all possible *equivalence classes* of our extracted fix templates *w.r.t.* GAST similarity. Each class can consist of several member-expressions and any one of them can be viewed as the class *representative*. Each representative can then be used as a fix template to produce repairs for ill-typed programs.

For example, $\hat{x} \oplus \hat{n}$ and $\hat{n} \oplus \hat{x}$ are in the same class and either one can be used as the representative. The repair algorithm in section 5 will essentially consider both when fixing an erroneous program with this template.

Finally, our partitioning algorithm returns the top N equivalence classes based on their member-expressions frequency in the dataset. N is a parameter of the algorithm and is chosen to be as small as possible while the top N classes represent a large enough portion of the dataset.

4 Predicting Fix Templates

Given a candidate set of templates, our next task is to *train* a model that, when given an (erroneous) program, can predict which template to use for each location in that program. We do so by defining a function **predict** which takes as input (1) a feature extraction function Features, (2) a dataset DataSet of program pairs $(p_{err},\ p_{fix})$, and (3) a list of fix templates T. It returns as output a *fix-template-predictor* which, given an erroneous program, returns the locations of likely fixes, and the templates to be applied at those locations.

We build **predict** using three helper functions that carry out each of the high-level steps. First, the **extract** function extracts *features* and *labels* from the program pair dataset. Next, these feature vectors are grouped and fed into **train** which produces two models, LModel and TModel, that are respectively used for error localization and predicting fix templates. Finally, **rank** takes the features for a new (erroneous) program and queries the trained models to return the likely fix locations and corresponding fix templates.

Next, we describe the key data-types in Figure 6, our implementations of the three key steps, and how they are combined to yield the **predict** algorithm.

Confidences, Data and Labels. As shown in Figure 6, we define EMap a as a mapping from expressions e to values of type a, and TMap a as a mapping from templates T to such values. For example, TMap C is a mapping from templates T to their confidence scores C. Data represents feature vectors used to train our predictive models, while Label $\mathcal{B}$ are the dataset labels for training and Label C are the output confidence scores. Finally, Pair is a program pair $(p_{err},\ p_{fix})$.

Features and Predictors. We define Features as a function that generates the feature vectors Data for each subexpression of an input program e. Those feature vectors are given in the form of a map EMap Data, which maps all subexpressions of the input program e to its feature vector Data.

Predictors are learned fix-template-predictors returned from our algorithm that are used to generate confidence score mappings for input programs e. Specifically, they return a map EMap (Label C) that associates each subexpression of the input program e with a confidence score Label C.

Architecture. First, the **extract** function takes as input the feature extraction functions Features, a list of templates [T] and a single program pair Pair and generates a map EMap (Data $\times$ Label $\mathcal{B}$) of feature vectors and boolean labels for all subexpressions of the erroneous input program from Pair. All feature vectors Data and labels Label $\mathcal{B}$ are then accumulated into one list, which is given as input to **train** and are used for training the two models LModel and TModel that are respectively used for predicting error locations and fix templates. Next, the two trained models LModel and TModel, along with Data from a new and previously unseen program, can be fed into **rank**. This produces a Predictor, which can be used to map subexpressions of the new program to possible error locations and fix templates.

4.1 Feature and Label Extraction

The machine learning algorithms that we use for predicting fix templates and error locations expect fixed-length *feature vectors* Data as their input. However, we want to repair variable-sized programs over λ^{ML}. We thus use the **extract** function to convert programs to feature vectors.

Following Seidel *et al.* [37], we choose to model a program as a *set* of feature vectors, where each element corresponds to a subexpression in the program. Thus, given an erroneous program p_{err} we first split it into its constituent subexpressions and then transform each subexpression into a single feature vector, *i.e.* Features p_{err} :: EMap Data. We only consider expressions inside a minimal type-error *slice*. We show here the five major feature categories used.

Local syntactic features. These features describe the syntactic category of each expression e. In other words, for each production rule of e in Figure 3 we introduce a feature

C	$\doteq$	$\{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$
$\mathcal{B}$	$\doteq$	$\{b \in \mathbb{R} \mid b = 0 \vee b = 1\}$
T	$\doteq$	e^{RTL}
EMap a	$\doteq$	$e \to a$
TMap a	$\doteq$	T $\to a$
Data	$\doteq$	$[C]$
Label a	$\doteq$	$a \times$ TMap a
Pair	$\doteq$	$e \times e$
DataSet	$\doteq$	[Pair]
Features	$\doteq$	$e \to$ EMap Data
Predictor	$\doteq$	$e \to$ EMap (Label C)
abstract	:	$e \to$ T
diff	:	Pair $\to$ [e]
extract	:	Features $\to$ [T] $\to$ Pair $\to$ EMap (Data $\times$ Label $\mathcal{B}$)
train	:	[Data $\times$ Label $\mathcal{B}$] $\to$ LModel $\times$ TModel
rank	:	LModel $\to$ TModel $\to$ Data $\to$ Label C
predict	:	Features $\to$ [T] $\to$ DataSet $\to$ Predictor

Figure 6. A high-level API for converting program pairs to feature vectors and template labels.

that is enabled (set to 1) if the expression was built with that production, and disabled (set to 0) otherwise.

Contextual syntactic features. The *context* in which an expression occurs can be critical for correctly predicting error sources and fix templates. Therefore, we include contextual features, which are similar to the local syntactic features but describe the parent and children of an expression. For example, the Is-[]-C1 feature would describe whether an expression's *first child* is []. This is similar to the *n-grams* used in linguistic models [9, 15].

Expression size. We also include a feature representing the *size* of each expression, *i.e.* how many subexpressions does it contain? This allows the model to learn that, *e.g.*, expressions closer to the leaves are more likely to be fixed than expressions closer to the root.

Typing features. The programs we are trying to repair are *untypeable*, but a *partial typing* derivation from the type checker could still provide useful information to the model. Therefore, we include *typing* features in our representation. Due to the parametric type constructors $\cdot \to \cdot$, $\cdot \times \cdot$, and $[\cdot]$, there is an *infinite* set of possible types — but we must have a *finite* set of features. We add features for each abstract type constructor that describes whether a given type uses that constructor. For example, the type `int` $\to$ `int` $\to$ `bool` would enable the $\cdot \to \cdot$, `int`, and `bool` features.

We add these features for parent and child expressions to summarize the context, but also for the current expression, as the type of an expression is not always clear syntactically.

Type error slice. We wish to distinguish changes that could fix the error from changes that *cannot possibly* fix the error. Thus, we compute a minimal type-error *slice* (*e.g.* [12, 40]) for the program (*i.e.* the set of expressions that contribute to the error) and if the program contains multiple type-errors, we compute a minimal slice for each error. We then have a post-processing step that discards all expressions that are not included in those slices.

Labels. Recall that we use two predictive models, LModel for error localization and TModel for predicting fix templates. We thus require two sets of *labels* associated with each feature vector, given by Label $\mathcal{B}$. LModel is trained using the set [Data $\times \mathcal{B}$], while TModel using the set [Data $\times$ TMap $\mathcal{B}$].

LModel's labels of type $\mathcal{B}$ are set to "true" for each subexpression of a program p_{err} that changed in p_{fix}. A label TMap $\mathcal{B}$, for a subexpression of p_{err}, maps to the repair template T that was used to fix it. TMap $\mathcal{B}$ associates all subexpressions with a fixed number of templates [T] given as input to **extract**. Therefore, for the purpose of template prediction, TMap $\mathcal{B}$ can be viewed as a fixed-length boolean vector that represents the fix templates used to repair each subexpression. This vector has at most one slot set to "true", representing the template used to fix p_{err}. These labels are extracted using **diff** and **abstract**, similarly to the way that templates were extracted in § 3.2.

4.2 Training Predictive Models

Our goal with the **train** function is to train two separate *classifiers* given a training set [Data $\times$ Label $\mathcal{B}$] of labeled examples. LModel predicts error locations and TModel predicts fix templates for a new input program p_{err}. Critically, we require that the error localization classifier output a *confidence score* C that represents the probability that a subexpression is the error that needs to be fixed. We also require that the fix template classifier output a confidence score C for each fix template that measures how sure the classifier is that the template can be used to repair the associated location of the input program p_{err}.

We consider a standard learning algorithm to generate our models: *neural networks*. A thorough introduction to neural networks is beyond the scope of this work [13, 28].

Neural Networks. The model that we use is a type of neural network called a *multi-layer perceptron*. A multi-layer perceptron can be represented as a directed acyclic graph whose nodes are arranged in layers that are fully connected by weighted edges. The first layer corresponds to the input features, and the final to the output. The output of an internal node is the sum of the weighted outputs of the previous layer passed to a non-linear function, called the activation function. The number of layers, the number of nodes per layer, and the connections between layers constitute the *architecture* of a neural network. In this work, we use relatively *deep neural networks* (DNN). We can train a DNN LModel as a binary

Algorithm 1 Predicting Templates Algorithm

Input: Feature Extraction Functions F, Fix Templates Ts, Program Pair Dataset D
Output: Predictor Pr

1: **procedure** PREDICT(F, Ts, D)
2: $D_{ML} \leftarrow \emptyset$
3: **for all** $p_{err} \times p_{fix} \in D$ **do**
4: $d \leftarrow$ EXTRACT(F, Ts, $p_{err} \times p_{fix}$)
5: $D_{ML} \leftarrow D_{ML} \cup$ INSLICE(p_{err}, d)
6: $Models \leftarrow$ TRAIN(D_{ML})
7: $Data \leftarrow \lambda p.$ INSLICE(p, EXTRACT(F, Ts, $p \times p$))
8: $Pr \leftarrow \lambda p.$ MAP($\lambda \tilde{p}.$ RANK($Models$, $\tilde{p}[0]$), $Data(p)$)
9: **return** Pr

classifier, which will predict whether a location in a program p_{err} has to be fixed or not.

Multi-class DNNs. While the above model is enough for error localization, in the case of template prediction we have to select from more than two *classes*. We again use a DNN for our template prediction TModel, but we adjust the output layer to have N nodes for the N chosen template-classes. For multi-class classification problems solved with neural networks, usually a *softmax* function is used at output layer [5, 10]. Softmax assigns probabilities to each class that must add up to 1. This additional constraint speeds up training.

4.3 Predicting Fix Templates

Our ultimate goal is to be able to pinpoint what parts of an erroneous program should be repaired and what fix templates should be used for that purpose. Therefore, the **predict** function uses **rank** to predict all subexpressions' confidence scores C to be an error location and confidence scores TMap C for each fix template. We show here how all the functions in our high-level API in Figure 6 are combined to produce a final list of confidence scores for a new program p. Algorithm 1 presents our high-level **predict** algorithm.

The Prediction Algorithm. Our algorithm first extracts the machine-learning-amenable dataset D_{ML} from the program pairs dataset D. For each program pair in D, EXTRACT returns a mapping from the erroneous program's subexpressions to features and labels. Then, INSLICE keeps only the expressions in the the type-error slice and evaluates to a list of the respective feature and label vectors, which is added to the D_{ML} dataset. This dataset is used by the TRAIN function to generate our predictive *Models*, *i.e.* LModel and TModel.

At this point we want to generate a Predictor for a new unknown program p. We perform feature extraction for p with EXTRACT, and use INSLICE to restrict to expressions in p's type-error slice. The result is given by $Data(p)$.

RANK is then applied to all subexpressions produced by $Data(p)$ with MAP, which will create a mapping of the type EMap (Label C) associating expressions with confidence scores. We apply RANK to each feature vector that corresponds to an expression in the type-error slice of p. These vectors are the first elements of $\tilde{p} \in Data(p)$, which are of type Data $\times$ Label $\mathcal{B}$. Finally, Predictor Pr is returned, which is used by our synthesis algorithm in section 5 to correlate subexpressions in p with their confidence scores.

4.4 Discussion

An alternative to the two separate predictive models, LModel and TModel, would be to have one *joint* model to predict both error locations and fix templates. One could simply add an "empty" fix template to the set of the N extracted templates. Then, a multi-class DNN could be trained on the dataset, using $N + 1$ classes instead. When the "empty" fix template is predicted, it denotes no error at that location, while the rest of the classes denote an error along with the fix template to be used. While the approach of one joint model is quite intuitive, we found in our early experiments that it does not produce as accurate predictions as the two separate models.

Learning representations is a remarkable strength of DNNs, so manually extracting features is usually discouraged. Recently, there has been some work in learning program representations for use in predictive models [2, 4]. However, we found that the BOAT features are essential for high accuracy (see subsection 6.1) given the relatively small size of our dataset, similarly to previous work [37]. In future work, however, it would be interesting to learn features automatically and avoid the step of manually extracting them.

5 Template-Guided Repair Synthesis

We use program synthesis to fully repair a program using predicted fix templates and locations from our machine learning models. We present in § 5.1 a synthesis algorithm for producing *local repairs* for a given program location. In § 5.2, we show how we use local repairs to repair programs that may have *multiple* error locations.

5.1 Local Synthesis from Templates

Enumerative Program Synthesis. We utilize classic *enumerative* program synthesis that is guided by a fix template. Enumerative synthesis searches all possible expressions over a language until a high-level specification is reached. In our case, we initially synthesize independent *local repairs* for a program that already captures the user's intent. Therefore, the required specification is that the repaired program is type-safe. However, if the users provide type signatures for their programs, they can be used as a stricter specification.

Given a location l, a template t and a maximum depth d, Algorithm 2 searches over all possible expressions over λ^{ML} that will satisfy those goals by generating a local repair that fills t's GAST with concrete variables, literals, functions *etc.*

Algorithm 2 Local Repair Algorithm

Input: Language Grammar λ^{ML}, Program P, Template T, Repair Location L, Max Repair Depth D
Output: Local Repairs R

```
1:  procedure Repair(λ^ML, P, T, L, D)
2:      R ← ∅
3:      for all d ∈ [1 ... D] do
4:          α̃ ← NonTerminalsAt(T, d)
5:          for all α ∈ RankNonTerminals(α̃, P, L) do
6:              if IsHole(α) then
7:                  Q ← GrammarRules(λ^ML)
8:                  β̃ ← {β | (α, β) ∈ Q}
9:                  for all β ∈ RankRules(β̃, T) do
10:                     T̂ ← ApplyRule(T, (α, β))
11:                     R ← R ∪ {T̂}
12:             else
13:                 for all σ ∈ GetTerminals(P, L, λ^ML) do
14:                     T̂ ← ReplaceNode(T, α, σ)
15:                     R ← R ∪ {T̂}
16:     return R
```

Our technique can also reuse subexpressions e at location l for t's concretization to further optimize the search.

Template-Guided Local Repair. Using the Repair method (Algorithm 2), we produce local repairs R for a given location L of an erroneous program P. Repair fills in a template T based on the context-free grammar λ^{ML}. It traverses the GAST of template T from root node downward, producing candidate local repairs of maximum depth D.

When a hole $\alpha \in T$ is found, the algorithm expands T's GAST one more level using λ^{ML}'s production rules Q. The production rules are considered in a ranked order based on the subexpressions that already appear in the rest of the template T and program location L. Each rule is then applied to template T, returning an *instantiated* template $\hat{T}$, which is inserted into the list of candidate local repairs R.

If node α is not a hole, terminals from the subexpressions at location L, the program P in general and the grammar λ^{ML} are used to concretize that node, depending on the λ^{RTL} terminal node α. For each of these template T modifications, we insert an instantiated template $\hat{T}$ into R.

5.2 Ranking Error Locations

Error Location Confidence. Recall from section 4 that for each subexpression in a program's type-error slice, LModel generates a confidence score C for it being the error location, and TModel generates scores for the fix templates.

Our synthesis algorithm ranks all program locations based on their confidence scores C. For all locations in descending confidence score order, a fix template is used to produce a local repair using Algorithm 2. Fix templates are considered in descending order of confidence. Then expressions from the returned list of local repairs R replace the expression at the given program location. The procedure tries the remaining repairs, templates, and locations until a type-correct program is found.

Following [21], we allow our final local repairs to have program holes _ or abstracted variable $\hat{x}$ in them. However, Algorithm 2 will prioritize the synthesis of complete solutions. Abstract λ^{RTL} terms can have any type when type-checking concrete solutions, similarly to OCaml's `raise Exn`.

Multiple Error Locations. In practice, frequently more than one program location needs to be repaired. We thus extend the above approach to fix programs with multiple errors. Let the confidence scores C for all locations L in the type error slice from our error localization model LModel be $(l_1, c_1), \ldots, (l_k, c_k)$, where l_i is a program location and c_i its error confidence score. We assume for simplicity that the probabilities c_i are independent. Thus the probability that *all* the locations $\{l_i \ldots l_j\}$ need to be fixed is the product $c_i \cdots c_j$. Therefore, instead of ranking and trying to find fixes for single locations l, we use *sets* of locations ($\{l_i\}$, $\{l_i, l_j\}$, $\{l_i, l_j, l_k\}$, *etc.*), ranked by the products of their confidence scores. For a given set, we use Algorithm 2 independently for each location in the set and apply all possible combinations of local repairs, looking again for a type-correct solution.

6 Evaluation

We have implemented analytic program repair in Rite: a system for repairing type errors for a purely functional subset of OCaml. Next, we describe our implementation and an evaluation that addresses three questions:

- **RQ1**: How *accurate* are Rite's predicted repairs? (§ 6.1)
- **RQ2**: How *efficiently* can Rite synthesize fixes? (§ 6.2)
- **RQ3**: How *useful* are Rite's error messages? (§ 6.3)
- **RQ4**: How *precise* are Rite's template fixes? (§ 6.4)

Training Dataset. For our evaluation, we use an OCaml dataset gathered from an undergraduate Programming Languages university course, previously used in related work [35, 37]. It consists of erroneous programs and their subsequent fixes and is divided in two parts; the Spring 2014 class (SP14) and the Fall 2015 class (FA15). The homework required students to write 23 distinct programs that demonstrate a range of functional programming idioms, *e.g.* higher-order functions and (polymorphic) algebraic data types.

Feature Extraction. Rite represents programs with BOAT vectors of 449 features from each expression in a program: 45 local syntactic, 315 contextual, 88 typing features, and 1 expression size feature. For contextual features, for each expression we extract the local syntactic features of its first 4 (left-to-right) children. In addition, we extract those features for its ancestors, starting from its parent and going up to two more parent nodes. For typing features, we support `int`s,

`floats`, `chars`, `strings`, and the user-defined `expr`. These features are extracted for each expression and its context.

Dataset Cleaning. We extract fixes as expressions replacements over a program pair using **diff**. A disadvantage of using **diff**s with this dataset is that some students may have made many, potentially unrelated, changes between compilations; at some point the "fix" becomes a "rewrite". These rewrites can lead to meaningless fix templates and error locations. We discard such outliers when the fraction of subexpressions that have changed in a program is more than one standard deviation above the mean, establishing a **diff** threshold of 40%. We also discard programs that have changes in 5 or more locations, noting that even state-of-the-art multi-location repair techniques cannot reproduce such "fixes" [33]. The discarded changes account for roughly 32% of each dataset, leaving 2,475 program pairs for SP14 and 2,177 pairs for FA15. Throughout, we use SP14 as a training set and FA15 as a test set.

DNN based Classifier. RITE's template prediction uses a multi-layer neural network DNN based classifier with three fully-connected hidden layers of 512 neurons. The neurons use rectified linear units (ReLU) as their activation function [27]. The DNN was trained using *early stopping* [13]: training is stopped when the accuracy on a distinct small part of the training set is not improved after a certain amount of epochs (5 epochs, in our implementation). We set the maximum number of epochs to 200. We used the ADAM optimizer [17], a variant of stochastic gradient descent that converges faster.

6.1 RQ1: Accuracy

Most developers will consider around five or six suggestions before falling back to manual debugging [18, 29]. Therefore, we consider RITE's accuracy up to the *top six* fix template predictions, *i.e.* we check if any of the top-N predicted templates actually correspond to the users's edit. These predicted templates are not shown to the user; they are only used to guide the synthesis of concrete repairs which are then presented to the user.

Baselines. We compare RITE's DNN-based predictor against two baseline classifiers: a RANDOM classifier that returns templates chosen uniformly at random from the 50 templates learned from the SP14 training dataset, and a POPULAR classifier that returns the most popular templates in the training set in decreasing order. We also compare to a *decision tree* (DTREE) and an SVM classifier trained on the SP14 data, since these are two of the most common learning algorithms [13].

Results: Accuracy of Prediction. Figure 7 shows the accuracy results of our template prediction experiments. The y-axis describes the fraction of *erroneous* sub-terms (locations) for which the actual repair was one of the top-K predicted repairs. The naive baseline of selecting templates at random

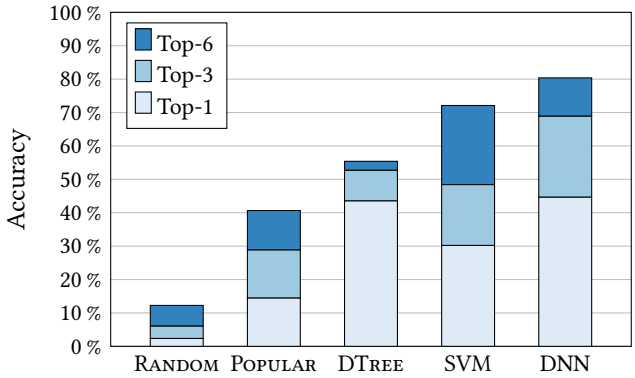


Figure 7. Results of our template prediction classifiers using the *50 most popular* templates. We present the results up to the top 6 predictions, since our synthesis algorithm considers that many templates before falling to a different location.

achieves 2% Top-1 accuracy (12% Top-6), while the POPULAR classifier achieves a Top-1 accuracy of 14% (41% Top-6). Our DNN classifier significantly outperforms these naive classifiers, ranging from 45% Top-1 accuracy to 80% Top-6 accuracy. In fact, even with only DNN's first prediction one outperforms top 6 predictions of both RANDOM and POPULAR. The RANDOM classifier's low performance is as expected. The POPULAR classifier performs better: some homework assignments were shared between SP14 and FA15 quarters and, while different groups of students solved these problems for each quarter, the novice mistakes that they made seem to have a pattern. Thus, the most *popular "fixes"* (and therefore the relevant templates) from SP14 were also popular in FA15.

We also observe that DTREE achieves a Top-1 accuracy close to that of DNN's (*i.e.* 44% vs. 45%) but fails to improve with more predictions (*i.e.* with Top-6, 55% vs. 80%). On the other hand, the SVM does poorly on the Top-1 accuracy (*i.e.* 30% vs. 45%) but does significantly better with more predictions (*i.e.* with Top-6, 72% vs. 80%). Therefore, we observe that more sophisticated learning algorithms can actually learn patterns from a corpus of fixed programs, with DNN classifiers achieving the best performance in each category.

Results: Template "Confusion". The *confusion matrix* of the each location's top prediction shows which templates our models mix up. Figure 8 shows this matrix for the top 30 templates acquired from the SP14 training set and were tested on the FA15 dataset. Note that most templates are predicted correctly and only a few of them are often mis-predicted for another template. For example, we see that programs that require template 20 (`let` $\hat{z}$ `=` `match` $\hat{t}$ `with` ($\hat{x}$, $\hat{y}$) $\rightarrow$ $\hat{a}$ `in _`) to be fixed, almost always are mis-predicted with template 11 (`let` ($\hat{x}$, $\hat{y}$) `=` $\hat{t}$ `in` (_, _)). We observe that these templates are still very similar, with both of them having a top-level `let` that manipulates tuples $\hat{t}$.

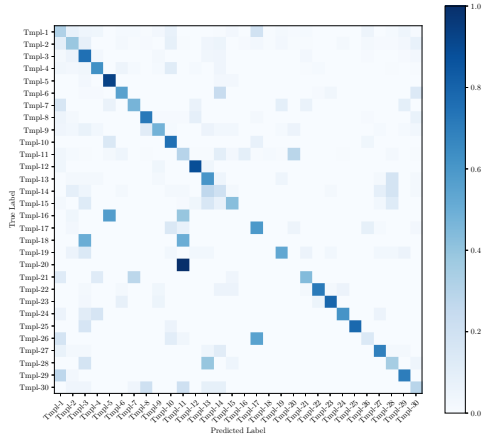


Figure 8. The confusion matrix of the *top 30* templates. Bolder parts of the heatmap show templates that are often mis-predicted with another template. The bolder the diagonal is, the more accurate predictions we make.

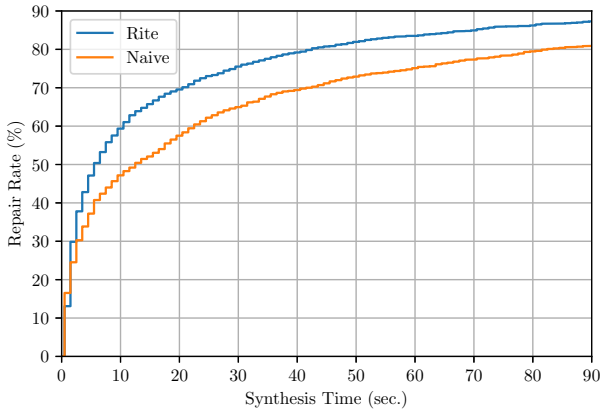


Figure 9. The proportion of the test set that can be repaired within a given time.

RITE learns correlations between program features and repair templates, yielding almost *2x higher* accuracy than the naive baselines and 8% more than the other sophisticated learning algorithms. By abstracting programs into features, RITE is able to *generalize* across years and different kinds of programs.

RITE can generate type-correct repairs for the vast majority of ill-typed programs in under 20 seconds.

6.2 RQ2: Efficiency

Next we evaluate RITE's efficiency by measuring how many programs it is able to generate a (well-typed) repair for. We limit the synthesizer to 90 seconds. (In general the procedure is undecidable, and we conjecture that a longer timeout will diminish the practical usability for novices.) Recall that the repair synthesis algorithm is guided by the repair template predictions. We evaluate the efficiency of RITE by comparing it against a baseline NAIVE implementation that, given the predicted fix location, attempts to synthesize a repair from the trivial "hole" template.

Figure 9 shows the cumulative distribution function of RITE's and NAIVE's repair rates over their synthesis time. We observe that using the predicted templates for synthesis allows RITE to generate type-correct repairs for almost 70% of the programs in under 20 seconds, which is nearly 12 points higher than the NAIVE baseline. We also observe that RITE successfully repairs around 10% more programs than NAIVE for times greater than 20 seconds. While the NAIVE approach is still able to synthesize well-typed repairs relatively quickly, we will see that these repairs are of much lower quality than those generated from the predicted templates (§ 6.4).

6.3 RQ3: Usefulness

The primary outcome is whether the repair-based error messages generated by RITE were actually useful to novices. To assess the quality of RITE's repairs, we conducted an online human study with 29 participants. Each participant was asked to evaluate the quality of the program fixes and their locations against a state-of-the-art baseline (SEMINAL [21]). For each program, beyond the two repairs, participants were presented with the original ill-typed program, along with the standard OCAML compiler's error message and a short description of what the original author of the program intended it to do. From this study, we found that both the edit locations and final repairs produced by RITE were better than SEMINAL's in a statistically significant manner.

User Study Setup. Study participants were recruited from two public research institutes (University of California, San Diego and University of Michigan), and from advertisement on Twitter. Participants had to assess the quality of, and give comprehensible bug descriptions for, at least 5 / 10 stimuli. The study took around 25 minutes to complete. Participants were compensated by entering a drawing for an Amazon Echo voice assistant. There were 29 valid participants. We created the stimuli by randomly selecting a corpus of 21 buggy programs from the 1834 programs in our dataset where repairs were synthesized. From this corpus, each participant was shown 10 randomly-selected buggy programs, and two candidate repairs: one generated by RITE and one by SEMINAL. For both algorithms, we used the highest-ranked solution returned. Participant were always unaware which tool generated which candidate patch. Participants were then asked to assess the quality of each candidate repair

on a Likert scale of 1 to 5 and were asked for a binary assessment of the quality of each repair's edit location. We also collected self-reported estimates of both programming and OCaml-specific experience as well as qualitative data assessing factors influencing each participant's subjective judgment of repair quality. From the 29 participants, we collected 554 patch quality assessments, 277 each for Rite and Seminal generated repairs.

Results. In a statistically-significant manner, humans perceive that Rite's fault localization and final repairs are both of higher quality than those produced by Seminal ($p = 0.030$ and $p = 0.024$ respectively).[1] Regarding fault localization, we find that humans agreed with Rite-identified edit locations 81.6% of the time but only agreed with those of Seminal 74.0% of the time. As for the final repair, humans also preferred Rite's patches to those produced by Seminal. Specifically, Rite's repairs achieved an average quality rating of 2.41/5 while Seminal's repairs had an average rating of only 2.11/5, a 14% increase ($p = 0.030$), showing a statistically-significant improvement over Seminal.

Qualitative Comparison. We consider several case studies where there were statistically-significant differences between the human ratings for Rite's and Seminal's repairs. The task in Figure 10a is that `wwhile(f, b)` should return x where there exist values $v_0, ..., v_n$ such that: $b = v_0$, $x = v_n$, and for each i between 0 and $n-2$, we have $fv_i = (v_i+1, true)$ and $fv_{n-1} = (v_n, false)$. The task in Figure 10b is to return a list of `n` copies of `x`. The task in Figure 10c is to return the sum of the squares of the numbers in the list `xs`. Humans rated Rite's repairs better for the programs in Fig 10a and 10c. In both cases, Rite's found a solution which type-checks and conforms to the problem's semantic specification. Seminal, however, found a repair that was either incomplete (10a) or semantically incorrect (10c). On the other hand, in 10b, Rite does worse as the *second* parameter should be `n-1`. In fact, Rite's second ranked repair is the correct one, but it is equal to the first in terms of edit distance.

> Humans perceive both Rite's edit locations and final repair quality to be better than those produced by Seminal, a state-of-the-art OCaml repair tool, in a statistically-significant manner.

6.4 RQ4: Impact of Templates on Quality

Finally, we seek to evaluate whether Rite's template-guided approach is really at the heart of its effectiveness. To do so, as in § 6.2, we compared the results of using Rite's error messages synthesized from predicted templates to those generated by a Naive synthesizer that returns the first well-typed term (*i.e.* synthesized from the trivial "hole" template).

[1] All tests for statistical significance used the Wilcoxon signed-rank test.

User Study Setup. For this user study, we used a corpus of 20 buggy programs randomly chosen in § 6.3. For each of the programs we generated three messages: using Rite, using Seminal, and using the Naive approach but at the *same location* predicted by Rite. We then randomized and masked the order in which the tools' messages were reported, and asked three experts (authors of this paper who had not seen the output of any tool for any of those instances) to rate the messages as one of "Good", "Ok" or "Bad".

Results. Figure 11 summarizes the results of the rating. Since each of 20 programs received 3 ratings, there are a total of 60 ratings per tool. Rite dominates with 22 Good, 20 Ok and 18 Bad ratings; Seminal follows with only 12 Good, 11 Ok and 37 Bad; while Naive received no Good scores, 12 Ok scores and a dismal 48 Bad scores. On average (with Bad = 0, Ok = 0.5, Good = 1), Rite scored 0.53, Seminal 0.30, and Naive just 0.1. Our rating agreement kappa is 0.54, which is considered "moderate agreement".

> Repairs generated from predicted templates were of significantly higher quality than those from expert-biased enumeration (Seminal) or Naive enumeration.

7 Related Work

There is a vast literature on automatically repairing or patching programs: we focus on the most closely related work on providing feedback for novice errors.

Example-Based Feedback. Recent work uses *counterexamples* that show how a program went wrong, for type errors [36] or for general correctness properties where the generated inputs show divergence from a reference implementation or other correctness oracle [39]. In contrast, we provide feedback on how to fix the error.

Fault Localization. Several authors have studied the problem of *fault localization*, *i.e.* winnowing down the set of locations that are relevant for the error, often using slicing [12, 31, 40, 41], counterfactual typing [6] or bayesian methods [43]. Nate [37] introduced the BOAT representation, and showed it could be used for accurate localization. We aim to go beyond localization, into suggesting concrete *changes* that novices can make to understand and fix the problem.

Repair-model based feedback. Seminal [21] *enumerates* minimal fixes using an expert-guided heuristic search. The above approach is generalized to general correctness properties by [38] which additionally performs a *symbolic* search using a set of expert provided *sketches* that represent possible repairs. In contrast, Rite learns a template of repairs from a corpus yielding higher quality feedback (§ 6).

Corpus-based feedback. Clara [11] uses code and execution traces to match a given incorrect program with a

```
let rec wwhile (f, b) =
  let (b', c') = f b in
  if c' = true then wwhile (f b')
               else b'
```

RITE: `(f, b')`

SEMINAL: `((f b'); [[...]])`

(a) Rite (4.5/5) better than Seminal (1.1/5) with 12 responses $p = 0.002$.

```
let rec clone x n =
  if n <= 0 then [] else
    x :: clone (n-1)
```

RITE: `clone (n-1) n`

SEMINAL: `clone [[...]] (n-1)`

(b) Rite (1.5/5) worse than Seminal (4.1/5) with 18 responses $p = 0.0002$.

```
let sqsum xs =
  let f a x = a + (x ** 2) in
  let base = 0             in
  List.fold_left f base xs
```

RITE: `(x * x)`

SEMINAL: `(x + 2)`

(c) Rite (4.8/5) better than Seminal (1.2/5) with 17 responses $p = 0.0003$.

Figure 10. Three erroneous programs with the repairs that Rite and Seminal generated for the red error locations.

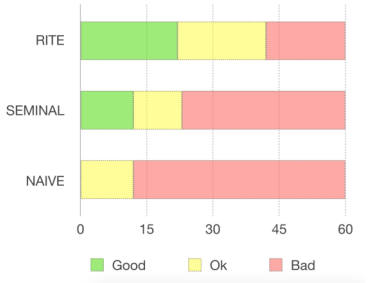


Figure 11. Rating the errors generated by Rite, Seminal and Naive enumeration.

"nearby" correct solution obtained by clustering all the correct answers for a particular task. The matched representative is used to extract repair expressions. Similarly, Sarfgen [42] focuses on structural and control-flow similarity of programs to produce repairs, by using AST vector embeddings to calculate distance metrics (to "nearby" correct programs) more robustly. Clara and Sarfgen are data-driven, but both assume there is a "close" correct sample in the corpus. In contrast, Rite has a more general philosophy that *similar errors have similar repairs*: we extract generic fix templates that can be applied to arbitrary programs whose errors (BOAT vectors) are similar. The Tracer system [1] is closest in philosophy to ours, except that it focuses on single-line compilation errors for C programs, where it shows that NLP-based methods like sequence-to-sequence predicting DNNs can effectively suggest repairs, but this does not scale up to fixing general type errors. We have found that OCaml's relatively simple *syntactic* structure but rich *type* structure make token-level seq-to-seq methods quite imprecise (*e.g.* *deleting* offending statements suffices to "repair" C but yields ill-typed OCaml) necessitating Rite's higher-level semantic features and (learned) repair templates.

Hoppity [7] is a DNN-based approach for fixing buggy JavaScript programs. Hoppity treats programs as graphs that are fed to a *Graph Neural Network* to produce fixed-length embeddings, which are then used in an LSTM model that generates a sequence of primitive edits of the program graph. Hoppity is one of the few tools that can repair errors spanning multiple locations. However, it relies solely on the learned models to generate a sequence of edits, so it doesn't guarantee returning valid JavaScript programs. In contrast, Rite, uses the learned models to get appropriate error locations and fix templates, but then uses a synthesis procedure to always generate type-correct programs.

Getafix [3] and Revisar [32] are two more systems that learn fix patterns using AST-level differencing on a corpus of past bug fixes. They both use *anti-unification* [19] for generalizing expressions and, thus, grouping together fix patterns. They cluster together bug fixes in order to reduce the search space of candidate patches. While Revisar [32] ends up with one fix pattern per bug category using anti-unification, Getafix [3] builds a hierarchy of patterns that also include the context of the edit to be made. They both keep before and after expression pairs as their fix patterns, and they use the before expression as a means to match an expression in a new buggy program and replace it with the after expression. While these methods are quite effective, they are only applicable in recurring bug categories *e.g.* how to deal with a null pointer exception. Rite on the other hand, attempts to generalize fix patterns even more by using the GAST abstractions, and predicts proper error locations and fix patterns with a learned model from the corpus of bug fixes, and so so can be applied to a diverse variety of errors.

Prophet [23] is another technique that uses a corpus of fixed buggy programs to learn a probabilistic model that will rank candidate patches. Patches are generated using a set of predefined transformation schemas and condition synthesis. Prophet uses logistic regression to learn the parameters of this model and uses over 3500 extracted program features to do so. It also uses an instrumented recompile of a faulty program together with some failing input test cases to identify what program locations are of interest. While this method can be highly accurate for error localization, their experimental results show that it can take up to 2 hours to produce a valid candidate fix. In contrast, Rite's pretrained models make finding proper error locations and possible fix templates more robust.

8 Conclusion

We have presented analytic program repair, a new data-driven approach to provide repairs as feedback for type errors. Our approach is to use a dataset of ill-typed programs and their fixed versions to learn a representative set of fix templates, which, via multi-class classification allows us to accurately predict fix templates for new ill-typed programs. These templates guide the synthesis of program repairs in a tractable and precise manner.

We have implemented our approach in Rite, and demonstrate, using a corpus of 4,500 ill-typed OCaml programs drawn from two instances of an introductory programming course, that Rite makes accurate fix predictions 69% of the time when considering the top three templates and surpass 80% when we consider the top six, and that the predicted templates let us synthesize repairs for over 70% of the test set in under 20 sec. Finally, we conducted a user study with 29 participants which showed that Rite's repairs are of higher quality than those from the state-of-the-art Seminal tool which incorporates several expert-guided heuristics for improving the quality of repairs and error messages. Thus, our results demonstrate the unreasonable effectiveness of data for generating better error messages.

Acknowledgments

We thank the anonymous referees and our shepherd Ke Wang for their excellent suggestions for improving the paper. This work was supported by the NSF grants (CCF-1908633, CCF1763674) and the Air Force grants (FA8750-19-2-0006, FA8750-19-1-0501).

References

[1] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *International Conference on Software Engineering: Software Engineering Education and Training*. 78–87. https://doi.org/10.1145/3183377.3183383

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. arXiv:cs.LG/1711.00740

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct 2019), 1–27. https://doi.org/10.1145/3360585

[4] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning Python Code Suggestion with a Sparse Pointer Network. arXiv:cs.NE/1611.08307

[5] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 209–210.

[6] Sheng Chen and Martin Erwig. 2014. Counter-factual Typing for Debugging Type Errors. In *Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 583–594. https://doi.org/10.1145/2535838.2535863

[7] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Laerning Graph Transformations to Detect and Fix Bugs in Programs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJeqs6EFvB

[8] Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103, 2 (1992), 235 – 271. https://doi.org/10.1016/0304-3975(92)90014-7

[9] Mark Gabel and Zhendong Su. 2010. A Study of the Uniqueness of Source Code. In *Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/1882291.1882315

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press, 180–184. http://www.deeplearningbook.org.

[11] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *Programming Language Design and Implementation* (2018). https://doi.org/10.1145/3192366.3192387

[12] Christian Haack and J B Wells. 2003. Type Error Slicing in Implicitly Typed Higher-Order Languages. In *Programming Languages and Systems*. Springer Berlin Heidelberg, 284–301. https://doi.org/10.1007/3-540-36575-3_20

[13] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer New York. https://doi.org/10.1007/978-0-387-84858-7

[14] 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 *Learning @ Scale*. 89–98. https://doi.org/10.1145/3051457.3051467

[15] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *International Conference on Software Engineering (ICSE '12)*. Piscataway, NJ, USA, 837–847.

[16] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*. 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[17] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. arXiv:cs.LG/1412.6980

[18] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *International Symposium on Software Testing and Analysis*. ACM, 165–176. https://doi.org/10.1145/2931037.2931051

[19] Temur Kutsia, Jordi Levy, and Mateu Villaret. 2011. Anti-Unification for Unranked Terms and Hedges. *Journal of Automated Reasoning* 52, 219–234. https://doi.org/10.4230/LIPIcs.RTA.2011.219

[20] Eelco Lempsink. 2009. *Generic type-safe diff and patch for families of datatypes*. Master's thesis. Universiteit Utrecht.

[21] Benjamin S Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-error Messages. In *Programming Language Design and Implementation*. ACM, 425–434. https://doi.org/10.1145/1250734.1250783

[22] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A practical framework for type inference error explanation. In *Object-Oriented Programming, Systems, Languages, and Applications*. 781–799. https://doi.org/10.1145/2983990.2983994

[23] 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)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[24] Matias Martinez, Laurence Duchien, and Martin Monperrus. 2013. Automatically extracting instances of code change patterns with AST analysis. In *2013 IEEE international conference on software maintenance*. IEEE, 388–391.

[25] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205.

[26] Jonathan P. Munson and Elizabeth A. Schilling. 2016. Analyzing Novice Programmers' Response to Compiler Error Messages. *J. Comput. Sci. Coll.* 31, 3 (Jan. 2016), 53–61. http://dl.acm.org/citation.cfm?id=2835377.2835386

[27] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning*. 807–814.

[28] Michael A Nielsen. 2015. *Neural Networks and Deep Learning*. Determination Press.

[29] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *International Symposium on Software Testing and Analysis*. ACM, 199–209. https://doi.org/10.1145/2001420.2001445

[30] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *Object Oriented Programming Systems Languages & Applications*. ACM, 525–542. https://doi.org/10.1145/2660193.2660230

[31] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. 2015. Skalpel: A Type Error Slicer for Standard ML. *Electron. Notes Theor. Comput. Sci.* 312 (24 April 2015), 197–213. https://doi.org/10.1016/j.entcs.2015.04.012

[32] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. arXiv:cs.SE/1803.03806

[33] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-hunk Program Repair. In *International Conference on Software Engineering*. 13–24. https://doi.org/10.1109/ICSE.2019.00020

[34] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (Jan 2015), 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

[35] Eric L Seidel and Ranjit Jhala. 2017. A Collection of Novice Interactions with the OCaml Top-Level System. https://doi.org/10.5281/zenodo.806813

[36] Eric L Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic Witnesses for Static Type Errors (or, Ill-typed Programs Usually Go Wrong). In *International Conference on Functional Programming*. 228–242. https://doi.org/10.1145/2951913.2951915

[37] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame: Localizing Novice Type Errors with Data-driven Diagnosis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 60 (Oct. 2017), 27 pages. https://doi.org/10.1145/3138818

[38] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *Acm Sigplan Notices* 48, 6 (2013), 15–26.

[39] 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

[40] Frank Tip and T B Dinesh. 2001. A Slicing-based Approach for Locating Type Errors. *ACM Trans. Softw. Eng. Methodol.* 10, 1 (Jan. 2001), 5–55. https://doi.org/10.1145/366378.366379

[41] Mitchell Wand. 1986. Finding the Source of Type Errors. In *Principles of Programming Languages*. 38–43. https://doi.org/10.1145/512644.512648

[42] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-driven Feedback Generation for Introductory Programming Exercises. In *Programming Language Design and Implementation*. 481–495. https://doi.org/10.1145/3192366.3192384

[43] Danfeng Zhang and Andrew C Myers. 2014. Toward General Diagnosis of Static Errors. In *Principles of Programming Languages*. 569–581. https://doi.org/10.1145/2535838.2535870