Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors

Yun Peng
The Chinese University of Hong Kong
Hong Kong, China
ypeng@cse.cuhk.edu.hk

Shuzheng Gao
The Chinese University of Hong Kong
Hong Kong, China
1155203205@link.cuhk.edu.hk

Cuiyun Gao*
Harbin Institute of Technology
Shenzhen, China
gaocuiyun@hit.edu.cn

Yintong Huo
The Chinese University of Hong Kong
Hong Kong, China
ythuo@cse.cuhk.edu.hk

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong, China
lyu@cse.cuhk.edu.hk

ABSTRACT

As a dynamic programming language, Python has become increasingly popular in recent years. Although the dynamic type system of Python facilitates the developers in writing Python programs, it also brings type errors at run-time which are prevalent yet not easy to fix. There exist rule-based approaches for automatically repairing Python type errors. The approaches can generate accurate patches for the type errors covered by manually defined templates, but they require domain experts to design patch synthesis rules and suffer from low template coverage of real-world type errors. Learning-based approaches alleviate the manual efforts in designing patch synthesis rules and have become prevalent due to the recent advances in deep learning. Among the learning-based approaches, the prompt-based approach which leverages the knowledge base of code pre-trained models via pre-defined prompts, obtains state-of-the-art performance in general program repair tasks. However, such prompts are manually defined and do not involve any specific clues for repairing Python type errors, resulting in limited effectiveness. How to automatically improve prompts with the domain knowledge for type error repair is challenging yet under-explored.

In this paper, we present TypeFix, a novel prompt-based approach with fix templates incorporated for repairing Python type errors. TypeFix first mines generalized fix templates via a novel hierarchical clustering algorithm. The identified fix templates indicate the common edit patterns and contexts of existing type error fixes. TypeFix then generates code prompts for code pre-trained models by employing the generalized fix templates as domain knowledge, in which the masks are adaptively located for each type error instead of being pre-determined. Experiments on two benchmarks, including BugsInPy and TypeBugs, show that TypeFix successfully repairs 26 and 55 type errors, outperforming the best baseline approach by 9 and 14, respectively. Besides, the proposed fix template mining approach can cover 75% of developers' patches in both benchmarks, increasing the best rule-based approach PyTER by more than 30%.

ACM Reference Format:
Yun Peng, Shuzheng Gao, Cuiyun Gao, Yintong Huo, and Michael R. Lyu. 2023. Domain Knowledge Matters: Improving Prompts with Fix Templates for Repairing Python Type Errors . In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Being used in most artificial intelligence and data science applications, Python becomes extremely popular in recent years. According to GitHub Octoverse [14], which records the state of open-source software, Python is the second most-used programming language in 2022. Moreover, Python continues to see gains in its usage across GitHub with a 22.5% year-over-year increase [14].

Python adopts a dynamic type system, in which the type of a variable will be resolved only at run-time. This enables fast prototyping and brings much convenience for developers to write an executable program. The catch, however, is that more type errors occur at run-time, threatening the reliability of Python applications. Oh *et al.* [33] find that about 30% of questions in Stack Overflow and issues in GitHub of Python are about type errors. To avoid type errors, Python Software Foundation accepts several Python Enhancement Proposals (PEPs) [21, 22, 44, 59] and releases a static type checker named *mypy* [32], allowing developers to add type annotations and check potential type conflicts statically. What's more, the recent research [1, 30, 35, 42] on type inference aims at statically inferring the types of variables, which further reduces the burden of manual type annotation. These approaches can reduce potential type errors but provide limited help to repair existing type errors.

To automatically fix type errors, Oh *et al.* [33] propose the first rule-based approach. They manually define nine templates and several synthesis rules to generate patches via dynamic analysis, but the manually defined templates suffer from low coverage of real-world type errors and designing patch synthesis rules requires substantial efforts from domain experts.

General learning-based automatic program repair (APR) approaches [7, 19, 26, 52, 57, 58] become quite popular and powerful

*Corresponding author

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

in recent years, since they are feature-agnostic and can automatically learn to generate patches from existing bug fixes, without explicit definitions of synthesis rules. Among the learning-based approaches, the Neural Machine Translation (NMT)-based approach that translates the buggy lines into correct lines was typically used in the past. Most recently, Xia *et al.* [52] propose the first prompt-based APR approach named *AlphaRepair* and obtain state-of-the-art performance. Unlike NMT-based APR approaches, AlphaRepair transforms the APR problem into a fill-in-the-blank problem by masking several tokens in buggy lines and invoking pre-trained models to predict the masked tokens. Despite the superior performance of the prompt-based approach over NMT-based approaches, the prompts in AlphaRepair are pre-defined, i.e., where to mask and how to add masks in buggy code are manually designed. Without domain-specific knowledge, the prompt-based approach can hardly fix the type errors with complex patterns [33]. However, automatically incorporating the prompt with domain knowledge is challenging due to different levels of type errors and various type error fixing patterns.

Our Work. To address the aforementioned challenge, we propose TypeFix, a domain-aware prompt-based approach for repairing type errors. TypeFix has two main phases: the template mining phase and the patch generation phase. The template mining phase aims at extracting and organizing fix templates from existing type error fixes. Fix templates are designed to handle type errors at different levels (e.g., expression level and statement level). TypeFix first parses type error fixes into *specific fix templates* and then employs a novel hierarchical clustering algorithm to abstract and merge the *specific fix templates* into *general fix templates*. The patch generation phase aims at exploiting the mined fix templates in the first phase for producing patches. TypeFix selects the most matched and commonly-used fix templates based on Breadth-First Search (BFS) and a frequency-aware ranking algorithm, and then generates code prompts by applying the ranked fix templates, and invokes CodeT5 [49] for prediction. TypeFix is fully automated and extendable, as it does not need manually defined templates as well as patch synthesis rules. Additionally, the minded fix templates enable the proposed prompt-based TypeFix to be aware of domain knowledge when generating patches.

We evaluate TypeFix on two benchmarks BugsInPy [50] and TypeBugs [33] by comparing it with four baselines including both the recent rule-based and learning-based approaches. In the BugsInPy benchmark, TypeFix successfully fixes 26 out of 54 type errors, outperforming the most effective baseline Codex [4] by 9. In the TypeBugs benchmark, TypeFix successfully fixes 55 out of 109 bugs, outperforming the most effective baseline PyTER [33] by 14. Experiments also show that the fix templates mined by TypeFix can cover about 75% of type errors in both benchmarks, much higher than PyTER which only covers 40% of the type errors. The results demonstrate the effectiveness of TypeFix in repairing Python type errors.

Contributions. We conclude our contributions as follows.

- To the best of our knowledge, TypeFix is the first domain-aware prompt-based approach for repairing Python type errors.
- We propose a novel fix template design that can handle type errors at different levels, along with a novel hierarchical clustering approach to mine various fix templates from existing type error fixes.
- Extensive experiments demonstrate the effectiveness of TypeFix compared with state-of-the-art rule-based and learning-based baselines, and the high coverage of the mined fix templates.

2 MOTIVATION

To better illustrate our motivation, we give an example in Listing 1. The type error in Listing 1 is from a popular GitHub project *scrapy* in the BugsInPy benchmark. The correct fix for this type error is to add a user-defined type conversion function *to_bytes* to the entire string, as shown in the green-colored line. We also provide the patches provided by the baseline approaches and TypeFix in Listing 1.

Baseline Approaches. CoCoNuT [26] is an NMT-based APR approach that translates the buggy line into the correct line. In the patch, it modifies the content of the string since the variable *user* and *password* are often used in authorization. However, this cannot fix the type error. AlphaRepair [52] is a prompt-based APR approach that masks the tokens in the buggy line to generate patches. In the patch, it masks the function name *unquote* and fills a new name *ascii* to generate the patch. Without the domain knowledge indicating there should be a new function call wrapping the entire buggy string, it fails to identify the correct location to add masks and thus fails to fix this type error. Codex [4] is a large language model from OpenAI. Powered by the huge knowledge base stored in the model, Codex identifies that this type error is related to *bytes* types, but it adds checks for the *user* and *password* instead of the entire string, failing to fix this type error. This may be because adding type conversions for variables is much more frequent than that for the entire expression. PyTER [33] is a rule-based approach via dynamic analysis. It fails to find the correct variable inducing the type error, and also cannot introduce the required user-defined type conversion function *to_bytes*.

```
# Buggy Code: scrapy/scrapy:f042ad
if user:
-     user_pass = '%s:%s' % (unquote(user), unquote(password))
+     user_pass = to_bytes('%s:%s' % (unquote(user), unquote(password)))
      raise ValueError('Port cannot be 0 or less.')
      creds = base64.b64encode(user_pass).strip()
else:
      creds = None
# Patches:
# Incorrect Patch provided by CoCoNuT
user_pass = 'Proxy-Authorization'%(unquote(user), unquote(password))
# Incorrect Patch provided by AlphaRepair
user_pass = '%s:%s' % (ascii(user), unquote(password))
# Incorrect Patch provided by Codex
if not isinstance(user, bytes):
    user = user.encode('ascii')
if not isinstance(password, bytes):
    password = password.encode('ascii')
# Incorrect Patch provided by PyTER
if isinstance(creds, bytes):
    creds = str(creds, 'utf-8')
# Correct Patch provided by TypeFix
```

```
user_pass = to_bytes('%s:%s' % (unquote(user), unquote(password)))
```

Listing 1: A type error in BugsInPy benchmark

TypeFix. Before fixing a type error, TypeFix first mines fix templates from existing type error fixes via hierarchical clustering. In the clustering process, TypeFix can generalize the fix pattern of adding a type conversion for a variable into that of adding a type conversion for an expression. Even though the later fix pattern has low occurrence frequency, TypeFix can still successfully identify and apply the fix pattern to this type error. Guided by the selected fix template, TypeFix adds a new function to wrap the original buggy string, and inserts masks for the name of the new function, instead of randomly masking several tokens. As a prompt-based APR approach, TypeFix also mitigates the problem of introducing user-defined type conversion functions in rule-based approaches like PyTER, since language models can learn from the contexts of the type error. Therefore, TypeFix can successfully fix the type error.

3 APPROACH

TypeFix contains two main phases: the template mining phase and patch generation phase, with the overview shown in Fig. 1. In the *template mining* phase, TypeFix aims at extracting domain-aware fix templates. TypeFix first parses existing type error fixes into specific fix templates and then employs a novel hierarchical clustering algorithm to abstract and merge them into general fix templates. TypeFix also organizes the specific to general fix templates into *clustering trees*. In the *patch generation* phase, TypeFix aims at generating patches for new buggy programs by incorporating prompts with the mined fix templates. Specifically, it selects and ranks the mined fix templates, and applies them on buggy code to automatically generate domain-aware code prompts. The CodeT5 [49] model is finally invoked to generate patches by filling the masks in the code prompts.

3.1 Mining Phase

The template mining phase mainly contains two stages: fix parsing and fix template mining. The fix parsing stage aims to transform type error fixes into specific fix templates, and the fix template mining stage abstracts and merges parsed specific fix templates into general fix templates via the proposed hierarchical clustering algorithm. We first give formal definitions of fix templates for ease of understanding.

3.1.1 Definition of Fix Template. To represent the domain knowledge of where and how to add masks in buggy code for building code prompts, we define fix template as a combination of three parts: *fix pattern*, *internal context* and *external context*. The fix pattern indicates how the buggy code is edited to fix the type error, the internal context pinpoints the locations for applying fix patterns to handle type errors at different levels, and the external context indicates the location of the internal context in the entire buggy program. The three components are all represented based on *template trees* which are defined below.

Definition 3.1.1.1 (Template Tree). A template tree is a tree (N, E, rt) with nodes N, edges E and root node $rt \in N$. An edge is a triple (n, n', r) where node n is the parent of n' with relation r. A node is a quadruple (bt, t, v, i) where $bt \in$ {Variable, Op, Literal, Builtin, Type, Attribute, Expr, Stmt} is the base type of node, t is the AST node type, v is the value, and i is the id. bt, t, and v have a special value *ABS* to represent a hole.

We define template trees based on the abstract syntax tree (AST) [9] of Python. Keeping the original AST node type t, we add a base type bt by re-classifying all original AST node types and attribute types into eight base types, and thus a base type can include multiple AST node types, for example, AST node types *BoolOp*, *BinOp*, and *UnaryOp* belong to the same base type *Expr*. We design base types to obtain a higher level of abstraction than that of ASTs. For instance, the above three AST node types can all be the conditions of *If* statements that serve as guards to prevent type errors. Representing the three AST node types as *Expr* to indicate general conditions help create more general fix templates.

Definition 3.1.1.2 (Fix Pattern). A fix pattern is a map $B\_Tree \rightarrow A\_Tree$, where B_Tree is a template tree of the buggy code, and A_Tree is a template tree of the fixed code.

Definition 3.1.1.3 (Internal Context). An internal context is a pair (IC_Tree, rn), in which IC_Tree is a template tree of the deepest statement where a fix pattern locates, and rn is a map $n \rightarrow (br, ar)$ where br and ar are edge relations, $n \in IC\_Tree.N$ indicates the node where B_Tree is removed with the edge $(n, B\_Tree.rt, br)$ and A_Tree is added with the edge $(n, A\_Tree.rt, ar)$.

The internal context is defined to handle edits at different levels. For example, some expression-level edits only modify single expressions in the statements, while other statement-level edits replace the entire statements. Since fix patterns only represent the edits themselves, we use internal contexts to represent the rest parts of the deepest statements for expression-level edits. The internal contexts are empty when the edits are at the statement level.

Definition 3.1.1.4 (External Context). An external context is a pair (BC_Tree, AC_Tree), where BC_Tree is a template tree of statements before the internal context and AC_Tree is a template tree of the statements after the internal context.

We define external contexts to provide extra location information when B_Tree in the fix pattern and the internal context are both empty. This usually happens when the fix is about adding a new statement and does not modify existing buggy code. The *fix template* is a combination of three components including the fix pattern, internal context and external context, defined as below.

Definition 3.1.1.5 (Fix Template). A fix template is a triple (P, IC, EC), where P ($P \neq \emptyset$) is the fix pattern, IC is the internal context, and EC is the external context.

We classify fix templates into four categories based on the fix patterns P:

- **Add**: $B\_Tree = \emptyset \wedge A\_Tree \neq \emptyset$
- **Remove**: $B\_Tree \neq \emptyset \wedge A\_Tree = \emptyset$
- **Insert**: $B\_Tree \neq \emptyset \wedge A\_Tree \neq \emptyset \wedge B\_Tree \subset A\_Tree$
- **Replace**: $B\_Tree \neq \emptyset \wedge A\_Tree \neq \emptyset \wedge B\_Tree \nsubseteq A\_Tree$

Note that there could be more fine-grained classifications under the *Replace* category such as shuffling the order of statements. However, we find that except for the *Insert* category, these cases are really rare (less than 10 cases in the dataset), so we just adopt the general *Replace* category.

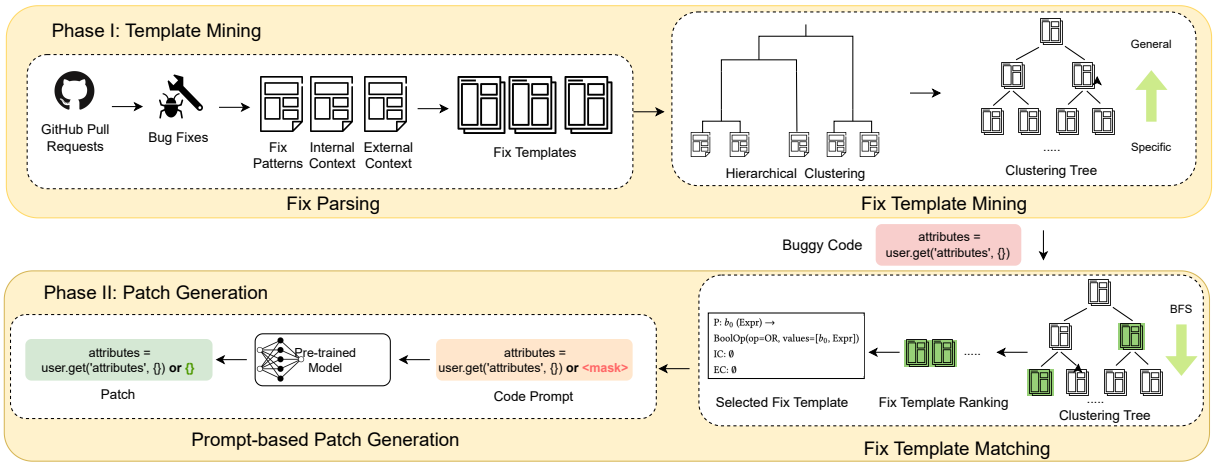


Figure 1: Overview of TypeFix

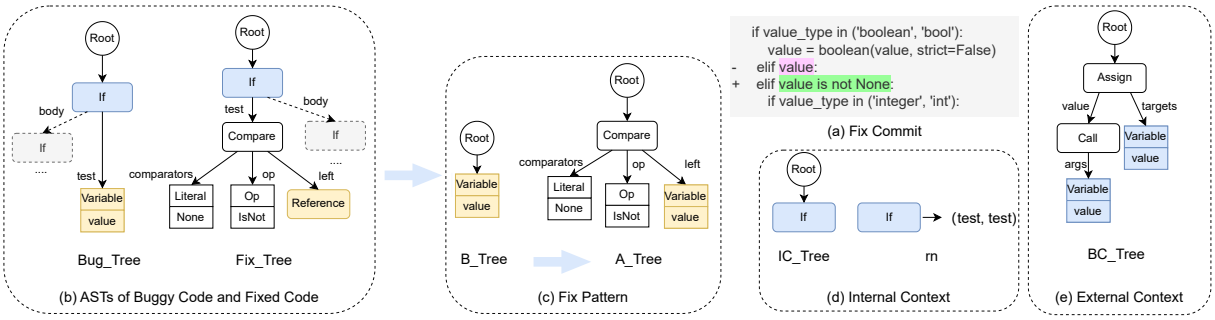


Figure 2: An example of fix parsing process on the fix commit ansible:075c6e.

3.1.2 Fix Parsing. In the fix parsing process, TypeFix parses all type error fixes into specific fix templates, with an example illustrated in Fig. 2.

Parsing Fix Patterns and Internal Contexts. Given a fix commit, TypeFix first extracts the line information of all added and deleted statements, and then walks through the ASTs of buggy code and fixed code to build template trees. To handle edits at different levels, TypeFix locates the deepest statement-level AST nodes that contain the modified lines, and extracts the corresponding sub-trees in buggy code and fixed code as *Bug_Tree* and *Fix_Tree*, respectively. For example, in the fix commit shown in Fig. 2(a), TypeFix locates the *If* nodes in the ASTs of buggy code and fixed code, since it is the deepest statement-level AST node containing the edits about variable *value*. Fig. 2(b) illustrates the extracted sub-trees as *Bug_Tree* and *Fix_Tree*. TypeFix then prunes the same sub-trees shared by *Bug_Tree* and *Fix_Tree* to leave only the changed part. For example, the bodies of *If* nodes (grey nodes) are pruned and only the conditions remain in Fig. 2(b).

As the pruned *Bug_Tree* and *Fix_Tree* contain only the edit, TypeFix can check whether the edit is at the expression level or the statement level. If *Bug_Tree* and *Fix_Tree* share the same root node, TypeFix determines that the edit does not rewrite the entire statement and thus it is at the expression level. Otherwise, TypeFix can determine that the edit is at the statement level. For instance, *Bug_Tree* and *Fix_Tree* in Fig. 2(b) have the same root node *If* (blue nodes), so the edit is at the expression level. For statement-level edits, the internal context is empty. For expression-level edits, TypeFix creates the internal context by extracting the same nodes shared by *Bug_Tree* and *Fix_Tree* to form a new template tree *IC_Tree*, and subtracts *IC_Tree* from *Bug_Tree* and *Fix_Tree* to build *B_Tree* and *A_Tree* in the fix pattern. The relations of edges that connect *IC_Tree* in the internal context and *B_Tree* and *A_Tree* in the fix pattern are also recorded in the internal context. In the example of Fig. 2, the *If* node is extracted as *IC_tree* in the internal context in Fig. 2(d), and the final *B_Tree* and *A_Tree* constitute fix pattern in Fig. 2(c).

Parsing External Contexts. TypeFix identifies statements that locate outside the scope of the internal context but have direct data dependencies with the fix pattern as external contexts. Specifically, it extracts statements that share the same variables with fix patterns before and after the internal context to build *BC_Tree* and *AC_Tree*, respectively. To simplify the fix template abstraction process, TypeFix also prunes the sub-trees in *BC_Tree* and *AC_Tree* that do not contain shared variables. For example, in Fig. 2, TypeFix identifies the statement *value = boolean(value, strict=False)* because it contains the same variable *value* used in the fix pattern. TypeFix builds a template tree *BC_Tree* based on this statement and prunes irrelevant sub-trees such as *strict=False*. Since there is no statement

after internal contexts that shares the same variables with fix pattern in Fig. 2(c), AC_Tree is left empty. The final parsed external context is shown in Fig. 2(e).

3.1.3 Fix Template Mining. In the fix template mining process, TypeFix abstracts and merges the specific fix templates into general fix templates via hierarchical clustering. The rationale of template mining is to ensure the least loss of domain knowledge in fix templates. Based on this rationale, TypeFix abstracts or merges the two most similar fix templates each time, and organizes specific to general fix templates as clustering trees. To measure the similarity between two fix templates, we define two kinds of similarity metrics: *value distance* and *structural distance*. The *structural distance* measures the ratio of nodes in two template trees that have the same type regardless of values (type matching), while the *value distance* measures the ratio of nodes in two template trees that have the same types and values (value matching).

Definition 3.1.3.1 (Fix Pattern Distances). The value distance d_p and structural distance sd_p between two template trees in fix patterns are defined as

$$d_p(t_1, t_2) = 1 - \frac{VM_p(t_1.rt, t_2.rt)}{\text{Num}(t_1) + \text{Num}(t_2)}$$

$$sd_p(t_1, t_2) = 1 - \frac{TM_p(t_1.rt, t_2.rt)}{\text{Num}(t_1) + \text{Num}(t_2)},$$

where $Num(t)$ indicates the number of nodes in the template tree t, and ValueMatch VM_p and TypeMatch TM_p are defined as

$$VM_p(n_1, n_2) = \begin{cases} 0 & n_1 \neq n_2 \\ 2 + \sum_{i \in \text{child}(n_1, n_2)} VM_p(n_1^i, n_2^i) & \text{otherwise} \end{cases}$$

$$TM_p(n_1, n_2) = \begin{cases} 0 & n_1.t \neq n_2.t \\ 2 + \sum_{i \in \text{child}(n_1, n_2)} TM_p(n_1^i, n_2^i) & \text{otherwise} \end{cases}$$

Definition 3.1.3.2 (Context Distances). The value distance d_c and structural distance sd_c between two template trees in contexts are defined as

$$d_c(t_1, t_2) = 1 - \frac{\text{MAX}(\text{LeafNode}(t_1), \text{LeafNode}(t_2), VM_c)}{\text{Num}(t_1) + \text{Num}(t_2)}$$

$$sd_c(t_1, t_2) = 1 - \frac{\text{MAX}(\text{LeafNode}(t_1), \text{LeafNode}(t_2), TM_c)}{\text{Num}(t_1) + \text{Num}(t_2)},$$

where MAX(a, b, c) pairs the elements in a and b, and finds the highest similarity c achieved by the pairs, and returns the number of pairs, $Num(t)$ indicates the number of nodes in the template tree t, and $LeafNode(t)$ returns the leaf node set of a template tree t. The ValueMatch VM_c and TypeMatch TM_c are defined as

$$VM_c(n_1, n_2) = \begin{cases} 0 & n_1 \neq n_2 \\ 2 + VM_c(n_1.parent, n_2.parent) & \text{otherwise} \end{cases}$$

$$TM_c(n_1, n_2) = \begin{cases} 0 & n_1.t \neq n_2.t \\ 2 + TM_c(n_1.parent, n_2.parent) & \text{otherwise} \end{cases}$$

To calculate the distances of fix patterns, we adopt a top-down methodology. We start with the root node and require two nodes to be type-matching or value-matching before we compare their child nodes. To calculate the distances of contexts, we adopt a bottom-up methodology. We start with the leaf nodes and require two nodes to be type-matching or value-matching before we compare their parent nodes. Such a difference is caused by the functionality of fix patterns and contexts. The template trees in the fix patterns are used to generate patch code, so based on the definition of ASTs the children nodes are meaningful only if their parent nodes are determined. On the contrary, the template trees in the contexts are used to match the locations that fix patterns should apply instead of generating code, so the children nodes contain more specific location information than the parent nodes. For example, in Fig. 2(d), even if we remove the node *Assign*, it can still match the original statement through *Call*, but if we remove the node *Variable*, it can match more general statements that have no direct data dependency with fix pattern in Fig. 2(b).

Template Abstraction. TypeFix does not abstract the whole fix template, instead, it abstracts one component, i.e., fix pattern, internal context or external context, each time. TypeFix abstracts the two similar components through a process named *Abstract*. Fig. 3 and Fig. 4 formally present the processes of *Abstract* on fix patterns and contexts, respectively.

The abstraction of fix patterns and contexts follows the aforementioned top-down and bottom-up methodology, respectively. Generally, there could be four cases when abstracting the template node a and b from two similar template trees:

- **Same Node**: a and b are exactly the same, and they can be reserved for the generalized fix template.
- **Value Abstraction**: a and b have the same types but different values. TypeFix creates a node with the same type and set the value as a special *ABS* token to indicate a hole.
- **Type Abstraction**: a and b have the same base types but different types and values. TypeFix creates a node with the same base type, and sets the type and value as a special *ABS* token to indicate a hole.
- **Node Removal**: a and b have no common attributes. TypeFix directly removes the two nodes.

TypeFix also prunes all child nodes in Type Abstraction and Node Removal, because the change of types for an AST node disables the functionality of its original child nodes.

Mining Fix Templates via Hierarchical Clustering. With the above-mentioned similarity metrics and abstraction processes, TypeFix selects similar fix templates and merges them via hierarchical clustering to build clustering trees. We give the definition of the clustering tree as follows.

Definition 3.1.3.3 (Clustering Tree). A clustering tree is a tree (T, E, rt) with fix template set T, edges E and root fix template $rt \in T$. An edge is a pair (t, t') where fix template t is the parent of fix template t', indicating that t is directly abstracted from t'.

To ensure the least loss of domain knowledge, TypeFix follows two strategies in the mining process. First, TypeFix follows the priority order of "external context > internal context > fix pattern" when selecting component pairs for abstraction, ensuring that abstraction of fix patterns happens only if no external context pairs and internal context pairs can be abstracted. Second, TypeFix prefers value abstraction to type abstraction, so it prioritizes component pairs, i.e., fix pattern pairs, internal context pairs or external context pairs, with a structural distance at 0.

We present the hierarchical clustering algorithm of TypeFix in Alg. 1. At the beginning of each iteration, TypeFix calculates the

$$\textbf{Abstract}(n_1(C^1_{r_1},...,C^1_{r_m}), n_2(C^2_{r_1},...,C^2_{r_n})) =$$

$$\begin{cases} n_1(O_{r_1},...,O_{r_k}) & \textbf{if } n_1 = n_2 \\ & \textbf{where } k = \min(m,n), \\ & p = \min(\text{len}(C^1_{r_i}), \text{len}(C^2_{r_i})),\ O_{r_i} = \{O^1_{r_i},...,O^p_{r_i}\}, \\ & O^j_{r_i} = \textbf{Abstract}(C^{1j}_{r_i}, C^{2j}_{r_i})\ \forall j \in [1,p] \\ & \text{(Same Node)} \\ o(O_{r_1},...,O_{r_k}) & \textbf{if } n_1.v \neq n_2.v \wedge n_1.t = n_2.t \wedge n_1.bt = n_2.bt \\ & \textbf{where } o.v = ABS,\ o.t = n_1.t,\ o.bt = n_1.bt, \\ & k = \min(m,n),\ p = \min(\text{len}(C^1_{r_i}), \text{len}(C^2_{r_i})), \\ & O_{r_i} = \{O^1_{r_i},...,O^p_{r_i}\}, \\ & O^j_{r_i} = \textbf{Abstract}(C^{1j}_{r_i}, C^{2j}_{r_i})\ \forall j \in [1,p] \\ & \text{(Value Abstraction)} \\ o & \textbf{if } n_1.t \neq n_2.t \wedge n_1.bt = n_2.bt \\ & \textbf{where } o.v = ABS,\ o.t = n_1.bt,\ o.bt = n_1.bt \\ & \text{(Type Abstraction)} \\ \emptyset & \textbf{otherwise} \\ & \text{(Node Removal)} \end{cases}$$

$$\textbf{where } n_1, n_2 \in A\_Tree.N \textbf{ or } n_1, n_2 \in B\_Tree.N, C^j_{r_i} = \{C^{jt}_{r_i}\}$$
$$\textbf{s.t. } \text{Edge}(n^j, C^{jt}_{r_i}, r_i) \in A\_Tree.E \textbf{ or } \text{Edge}(n^j, C^{jt}_{r_i}, r_i) \in B\_Tree.E$$

$$\textbf{Abstract}(P(a_1,b_1), P(a_2,b_2)) = P(a,b)$$
$$\textbf{where } a.rt = \textbf{Abstract}(a_1.rt, a_2.rt)$$
$$b.rt = \textbf{Abstract}(b_1.rt, b_2.rt)$$

Figure 3: The process of Abstraction for fix patterns.

$$\textbf{Abstract}(n_1(P^1_{r_1}), n_2(P^2_{r_2})) =$$

$$\begin{cases} n_1(O_r) & \textbf{if } n_1 = n_2 \\ & \textbf{where } O_r = \textbf{Abstract}(P^1_{r_1}, P^2_{r_2}) \text{ if } r_1 = r_2, \\ & O_r = \emptyset \text{ if } r_1 \neq r_2 \\ & \text{(Same Node)} \\ o(O_r) & \textbf{if } n_1.v \neq n_2.v \wedge n_1.t = n_2.t \wedge n_1.bt = n_2.bt \\ & \textbf{where } o.v = ABS,\ o.t = n_1.t,\ o.bt = n_1.bt, \\ & O_r = \textbf{Abstract}(P^1_{r_1}, P^2_{r_2}) \text{ if } r_1 = r_2, \\ & O_r = \emptyset \text{ if } r_1 \neq r_2 \\ & \text{(Value Abstraction)} \\ o & \textbf{if } n_1.t \neq n_2.t \wedge n_1.bt = n_2.bt \\ & \textbf{where } o.v = ABS,\ o.t = n_1.bt,\ o.bt = n_1.bt \\ & \text{(Type Abstraction)} \\ \emptyset & \textbf{otherwise} \\ & \text{(Node Removal)} \end{cases}$$

$$\textbf{where } n_1, n_2 \in IC\_Tree.N \textbf{ or } n_1, n_2 \in BC\_Tree.N$$
$$\textbf{or } n_1, n_2 \in AC\_Tree.N, \text{Edge}(n^j, P^j_{r_j}, r_j) \in IC\_Tree.E$$
$$\textbf{or } \text{Edge}(n^j, P^j_{r_j}, r_j) \in BC\_Tree.E \textbf{ or } \text{Edge}(n^j, P^j_{r_j}, r_j) \in AC\_Tree.E$$

$$\textbf{Abstract}(IC(a_1), IC(a_2), Pairs(c)) = IC(a)$$
$$\textbf{where } \text{LeafNode}(a) = \{\textbf{Abstract}(p^i_1, p^i_2)\}\ \forall p^i \in c$$
$$\textbf{Abstract}(EC(a_1,b_1), EC(a_2,b_2), Pairs(c_1,c_2)) = EC(a,b)$$
$$\textbf{where } \text{LeafNode}(a) = \{\textbf{Abstract}(p^i_1, p^i_2)\}\ \forall p^i \in c_1$$
$$\text{LeafNode}(b) = \{\textbf{Abstract}(p^i_1, p^i_2)\}\ \forall p^i \in c_2$$

Figure 4: The process of Abstraction for both internal context and external context.

distances of three components for every two fix templates (lines 3~4). TypeFix then removes duplicated fix templates. Based on the calculated distances, TypeFix first handles external context pairs (lines 7 ~ 13), then internal context pairs (lines 16 ~ 22), and finally fix patterns (lines 25 ~ 30). If any abstraction or merge happens, the current iteration will be terminated and a new iteration will begin (lines 14 and 23).

When handling external context pairs, TypeFix groups the fix templates with the same internal contexts and fix patterns into different clusters (line 6). When handling internal context pairs, TypeFix groups the fix templates with the same fix patterns into different clusters (line 15). When handling fix patterns, all fix templates are grouped into one single cluster (line 24). This ensures that only fix templates in the same cluster can be abstracted and merged into more general fix templates under the priority order. For each cluster, TypeFix selects the certain components with the lowest distance in two fix templates (lines 8, 17, 26), and abstracts them into more general components in each iteration (lines 10, 19, 27). The selection has two stages. In the first stage, only components with a structural distance of 0 in the fix templates are considered to prioritize value abstraction. In the second stage, when no such component exists, the rest components are considered. Note that there could be trivial abstractions such as removing all nodes for a template tree so that an empty template tree can represent any code. As empty fix patterns provide no domain knowledge for patch generation, TypeFix does not select two fix patterns whose distance and structural distance are both at 1 in the fix templates for further abstraction.

At the end of each iteration, the new fix templates are included in the set, and the old fix templates are removed from the set (lines 11, 20, 28). The relationships between the new fix templates and the old fix templates are recorded in the clustering tree (lines 12, 21, 29). The merge of fix templates happens only when handling external contexts since the fix patterns and internal contexts of fix templates are already required to be the same at this time. TypeFix also records the number of instances each fix template represent in the training set to facilitate fix template ranking in the next phase. When the template mining process completes, clustering trees that contain specific to general fix templates are generated, facilitating the next patch generation phase.

3.2 Patch Generation Phase

The patch generation phase of TypeFix mainly contains two processes: fix template matching and prompt-based patch generation.

Algorithm 1 Fix Template Mining

Input: A set of parsed specific fix templates, T
Output: Mined fix templates, CT

$CT \leftarrow T$
while isChanged(CT) **do**
  $D_p, D_{IC}, D_{EC} \leftarrow$ CalculateValueDistances(CT)
  $SD_p, SD_{IC}, SD_{EC} \leftarrow$ CalculateStructuralDistances(CT)
  $CT \leftarrow$ Deduplicate(CT)
  $clusters \leftarrow$ selectClusters(CT, D_P, D_{IC}, D_{EC}, SD_{EC})
  for $c \in clusters$ **do** ▷ Handle external contexts
    $t_1, t_2 \leftarrow$ argmin(c, D_{EC}); $nt \leftarrow t_1$
    $pairs \leftarrow$ getPairs($D_{EC}(t_1, t_2)$)
    $nt.EC \leftarrow$ Abstract($t_1.EC, t_2.EC, pairs$, Context)
    $CT \leftarrow CT - \{t_1, t_2\} + \{nt\}$
    $t_1.parent, t_2.parent \leftarrow nt$
  end for
  continue if isChanged(CT)
  $clusters \leftarrow$ selectClusters(CT, D_P, D_{IC}, SD_{IC})
  for $c \in clusters$ **do** ▷ Handle internal contexts
    $t_1, t_2 \leftarrow$ argmin(c, D_{EC}); $nt_1 \leftarrow t_1$; $nt_2 \leftarrow t_2$
    $pairs \leftarrow$ getPairs($D_{IC}(t_1, t_2)$)
    $nt_1.IC, nt_2.IC \leftarrow$ Abstract($t_1.IC, t_2.IC, pairs$, Context)
    $CT \leftarrow CT - \{t_1, t_2\} + \{nt_1, nt_2\}$
    $t_1.parent \leftarrow nt_1$; $t_2.parent \leftarrow nt_2$
  end for
  continue if isChanged(CT)
  $clusters \leftarrow$ selectClusters(CT, D_P, SD_P)
  for $c \in clusters$ **do** ▷ Handle fix patterns
    $t_1, t_2 \leftarrow$ argmin(c, D_P); $nt_1 \leftarrow t_1$; $nt_2 \leftarrow t_2$
    $nt_1.P, nt_2.P \leftarrow$ Abstract($t_1.P, t_2.P$, Pattern)
    $CT \leftarrow CT - \{t_1, t_2\} + \{nt_1, nt_2\}$
    $t_1.parent \leftarrow nt_1$; $t_2.parent \leftarrow nt_2$
  end for
end while

The fix template matching process aims to select and rank appropriate fix templates that could be applied to the buggy program. The prompt-based patch generation process aims to generate candidate patches by applying selected fix templates to generate code prompts and invoking code pre-trained models for mask prediction.

3.2.1 Fix Template Matching. In the fix template matching process, TypeFix selects matched fix templates on clustering trees via Breadth-First Search (BFS) and then ranks fix templates with frequency and abstraction ratio.

Selecting Fix Templates. Given a buggy program, TypeFix parses the bug lines into a template Tree *Bug_Tree*, and the contexts before and after the bug lines into template trees *BBug_Tree* and *ABug_Tree*, respectively. TypeFix compares the triple (*Bug_Tree*, *BBug_Tree*, *ABug_Tree*) with fix templates in the clustering trees to find the appropriate fix templates. We define the following rules to check whether a buggy program matches a fix template.

Definition 3.2.1.1 (Template Node Match). For two template nodes a and b, a matches b if $a.value$ matches $b.value$ and $(a.t, a.bt)$ matches $(b.t, b.bt)$. $a.value$ matches $b.value$ if $b.value = ABS \vee a.value = b.value$. $(a.t, a.bt)$ matches $(b.t, b.bt)$ if $a.bt = b.bt \wedge (a.bt = b.t \vee a.t = b.t)$.

Definition 3.2.1.2 (Template Tree Match). For two template trees A and B, A matches B if there is a node $a \in A.N$ where a matches to $B.rt$ and there exists node maps $\{ac_1 \rightarrow bc_1, ..., ac_n \rightarrow bc_n\}$ where ac_i matches bc_i, $\{ac_1, ..., ac_n\} \subseteq a.children$, $ac_{i+1}.id > ac_i.id$, and $\{bc_1, ..., bc_n\} = B.rt.children$. A always matches B if $B = \emptyset$.

Definition 3.2.1.3 (Fix Template Match). For a buggy program (*Bug_Tree*, *BBug_Tree*, *ABug_Tree*) and a fix template (P, IC, EC), the buggy program matches the fix template if *BBug_Tree* matches $EC.BC\_Tree$, *ABug_Tree* matches $EC.A\_Tree$ and *Bug_Tree* matches Concat($IC.IC\_Tree$, $P.B\_Tree$,$IC.rn$), where Concat(a, b, rn) indicates concatenating template tree b to template tree a with edge (n, $b.rt$, $rn[n].br$).

With the above rules, TypeFix starts with the root fix template (most general fix templates) of each clustering tree and walks through the clustering tree via bread-first search (BFS) until it finds the deepest fix template (most specific fix templates) matched by the buggy program. These fix templates are collected to be ranked in the next step.

Ranking Fix Templates. TypeFix ranks the fix templates before applying them to the buggy program. To provide the most domain knowledge for pre-trained models in the patch generation process, TypeFix utilizes a two-step strategy to prioritize fix templates.

TypeFix groups the fix templates with the same concatenated template tree of *IC_Tree* and *B_Tree*. These fix templates provide different fix solutions for the same buggy pattern. TypeFix ranks fix templates in one group based on the number of training instances they represent because a larger number indicates that the fix template is used more frequently on the given buggy program. TypeFix then ranks the groups based on the abstraction ratio of *A_Tree* of the first fix template in each group. The abstraction ratio of a template tree is defined by the ratio of nodes whose values or types are *ABS* tokens. A higher abstraction ratio of *A_Tree* indicates less domain knowledge associated, so that code pre-trained models need to predict more information before they can generate complete candidate patches. For example, an abstraction ratio of 1.0 indicates the fix template actually is a huge hole and there is no domain knowledge assisting the generation of patches. Therefore, TypeFix prioritizes the groups with a lower abstraction ratio to include more domain knowledge in the patch generation process.

3.2.2 Prompt-based Patch Generation. In the process, TypeFix applies ranked fix templates on the buggy program and generates code prompts. CodeT5 model is then invoked to fill the masks in code prompts and generate candidate patches.

Applying Fix Templates. For each selected fix template, TypeFix completes the *A_Tree* in its fix pattern by adding dummy AST nodes, i.e., AST nodes with values of *ABS* tokens, as placeholders, because some child AST nodes are removed in the fix template mining process. TypeFix then replaces the sub-tree AST of the buggy program that matches *B_Tree* with the completed *A_Tree*, and converts the modified AST of the buggy program to code prompts. Code prompts are source code that contains *ABS* tokens as masks to be predicted by code pre-trained models.

Generating Patches. Most code pre-trained models are trained to predict masks in source code, thus they can naturally be used to predict the value of *ABS* tokens in the code prompts. In this

paper, we choose CodeT5 [49] as the code pre-trained model in the patch generation process, since it is specially designed for the code generation task [49]. When generating patches, TypeFix replaces the *ABS* tokens in the code prompt with ordered mask tokens used in CodeT5, e.g., <extra_id_0>, ..., <extra_id_99>. TypeFix then invokes CodeT5 to predict tokens for each mask. The predicted values for the masks are filled into the code prompts to generate candidate patches.

Validating Patches. TypeFix adopts the classic generate-and-validate methodology in patch generation. For the generated patches, TypeFix first filters out those with syntax errors, and then runs the test suite on each patch to find plausible patches, i.e., those can successfully pass all test cases. Plausible patches are further examined by the authors to identify correct patches, i.e., those are semantically identical to the developer patch when ignoring I/O side effects such as messages in *print* statements.

4 EXPERIMENTAL DESIGN

4.1 Dataset

Training Set. Following previous work [33], we build a dataset for the fix template mining process of TypeFix and the training of baselines. We collect 8,722 merged pull requests from GitHub that contain the term "fix type error". We extract the fixes from the commits in collected pull requests. We remove the overlong commits that contain more than 50 lines of modified code. Finally, we get 10,981 fixes to form the training set.

Benchmarks. Following previous work [33], we use two benchmarks BugsInPy [50] and TypeBugs [33]. The two benchmarks initially separate type errors by commits, but we find that a single commit can also involve more than one type errors in different locations. To avoid the correct fix of one type error being hidden by another type error, we further split the commits that contain two or more type errors into multiple ones. We also remove the duplicated type errors, i.e., those that have the same commit signatures, in two benchmarks. Finally, we get 54 type errors from BugsInPy and 109 type errors from TypeBugs.

4.2 Baselines

We compare TypeFix with the following four baselines.

PyTER. PyTER [33] is a rule-based APR approach designed for repairing Python type errors. It has nine pre-defined templates and several rules to synthesize templates to generate candidate patches.

AlphaRepair. AlphaRepair [52] is the state-of-the-art prompt-based approach for general-purpose APR. It masks tokens in the buggy code based on some general prompt templates and invokes code pre-trained models to generate patches.

CoCoNuT. CoCoNuT [26] is an NMT-based approach for general-purpose APR. It translates the buggy code into candidate patches.

Codex. Codex [4] is a large GPT model fine-tuned on publicly available code from GitHub. It is designed by OpenAI and used to power GitHub Copilot [29] service.

4.3 Metrics

We adopt the commonly used **Correct** and **Plausible** metrics in previous work [19, 26, 33, 52] to evaluate the performance of TypeFix on repairing type errors. Besides, we add a new metric named

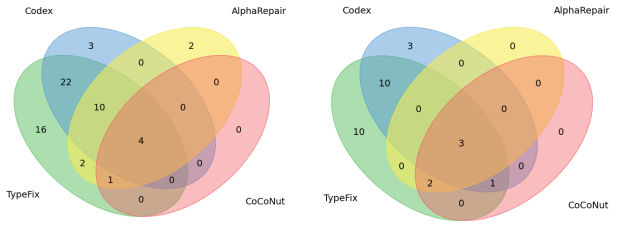
Table 1: Evaluation results of TypeFix compared with three baselines. Results are presented in the Correct/Plausible format. Fix rate is the ratio of correct patches.

TypeBugs						
Project	**#B**	**TypeFix**	**PyTER**	**Codex**	**AlphaRepair**	**CoCoNuT**
airflow	14	9/9	4/4	7/7	1/6	0/4
beets	1	0/0	0/1	0/0	0/0	0/0
core	9	7/7	5/7	4/5	4/4	2/3
kivy	1	0/0	0/1	0/0	0/1	0/1
luigi	2	0/2	0/0	0/2	1/2	0/0
numpy	3	0/3	0/2	0/1	0/2	0/0
pandas	48	21/32	17/27	18/19	11/22	3/10
rasa	2	2/2	0/0	2/2	0/0	0/0
requests	4	4/4	4/4	2/2	0/1	0/0
rich	4	2/3	0/1	1/1	0/0	0/0
salt	8	5/8	5/5	4/5	1/5	0/2
sanic	2	0/0	2/2	0/0	0/0	0/0
scikit-learn	7	2/3	2/3	1/2	0/0	0/0
tornado	1	0/0	1/1	0/0	0/0	0/0
Zappa	3	3/3	1/1	0/0	1/3	0/1
Total	109	**55/76**	41/59	39/46	19/46	5/21
Fix Rate (%)	-	**50.5**	37.6	35.8	17.4	4.6
BugsInPy						
Project	**#B**	**TypeFix**	**PyTER**	**Codex**	**AlphaRepair**	**CoCoNuT**
ansible	1	0/0	0/0	0/0	0/0	0/0
fastapi	1	1/1	0/0	1/1	0/0	0/0
keras	7	4/6	1/1	0/3	0/4	0/4
luigi	7	4/5	3/5	3/3	0/0	0/0
pandas	19	4/13	4/6	2/6	3/10	3/8
scrapy	12	10/11	5/7	10/12	1/4	2/4
spacy	1	0/1	0/1	0/0	0/1	0/1
tornado	2	1/1	1/1	1/1	0/1	0/1
youtube-dl	4	2/3	1/1	0/2	1/1	1/1
Total	54	**26/41**	15/22	17/28	5/21	6/19
Fix Rate (%)	-	**48.1**	27.8	31.5	9.3	11.1

Template Coverage to evaluate the number of developer patches covered by the fix templates mined by TypeFix and pre-defined ones from PyTER. **Template Coverage** is defined by the ratio of bugs whose developer patch matches a fix template of an approach.

4.4 Implementation

The entire framework of TypeFix is implemented using Python, which contains more than 10,000 lines of code. We adopt the CodeT5-base [40] model to predict the masks in code prompts and generate candidate patches. For PyTER, AlphaRepair and CoCoNuT, we directly use the replication packages released by the authors and re-implement them on our task. We train CoCoNuT with its original training set and the training set we collected to adapt it to fix Python type errors. Since Codex is not publicly available, we use the public API [34] of engine *code-davinci-002* provided by OpenAI to query it with prompts. We use a similar prompt from previous work [51]. The only difference is that we use three examples instead



(a) TypeBugs (b) BugsInPy

Figure 5: Venn diagram of correct patches provided by learning-based APR approaches.

of two at the beginning of the prompt and only mask the buggy line to maximize the performance of Codex. We make all other settings consistent with previous work [19, 26, 33, 51, 52]. All experiments are conducted on a Linux machine (Ubuntu 20.04) with two Intel Xeon@2.20GHZ CPUs, one NVIDIA A100-SXM4-40GB GPU and 256GB RAM.

5 EVALUATION

In this section, we evaluate the performance of TypeFix on the following three research questions:

- **RQ1**: How effective is TypeFix to fix type errors?
- **RQ2**: How capable is TypeFix to mine fix templates from existing bug fixes?
- **RQ3**: When does TypeFix fail to fix type errors?

5.1 RQ1: Effectiveness of TypeFix

To evaluate the effectiveness of TypeFix on repairing type errors, we compare TypeFix with state-of-the-art rule-based APR approaches and learning-based APR approaches. Table 1 presents the performance of TypeFix along with baseline approaches on two benchmarks TypeBugs and BugsInPy.

Comparison with Rule-based Approach. As can be seen in Table 1, TypeFix can successfully fix 55 type errors in TypeBugs and 26 type errors in BugsInPy, outperforming rule-based approach PyTER by 14 and 11 type errors, respectively. We attribute the improvement of TypeFix to the higher coverage of fix templates mined from existing type error fixes and the generated domain-aware code prompts. Furthermore, we analyze the unique type errors that TypeFix and PyTER can fix in two benchmarks and present the results in Table 2. We find that TypeFix obtains 24 and 16 unique type error fixes in TypeBugs and BugsInPy, respectively, while PyTER only obtains 10 and 5 unique type error fixes in TypeBugs and BugsInPy, respectively. This further demonstrates the effectiveness of TypeFix when compared with PyTER.

Comparison with Learning-based Approaches. From Table 1 we can see that TypeFix, Codex and AlphaRepair generally perform much better than CoCoNuT, indicating the superior performance of prompt-based approaches. When comparing TypeFix with AlphaRepair which adopts general domain-unaware prompt templates, we find that TypeFix achieves a 1× ∼ 4× larger fix rate than AlphaRepair. This indicates that general domain-unaware prompt templates

Table 2: Comparison of the number of unique type error fixes and template coverage between TypeFix and PyTER.

Approach	TypeBugs		BugsInPy	
	#Unique	Coverage	#Unique	Coverage
TypeFix	24	83 (76.1%)	16	40 (74.1%)
PyTER	10	46 (42.2%)	5	18 (33.3%)

Table 3: Statistics of fix templates mining in TypeFix.

Category	#Instances	#Clustering Trees (>1/>5)	Mining Time/s
Add	2,656	150/27	2,819
Remove	570	10/5	59
Insert	1,648	350/47	659
Replace	6,107	184/70	32,621

Table 4: Ablation results.

	TypeBugs		BugsInPy	
	#Correct	#Plausible	#Correct	#Plausible
No Template	19	41	6	20
Add	+11	+10	+3	+5
Remove	+1	+1	+0	+0
Replace	+6	+8	+4	+9
Insert	+18	+16	+13	+17
Total	55	76	26	41

such as randomly replacing several tokens in code can hardly handle complicated type errors. Compared with the most advanced code pre-trained model Codex, TypeFix still obtains a significant improvement by fixing 16 and 9 more type errors than Codex in TypeBugs and BugsInPy, respectively. This improvement further demonstrates the importance of domain knowledge for repairing type errors, even though Codex has a much larger parameter size (12B) than that (220M) of the CodeT5 model utilized by TypeFix.

In addition to the total number of type errors fixed by each approach, we further evaluate the number of unique type error fixes. Fig. 5 presents the unique type errors that TypeFix and three learning-based approaches can correctly fix in the format of Venn diagrams. We observe that TypeFix obtains 16 and 10 unique type error fixes in TypeBugs and BugsInPy, respectively, while other approaches only obtain 0 ∼ 3 unique bug fixes in two benchmarks. This indicates that the contribution of domain-aware fix templates cannot be replaced by the combination of existing learning-based approaches.

Answer to RQ1: TypeFix successfully fixes 55 and 26 bugs in two benchmarks, outperforming state-of-the-art approaches by

at least 14 bugs and 9 bugs, respectively. Meanwhile, TypeFix obtains the most unique type error fixes in two benchmarks.

5.2 RQ2: Capability of TypeFix to Mine Fix Templates

To comprehensively investigate the capability of TypeFix to mine fix templates, we focus on the performance of TypeFix in template mining and the usefulness of fix templates mined by TypeFix.

Table 3 presents the performance of TypeFix in fix template mining process. Starting with thousands of existing bug fixes, TypeFix can mine 10 ∼ 350 clustering trees. After discarding the clustering trees with occurrence frequency lower than a threshold (5 in this paper), TypeFix finally gets 5 ∼ 70 clustering trees. The mining process generally takes shorter than one minute to at most nine hours. Table 2 presents the template coverage achieved by TypeFix and PyTER. From it we can observe that fix templates mined by TypeFix can cover about 75% of type errors in two benchmarks while the manually defined fix templates in PyTER can only cover about 30% ∼ 40% of type errors.

To further study how fix templates mined by TypeFix can help the patch generation process, we conduct an ablation study on fix templates under each category. Following previous study [52], we start with the case that no fix template is applied, i.e., the CodeT5 model is asked to generate a complete new line to replace the original buggy line. We then gradually apply fix templates under *Add*, *Remove*, *Replace* and *Insert* categories, and observe the number of new correct and plausible patches, respectively. We show the results in Table 4. We can find that all four categories of fix templates contribute to generating correct patches. This demonstrates the contribution of domain knowledge stored in the fix templates. We also note that fix templates under *Insert* and *Add* categories contribute the most. The reason could be attributed to that developers often add guards to guarantee the desired types or directly convert input types into desired types when fixing type errors.

Answer to RQ2: TypeFix achieves a template coverage of about 75% on both benchmarks. Ablation results also demonstrate the usefulness of fix templates mined by TypeFix under each category.

5.3 RQ3: Limitations of TypeFix

Our experiments also show the limitations of TypeFix as it cannot fix all type errors in two benchmarks. By analyzing the type errors that TypeFix cannot fix in two benchmarks, we conclude two possible limitations.

The first limitation is that TypeFix cannot always find matched fix templates to the current buggy program. Based on Table 2, we can find that even if fix templates mined by TypeFix can cover as many as 75% cases in two benchmarks, there exist a few cases (∼25%) that do not share similar patterns with instances in the training set. An example is illustrated in the first type error of Listing 2. To fix this type error, the developer changes a list comprehension into an attribute access, which does not appear in the training set. TypeFix thus cannot find proper fix templates for this type error and fails to fix it. This limitation can be mitigated by adapting TypeFix to new datasets, so that TypeFix can mine new fix templates to improve the template coverage.

```
#Type Error 1: apache/airflow:892d4d
if conf.getboolean('core', 'store_dag_code',\
fallback=False):
-     DagCode.bulk_sync_to_db([dag.fileloc for dag in orm_dag])
+     DagCode.bulk_sync_to_db([orm_dag.fileloc])
#Type Error 2: pandas-dev/pandas:a3e903
elif (is_extension_array_dtype(left) or\
- is_extension_array_dtype(right)):
+ (is_extension_array_dtype(right) and not is_scalar(right))):
        return dispatch_to_extension_op(op, left, right)
```

Listing 2: Two type errors that TypeFix fail to fix

The second reason is that the CodeT5 model TypeFix uses sometimes cannot generate the correct patches even if the correct fix template is given. By comparing Table 1 and Table 2, we can find that fix templates mined by TypeFix can cover 83 bugs in TypeBugs but only 55 of them are correctly fixed. This indicates the limitations of CodeT5 when generating candidate patches from code prompts. We also show an example as the second type error of Listing 2. In this type error, we need to add a new condition as the guards and this fix pattern is commonly used in the wild. However, CodeT5 cannot give *is_scalar* as the new condition and thus TypeFix fails to fix this type error. We believe this limitation can be mitigated by using more advanced code pre-trained models, as the parameter size of CodeT5 is only 220M.

Answer to RQ3: TypeFix sometimes fails to fix type errors due to the limited performance of pre-trained code models and a few cases (∼ 25%) that mined fix templates cannot cover.

6 RELATED WORK

6.1 Automatic Program Repair

As an important method to improve the reliability of software, automatic program repair (APR) has draw a lot of attention [12, 31] in recent years. Currently, most APR approaches can be classified into rule-based approaches and learning-based approaches.

Rule-based Approaches. Rule-based APR approaches leverage pre-defined templates and rules to generate patches for bugs via static and dynamic analysis. There are a series of rule-based APR approaches designed for Java programs via constant-solving [28, 54], fuzzing [11], testcase generation [53], bytecode mutation [13], and for the memory bugs of C programs [10, 16, 17, 20, 55]. For Python programs, PyTER [33] utilizes nine pre-defined templates with type-aware fault localization to repair type errors. Traditional rule-based approaches are domain-aware but can be used to fix a limited amount of real-world bugs. TypeFix addresses this challenge by automatically mining fix templates from real-world bug fixes. Different from previous work on mining code edit patterns [2, 6, 38, 39], TypeFix implements a novel fix template design to handle type errors at different levels.

Learning-based Approaches. Learning-based APR approaches become quite popular and demonstrate their superior performance recently. Motivated by the study [47, 48, 56] of neural machine translation (NMT) [43], there are many research efforts being devoted to

NMT-based APR approaches. SequenceR [5] presents a sequence-to-sequence LSTM model for program repair. DLFix [23] leverages tree-based RNN to transform code inputs and generate patches. CoCoNuT [26] separates the context and buggy line in NMT-based APR. CURE [19] is the first approach that integrates pre-trained models in NMT-based APR, followed by work [7, 27]. Recoder [58] generates code edits instead of modified code. RewardRepair [57] uses execution-based backpropagation to improve the compilation rate of patches generated by NMT-based APR approaches. AlphaRepair [52] is the first prompt-based APR approach that transforms the APR problem into a fill-in-the-blank problem. However, general domain-unaware prompts AlphaRepair uses can hardly handle complicated type errors. TypeFix addresses the problem by incorporating domain knowledge with proposed fix templates and mining them through existing type error fixes.

6.2 Pre-trained Language Models

Making use of large-scale unlabeled data in the wild, pre-trained language models are proven to be quite effective in the natural language processing (NLP) field. Inspired by such a success, many code pre-trained models are proposed. Most code pre-trained models utilize the same Transformer [45] structure. CodeBERT [8] is a BERT-style model pre-trained on both programming languages and natural languages. GraphCodeBERT [15] utilizes the data flow graphs in the pre-training stage. CodeGPT [25] is a GPT [37]-style model pre-trained on Python and Java functions. Codex [4] is a GPT-style model created by fine-tuning the GPT3 [3] model to generate Python functions. CodeT5 [49] is an encoder-decoder model pre-trained on CodeSearchNet [18]. Recently prompt learning [24, 46] draws a lot of attention. By transforming downstream tasks into fill-in-the-blank problems, code pre-trained models can be directly used for automatic program repair via predicting appropriate code tokens required to fix bugs [36, 41, 52].

7 CONCLUSION

We propose a domain-aware prompt-based approach named TypeFix for repairing Python type errors. TypeFix improves prompt-based approach by incorporating domain-aware fix templates. TypeFix implements a novel fix template design to handle type errors at different levels, and mines fix templates via a novel hierarchical clustering algorithm. TypeFix incorporates domain knowledge into code prompts by applying fix templates into buggy code and invokes code pre-trained models to generate candidate patches from code prompts. Experiments demonstrate the effectiveness of TypeFix and the usefulness of fix templates mined by TypeFix.

8 ACKNOWLEDGMENTS

The authors would like to thank the efforts made by anonymous reviewers. The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund). The work was also supported by National Natural Science Foundation of China under project (No. 62002084), Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), Shenzhen Basic Research (General Project No. JCYJ20220531095214031), and Key Program of Fundamental Research from Shenzhen Science and Technology Innovation Commission (Project No. JCYJ20200109113403826). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the above sponsoring entities.

9 DATA AVAILABILITY

The implementation of TypeFix and experiment results discussed in this paper are available at https://github.com/JohnnyPeng18/TypeFix.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 91–105. https://doi.org/10.1145/3385412.3385997

[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 159:1–159:27. https://doi.org/10.1145/3360585

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[5] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.* 47, 9 (2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[6] Reudismam Rolim de Sousa, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2021. Learning Quick Fixes from Code Repositories. In *35th Brazilian Symposium on Software Engineering, SBES 2021, Joinville, Santa Catarina, Brazil, 27 September 2021 - 1 October 2021*, Cristiano D. Vasconcellos, Karina Girardi Roggia, Vanessa Collere, and Paulo Bousfield (Eds.). ACM, 74–83. https://doi.org/10.1145/3474624.3474650

[7] Dawn Drain, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. 2021. DeepDebug: Fixing Python Bugs Using Stack Traces, Backtranslation, and Code Skeletons. *CoRR* abs/2105.09352 (2021). arXiv:2105.09352 https://arxiv.org/abs/2105.09352

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[9] Python Software Foundation. 2022. Python Abstract Syntax Tree. https://docs.python.org/3/library/ast.html.

[10] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing for C Programs. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 459–470. https://doi.org/10.1109/ICSE.2015.64

[11] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 8–18. https://doi.org/10.1145/3293882.3330558

[12] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Trans. Software Eng.* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[13] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 19–30. https://doi.org/10.1145/3293882.3330559

[14] Github. 2022. The 2022 state of the octoverse. https://octoverse.github.com/2022/top-programming-languages.

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

[16] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: scalable, precise, and safe memory-error repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 271–283. https://doi.org/10.1145/3377811.3380323

[17] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 388–398. https://doi.org/10.1109/ASE.2019.00044

[18] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[19] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[20] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: static analysis-based repair of memory deallocation errors for C. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 95–106. https://doi.org/10.1145/3236024.3236079

[21] Jukka Lehtosalo. 2019. PEP 589 – TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys. https://www.python.org/dev/peps/pep-0589/ https://www.python.org/dev/peps/pep-0589/.

[22] Ivan Levkivskyi, Jukka Lehtosalo, and Łukasz Langa. 2017. PEP 544 – Protocols: Structural subtyping (static duck typing). https://www.python.org/dev/peps/pep-0544/ https://www.python.org/dev/peps/pep-0544/.

[23] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 602–614. https://doi.org/10.1145/3377811.3380345

[24] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *CoRR* abs/2107.13586 (2021). arXiv:2107.13586 https://arxiv.org/abs/2107.13586

[25] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[26] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[27] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 505–509. https://doi.org/10.1109/MSR52588.2021.00063

[28] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 691–701. https://doi.org/10.1145/2884781.2884807

[29] Microsoft. 2023. Copilot. https://github.com/features/copilot.

[30] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2241–2252. https://doi.org/10.1145/3510003.3510124

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

[32] Mypy 2023. Mypy. https://github.com/python/mypy/.

[33] Wonseok Oh and Hakjoo Oh. 2022. PyTER: effective program repair for Python type errors. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 922–934. https://doi.org/10.1145/3540250.3549130

[34] OpenAI. 2023. Codex API Reference. https://platform.openai.com/docs/api-reference/completions/create.

[35] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael R. Lyu. 2022. Static Inference Meets Deep learning: A Hybrid Type Inference Approach for Python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2019–2030. https://doi.org/10.1145/3510003.3510038

[36] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. In *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*. IEEE, 69–75. https://doi.org/10.1145/3524459.3527351

[37] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training.

[38] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 404–415. https://doi.org/10.1109/ICSE.2017.44

[39] 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 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 16–30. https://doi.org/10.1145/3385412.3386005

[40] Salesforce. 2021. The CodeT5-base model. https://huggingface.co/Salesforce/codet5-base.

[41] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. *CoRR* abs/2301.08653 (2023). https://doi.org/10.48550/arXiv.2301.08653 arXiv:2301.08653

[42] Ke Sun, Yifan Zhao, Dan Hao, and Lu Zhang. 2022. Static Type Recommendation for Python. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 98:1–98:13. https://doi.org/10.1145/3551349.3561150

[43] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger (Eds.). 3104–3112. https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html

[44] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP 484 – Type Hints. https://www.python.org/dev/peps/pep-0484/ https://www.python.org/dev/peps/pep-0484/.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[46] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 382–394. https://doi.org/10.1145/3540250.3549113

[47] Wenxuan Wang, Wenxiang Jiao, Yongchang Hao, Xing Wang, Shuming Shi, Zhaopeng Tu, and Michael R. Lyu. 2022. Understanding and Improving Sequence-to-Sequence Pretraining for Neural Machine Translation. In *Annual Meeting of the Association for Computational Linguistics*.

[48] Wenxuan Wang and Zhaopeng Tu. 2020. Rethinking the Value of Transformer Components. In *International Conference on Computational Linguistics*.

[49] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[50] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1556–1560. https://doi.org/10.1145/3368089.3417943

[51] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical Program Repair in the Era of Large Pre-trained Language Models. *CoRR* abs/2210.14179 (2022). https://doi.org/10.48550/arXiv.2210.14179 arXiv:2210.14179

[52] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 959–971. https://doi.org/10.1145/3540250.3549101

[53] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 416–426. https://doi.org/10.1109/ICSE.2017.45

[54] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[55] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2016. Automated memory leak fixing on value-flow slices for C programs. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, Sascha Ossowski (Ed.). ACM, 1386–1393. https://doi.org/10.1145/2851613.2851773

[56] Shuoheng Yang, Yuxin Wang, and Xiaowen Chu. 2020. A Survey of Deep Learning Techniques for Neural Machine Translation. *CoRR* abs/2002.07526 (2020). arXiv:2002.07526 https://arxiv.org/abs/2002.07526

[57] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1506–1518. https://doi.org/10.1145/3510003.3510222

[58] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 341–353. https://doi.org/10.1145/3468264.3468544

[59] Łukasz Langa. 2019. PEP 589 – Type Hinting Generics In Standard Collections. https://www.python.org/dev/peps/pep-0585/ https://www.python.org/dev/peps/pep-0585/.