Tare: Type-Aware Neural Program Repair

Qihao Zhu†, Zeyu Sun ‡, Wenjie Zhang†, Yingfei Xiong*† and Lu Zhang†
†Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University);
School of Computer Science, Peking University, 100871, P. R. China
‡Zhongguancun Laboratory, 100871, P. R. China
{zhuqh,szy_,zhang_wen_jie,xiongyf,zhanglucs}@pku.edu.cn

***Abstract*—Automated program repair (APR) aims to reduce the effort of software development. With the development of deep learning, lots of DL-based APR approaches have been proposed using an encoder-decoder architecture. Despite the promising performance, these models share the same limitation: generating lots of untypable patches. The main reason for this phenomenon is that the existing models do not consider the constraints of code captured by a set of typing rules.**

In this paper, we propose, Tare, a type-aware model for neural program repair to learn the typing rules. To encode an individual typing rule, we introduce three novel components: (1) a novel type of grammars, T-Grammar, that integrates the type information into a standard grammar, (2) a novel representation of code, T-Graph, that integrates the key information needed for type checking an AST, and (3) a novel type-aware neural program repair approach, Tare, that encodes the T-Graph and generates the patches guided by T-Grammar.

The experiment was conducted on three benchmarks, 393 bugs from *Defects4J v1.2*, 444 additional bugs from *Defects4J v2.0*, and 40 bugs from *QuixBugs*. Our results show that Tare repairs 62, 32, and 27 bugs on these benchmarks respectively, and outperforms the existing APR approaches on all benchmarks. Further analysis also shows that Tare tends to generate more compilable patches than the existing DL-based APR approaches with the typing rule information.

***Index Terms*—program repair, neural networks**

I. Introduction

Automated program repair (APR) aims to generate patches automatically and has been extensively studied in the past decade. Due to the large search space of patches and the weak test suites [1], many recent APR tools adopt deep-learning-based (DL-based) approaches: learning from existing patches or source code to generate more probable patches using a neural network [2]–[10]. A newest trend is syntax-guided generation [4], [6]: the space of the patches is defined by a context-free grammar, and the generation of a patch is reduced into a sequence of generation steps, each choosing a grammar rule for expanding a non-terminal; the neural network is used to estimate the probabilities of the grammar rules for each step, and the beam search algorithm [11] is used to greedily select patches with large probabilities. In this way, the generated program is ensured to be syntactically correct. For example, Recoder [6], the state-of-the-art APR approach, builds a grammar for edit operations, which integrates the Java grammar for ensuring syntactic correctness, and generates such operations as patches using a syntax-guided generator.

*Corresponding author.

However, programming languages come with many constraints beyond a context-free grammar. For example, a variable must be used after declaration, the type of an argument should be compatible with the type of a parameter, etc. In modern programming languages, these constraints are uniformly captured by the type system, in the form of a set of typing rules. Since the neural models are not aware of the typing rules, syntax-guided generators may generate many patches leading to untypable code, called *untypable patches*. Though these untypable patches can be easily filtered out by a compiler, having these patches generated inevitably reduces the performance of the APR approaches. On the one hand, the beam search algorithm greedily selects a set of candidates (i.e., partially generated patches) with the largest probabilities at each generation step. A neural network unaware of types may assign higher probabilities to candidates leading to untypable patches, which may exclude the candidate leading to the correct patch from the selected candidates. On the other hand, filtering untypable patches requires time, and fewer typable patches may be validated within the time limit. Based on our experiments with existing syntax-guided generators, the typable rate of the generated patches is only about 30% - 40%, and the correct patch is often excluded from the result set by the untypable candidates.

A direct idea is to filter out the candidates that would not lead to a typable patch immediately at each step, as the existing approach [12]. While this approach is feasible, its effectiveness is limited. For example, let us assume that in a generation step we are generating a statement. Since assignments are very common, a neural network unaware of types may estimate a high probability for the grammar rule `Stmt` $\rightarrow$ `Var` $=$ `Exp`. However, if the local variables and fields are all of Boolean type, it is actually uncommon to assign an expression to a Boolean variable. Filtering cannot solve this problem because assigning to a Boolean variable is still feasible, though being less likely. Thus, it is important for the neural network to learn the typing rules and consider types during the probability estimation.

In this paper, we aim to enable the neural network to learn the typing rules and be aware of types during inference. We make the following assumptions: a computation procedure can be learned by a neural network if (1) the computation procedure is simple, and (2) the training input should not contain too much irrelevant information to overwhelm the neural network. In existing approaches, only faulty programs

and their corresponding patches are used as training set, and to generate typable patches, the neural network has to learn the whole type system from the training set, which is probably too complex for a neural network to learn.

Our insight here is that, while learning the whole type system is too complex, an individual typing rule is often not complex and is eligible to learn. For example, let us consider the following simplified typing rule for assignment.

$$\frac{\Gamma \vdash v : D \quad \Gamma \vdash t : C \quad C <: D}{\Gamma \vdash v = t : \text{Void}}$$

This rule states that, given a variable v whose type is D according to a typing context Γ which records the types of variables from their definitions, an expression t whose type is C, if C is a subtype of D, we know that the assignment $v = t$ is well typed, represented by the type `Void`. To enable a neural network to learn this typing rule, we need to make the neural network aware of the input and output of the typing rule. Basically, there are three relations in the input and the output: (1) a typing relation : between a sub-AST, e.g., an expression or a statement, and a type, (2) the typing context relation Γ between a user-defined element, e.g., a variable, and a type and (3) the subtyping relation $<:$ between types. We need to make the three relations available to the neural network. However, the three relations are huge, and directly encoding them would overwhelm the neural network.

We observe that, to type check a patch, we only need the part of the relations associated with the elements in the input program. For example, if a patch copies to an expression t' to replace t in $v = t$, we only need the type of t', the type of v, and the subtyping relation between the two types. **Our first technical contribution** is a novel graph representation, `T-Graph`, that captures both the AST of a program as well as the part of the three typing relations that are associated with the program. We attach attributes to the nodes in T-Graph to represent the types of the related elements and define several types of edges to represent the typing relations between these elements. Thus, T-Graph represents the key information to enable a neural network to learn the typing rules.

While a complete program can be easily converted into a T-Graph, in the syntax-guided generation [13] we also need to encode the partially generated program for inferring the grammar rule for the next non-terminal. Such partial programs often have ambiguous types and cannot be converted to T-Graph. For example, given a partial program $\texttt{Var} = \texttt{Exp}_1 + \texttt{Exp}_2$, where `Var`, $\texttt{Exp}_1$, $\texttt{Exp}_2$ are non-terminals yet to be further generated, we do not know the types of `Var` because both strings and numeric values can be added in Java. To address this problem, **our second technical contribution** is a novel form of context-free grammars, T-Grammar, that integrates the type information into standard grammars. Basically, we attach a type to each non-terminal symbol to form a set of new symbols and refine the original grammar with a grammar carrying type information. For example, instead of the production rule $\texttt{Exp} \rightarrow \texttt{Exp} + \texttt{Exp}$, T-Grammar has the production rules such as $\texttt{Exp}_{\texttt{Numeric}} \rightarrow \texttt{Exp}_{\texttt{Numeric}} + \texttt{Exp}_{\texttt{Numeric}}$ and $\texttt{Exp}_{\texttt{String}} \rightarrow \texttt{Exp}_{\texttt{String}} + \texttt{Exp}_{\texttt{String}}$, where `Numeric` is a super type for numeric values. In this way, when the neural network predicts a grammar rule, it also predicts its types, enabling the construction of T-Graph from partial programs. To ensure all correct patches can be covered, we ensure that T-Grammar is an upper approximation of the type system, i.e., all typable programs are within the language of T-Grammar.

Based on T-Grammar and T-Graph, **our third technical contribution** is a novel **t**ype-**a**ware neural program **re**pair approach, Tare, for the Java programming language. Tare is built upon Recoder [6], one of the state-of-the-art DL-based APR approaches. We change the grammar in Recoder into a T-Grammar, and replace the neural components of Recoder encoding ASTs with neural components encoding T-Graphs. To encode T-Graph, which is a heterogeneous graph with attributes, we combine two neural layers previously designed for encoding tables [14] and word sequences [15] as a novel neural component.

Our empirical contribution is a set of experiments evaluating the performance of Tare. The experiment is conducted on three widely-used benchmarks in the existing work. There are in total 877 bugs, including 393 bugs from *Defects4J v1.2*, 444 additional bugs from *Defects4J v2.0*, and 40 bugs from *Quixbugs*. Tare successfully repairs 62 bugs on *Defects4J v1.2*, which outperforms all the existing APR approaches. On the other two benchmarks, Tare also achieves best performance over all the existing APR approaches, 32 bugs with 33.3% improvement (8 bugs) on *Defects4J v2.0* and 27 bugs with 42.1% (8 bugs) improvement on *Quixbugs*. Furthermore, we investigate the improvement of Tare on the ranking of the correct patch and compilable rate of generated patches and find that: (1) Tare achieves 44.7% improvement over Recoder on the former metric, and the improvement is higher in complex programs. (2) Tare outperforms the existing DL-based APR approaches in compilable rate with an improvement of 9.3-13.5 percentage points. These results show that Tare has better effectiveness and generalizability than the existing APR tools.

II. Overview

A. Motivating Example

In this section, we present a real-world example to motivate our approach. Figure 1 shows a bug, Cli-25 from the widely-used benchmark *Defects4J*. For better illustration, we simplify the program by renaming some variables. In this example, an assignment statement is incorrect, and the correct patch is to replace the right-hand side with a constant value 1.

There are different ways to implement a DL-based APR for fixing this bug. Let us assume there is a fault localization approach that correctly localizes the faulty line of code. One way is to take the faulty line and the surrounding context (e.g., the faulty method or the faulty file) as input, and train a neural network to produce a fixed line of code as output [2]–[4], [8]. Another way is to make a more refined change: train a neural network to first predict which part needs to be replaced (the right-hand side of the assignment statement in this case), and then predict what new content should be generated [6]. A

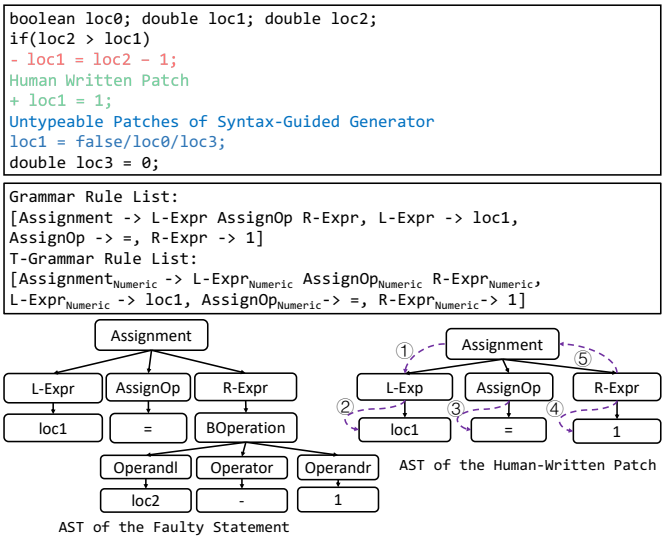

Fig. 1: An Example in Defects4J (Cli-25)

syntax-guided generator [12], [15] repeatedly selects grammar rules to expand non-terminals in an AST to generate code. For example, to generate the fixed line of the human-written patch, a syntax-guided generator starts with the non-terminal `Assignment`, and generates the grammar rule list shown in the central part of Figure 1. Here we assume that the syntax-guided generator always expands the leftmost, lowermost non-terminal first.

To generate the list, a neural network is used to estimate the probabilities of the grammar rules to be used in each step and the beam search algorithm is used to find the rule list with the largest probability. The algorithm keeps a pool of candidate (partial) ASTs up to a predefined size n (called *beam width*). Initially the pool contains one AST with only the root symbol `Assignment`. In each iteration, the algorithm picks the candidate with the highest probability in the pool and returns the candidate if it is complete. Otherwise, the algorithm selects an unexpanded non-terminal in the AST and asks the neural network to estimate the probabilities. The neural network takes the generated partial AST, the position of the non-terminal to be expanded, and the context code as input, and produces the probabilities of the grammar rules to expand the non-terminal as output. For example, in the last step of the generation, the neural network takes the AST of the partial program `loc1 = R-Expr`, the position of `R-Expr`, and the context code as input, and estimates the probabilities of all grammar rules for expanding `R-Expr`. Finally, the algorithm calculates the probabilities of all expanded candidates, and keeps n most probable ones in the pool from these newly expanded candidates and the unselected existing candidates.

When types are not considered, the neural network may incorrectly assign high probabilities to untypable candidates or candidates leading to untypable patches, excluding correct patches from the pool. Figure 1 shows some examples of untypable patches generated, which either replace the right-hand side with literals of incorrect types, or variables that are either undeclared, or of incorrect types.

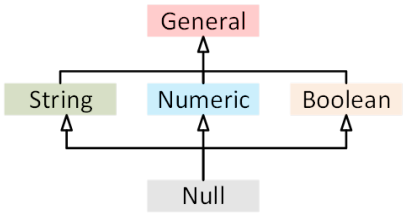

Fig. 2: Subtyping Lattice of the Abstract Types

B. Novel Components of Tare

As mentioned in the introduction, Tare introduces three novel components to guide the neural network to be aware of types. We introduce them one by one.

T-Grammar. As mentioned, T-Grammar refines the original grammar by introducing types. In T-Grammar, a non-terminal has the form N_T, where N is a non-terminal in the original grammar, and T is a type, indicating that N_T generates all sub ASTs of type T generable by N. However, since modern programming languages such as Java have a huge number of types, directly using these types would form too many non-terminals overwhelming the neural network. To avoid this, we use abstract types instead of concrete types of the target programming language. In our current implementation in Java, we use five types as shown in Figure 2, where `General` is the type of everything, `Null` is the type for the special literal `null`, `Numeric`, `String` and `Boolean` are the types of numeric, string, and Boolean expressions, respectively.

We extend the type system of Java programming language to ensure every sub-AST of a typable program has an abstract type. We also convert the original grammar rules into T-Grammar rules such that all typable programs that can be generated by the original grammar rules can still be generated by the T-Grammar rules. For example, the central part of Figure 1 shows a list of T-Grammar rules corresponding to the original grammar rule list. By attaching types, T-Grammar rules also exclude some untypable programs. For example, there is no grammar rule $\texttt{Exp}_{\texttt{General}} \rightarrow \texttt{Exp}_{\texttt{General}} + \texttt{Exp}_{\texttt{General}}$, because $+$ can only be used with either numeric values or string values, and thus this avoids the generation of `1+true` for instance.

By simply replacing the grammar with the corresponding T-Grammar in a syntax-guided generator, we force the generator to produce types for the partial AST, and also prevent some of the untypable programs not in the space of T-Grammar.

T-Graph. Existing syntax-guided generators treat programs either as a list of grammar rules or as a list of tokens. T-Graph represents the program in a graph with important type information attached. As described in Introduction, T-Graph preserves the three typing relations of the input program: (1) the typing relation between a sub-AST and a type, (2) the typing context relation between a user-defined element and a type, and (3) the subtyping relations between types.

Figure 3 shows an example of T-Graph of the faulty statement. In T-Graph, nodes are connected through different types of edges, and may also contain attributes. The difference between attributes and the ID of the node is that the value of the attribute is available to the neural network while ID is only used to distinguish different nodes. Because we use an adjacent matrix to represent the edges, we allow only one type of edges between a pair of two variable nodes.

As shown in the figure, we assign each node in the AST with a type attribute to represent the typing relation. For example, the node $\texttt{Operandr}_{\texttt{Numeric}}$ has an assigned attribute, Numeric. Here the type attribute still uses abstract type but not concrete types because we may need to encode partially generated AST, where the concrete types cannot be inferred. While the abstract type attribute seems to be duplicated with the type attached to the non-terminal, it is still important to have this attribute because (1) the terminals does not have the type annotation, and (2) when the non-terminals are encoded in a neural network through one-hot encoding, the original symbol and the attached type are not distinguishable. This attribute helps preserve the type information.

To represent the typing context relation, we introduce additional nodes (shown in ellipses), named *variable nodes*, for the user-defined elements (e.g. variables and parameters) in the context. For example, there are three variables nodes in the graph, which represent "loc0", "loc1", and "loc2". There is an orange line between the use of a variable and the variable node, and each variable also has an attribute of its type. In this way, we associate each use of a variable with its type. Here we use the concrete types because the variable definitions are obtained from the context but not generated.

Finally let us consider the subtyping relation between types. Since the subtyping relation is mainly used to determine whether the value of type A could be assigned to a variable of type B or not, we introduce three types of edges between variables to capture whether an assignment is possible. The bidirectional *InCompatible* edge indicates that the value of neither variable can be assigned to the other variable. The unidirectional *Compatible* edge indicates the source variable can be assigned to the target variable. The bidirectional *SameType* edge indicates the two variables are of the same type. For example, "loc1" and "loc2" have the *SameType* edge while "loc1" and "loc0" have the *InCompatible* edge. Please note that *Compatible* is not equivalent to subtyping because the autoboxing mechanism in Java allows assignments between two types that do not have the subtyping relation, e.g., from `Integer` to `int`. In other words, having two *Compatible* edges between two variable nodes in different directions is different from having a bidirectional *SameType* edge.

T-Graph Encoder. T-Graph is not the first graph representation of code for neural processing. Multiple existing approaches [16]–[18] have used a graph to represent code and a graph neural network (GNN) to encode the graph. However, T-Graph differs in two aspects compared with the graphs used in existing approaches. (1) The graphs used in existing approaches are homogeneous, in the sense that the edges only have one type. However, a T-Graph is heterogeneous, where the edges have different types. (2) The graphs in existing approaches do not have attributes on nodes, while the nodes in a T-Graph have attributes. GNN supports neither the heterogeneous edges nor the attributes of nodes. Therefore, we have to find a new way to encode the graph.

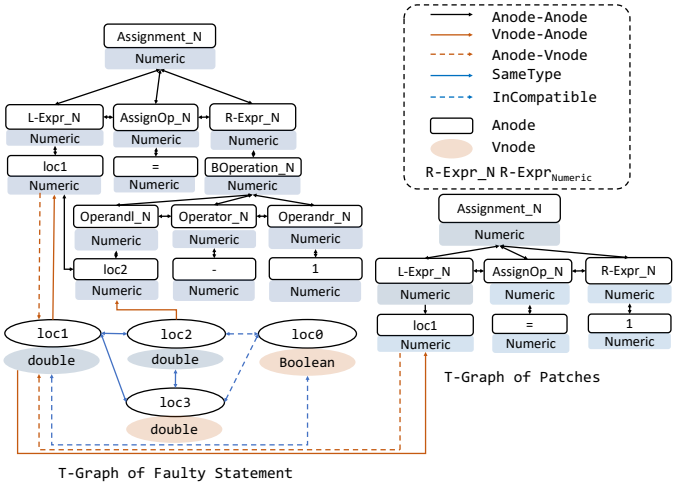

Fig. 3: T-Graph of the Example

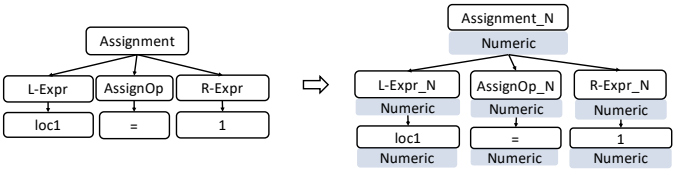

Fig. 4: T-AST of the Faulty Statement in the Motivating Example

To encode a T-Graph, we adapt two neural components from existing approaches on encoding tables [14] and word sequences [15] to process heterogeneous graph and graph with attributes, respectively. First, we adapt the relation-aware attention layer proposed by Wang et al. [14] to encode heterogeneous graphs. Different from the GNNs, this layer computes the node embedding with not only the embedding of the neighbors but also the corresponding edge embedding, and thus different types of edge have different influences. Second, we adapt the gating layer proposed by Sun et al. [15] to combine the attributes of each node. It first converts the attributes into real-value vectors and then integrates these vectors with the corresponding node embeddings. Combining these two layers as well as a standard linear layer, we have an encoder for T-Graph. By replacing the existing encoders for (partial) code in a syntax-guided code generator with the T-Graph encoder, we guide the syntax-guided generators to learn typing-rules and be ware of types during inference.

III. T-GRAMMAR

A. Abstract type system

As mentioned, we use an abstract type system instead of the original concrete type system so as not to overwhelm the neural network. Formally, an abstract type system consists of a set of abstract types, a subtyping relation between abstract types, and a type inference procedure, which, given an AST of a typable program in the original type system, assigns an abstract type to each sub-AST. We have introduced in Section 2 our abstract type system for Java, whose types and the subtyping relation are shown in Figure 2. Below we discuss several implications of this definition.

First, the definition requires that all typable programs in the original type system are still typable in the abstract type system. One way is to make the abstract type system an abstraction of the original type system, by designing a function mapping the original types to the abstract types. Yet more refined types are also possible. For example, in our abstract type system, the `Null` type is a refinement of the original type system, where in Java `null` is typed as other nullable types based on the context.

Second, the definition requires that every sub-AST has a type, even for those does not have a type in the target language. For example, the non-terminal `AssignOp` (appeared in the motivating example) could generate assignment operators such as =, +=, and /=, which do not have a type in Java. A standard way to handle this is to give all such sub-ASTs the type `General`, but more refined types can be assigned for specific cases. In our current abstract type system, we assign type T to an operator if all of its operands have type T. For example, operator += has type `String` in statement `s+="a"`, has type `Numeric` in statement `a+=1`, but would never have type `Boolean` in any statement.

Third, the definition requires that each sub-AST is assigned only one type. In a classic type system, because of the existence of the subtyping relation, an expression usually has multiple types, a minimal type (e.g., `String`) and all its super types (e.g., `Object`). From the perspective of a classic type system, this requirement is equivalent to requiring that all sub-ASTs have a minimal type. It is easy to see that our abstract type system for Java satisfies this property: (1) The `null` literal has the minimal type `Null`. (2) There is no intersection between the types `Numeric`, `Boolean`, and `String` except for the `null` literal, and thus for any sub-AST other than `null` that has one of the three types, the type is also minimal. (3) All other sub ASTs have the minimal type of `General`.

We call an AST with abstract type attached a *T-AST*. Figure 4 shows an example T-AST.

B. T-Grammar and its properties

Based on the abstract type system, we proceed to define T-Grammar. T-Grammar attaches types to the non-terminals in the original grammar, and modifies the grammar rules to (1) include all typable programs, and (2) exclude untypable programs as many as possible. Formally, let $G = (N, \Sigma, R, S)$ a context-free grammar where N is a set of non-terminals, Σ is a set of terminals, R is a set of grammar rules, S is a start symbol. Let T be an abstract type system. A T-Grammar based on G and T is a tuple (N^T, Σ, R^T, S), where N^T and R^T are the smallest sets satisfying the following conditions.

(1) For any nonterminal n in N and any abstract type t in T, we have $n^t \in N^T$.

(2) For any original grammar rule $N \to A^1\ A^2\ \dots\ A^k$ in R, if there exists an application of the original rule in any typable program, where the sub-ASTs corresponding to $N, A^1, A^2, \dots, A^k$ in this application have the abstract types $T, T_1, T_2, \dots, T_k$, respectively, we have that the grammar rule $N_T \to A^1_{T_1}\ A^2_{T_2}\ \dots\ A^k_{T_k}$ in R^T. Here A^i can be a non-terminal or a terminal. We define A^i_T as A^i when A^i is a terminal. For example, there are grammar rules $\texttt{AssignOp}_{\texttt{String}} \to$ += and $\texttt{AssignOp}_{\texttt{Numeric}} \to$ += because += can be typed as `String` or `Numeric` as analyzed above, but there is no grammar rule $\texttt{AssignOp}_{\texttt{Boolean}} \to$ +=.

(3) S is included in N^T. For any abstract type t in T, $S \to S_t$ is included in R^T.

We observe that this T-Grammar includes all typable programs in the original grammar. This is because (1) all the original typable programs are still typable in the abstract type system, (2) the second rule considers all possible applications of a grammar rule in all typable programs, and (3) the third rule ensures to generate the start symbol of any type.

The converted T-Grammar also excludes some untypable programs, as we consider only the minimal abstract types when converting the grammar rules. For example, there is no grammar rule $\texttt{AssignOp}_{\texttt{General}} \to$ /=, as operator /= is only applicable to Numerics.

IV. T-GRAPH

In this section, we will formally introduce the detailed structure of T-Graph. T-Graph can be defined as a tuple $\mathcal{G} = \langle \mathcal{V}, \mathcal{E}, \phi \rangle$, where $\mathcal{V}$ denotes the vertexes in the graph, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ denotes the edges representing the relations and $\phi : \mathcal{E} \to \mathcal{R}$ denotes an edge mapping function where $\mathcal{R}$ denotes the sets of the predefined edge types. We define the T-AST of the faulty code as $\mathcal{G}_{\text{ast}} = \langle \mathcal{V}_{\text{ast}}, \mathcal{E}_{\text{ast}} \rangle$, and the user-defined elements in the context as $\mathcal{V}_{\text{var}}$. We also use the T-Graph of the faulty statement in Figure 3 to better illustrate this process.

A. Node

We first introduce the composition of the nodes in $\mathcal{G}$. The nodes of $\mathcal{G}$ mainly consist of two types of nodes, AST nodes and variable nodes. Formally, the set of the nodes can be defined as $\mathcal{V} = \mathcal{V}_{\text{ast}} \cup \mathcal{V}_{\text{var}}$.

1) AST Node: The first part of the nodes comes from $\mathcal{V}_{\text{ast}}$, the nodes of $\mathcal{G}_{ast}$. We name this type of node as *Anode*. We preserve the name of the nodes in T-AST as the ID in T-Graph. Each Anode has two attributes in our design. First, to represent the type information of each renamed symbol, each Anode also has a type attribute. Second, since several syntax-guided generators [4], [6] allow to copy a subtree of the faulty

TABLE I: Edges in T-Graph

Information	Node x	Node y	Edge Label	Description
Syntax	Anode	Anode	*Parent-Child*	x is the Parent node of y
			Child-Parent	x is the child node of y
			Left-Sibling	x is the left sibling node of y
			Right-Sibling	x is the right sibling node of y
Context	Anode	Vnode	*Declaration-Var*	x is the declaration of y
			Use-Var	x uses variable y
Context	Vnode	Anode	*Var-Declaration*	y is the declaration of x
			Var-Use	y uses variable x
Type	Vnode	Vnode	*Same-Type*	x has the same type as y
			Compatible-Type	x is unidirectional compatible with y
			InCompatible-Type	x is unidirectional incompatible with y

method, we also assign each AST node a Boolean copyable attribute, indicating whether the AST depends on variable in the local context and cannot be copied to other places. As shown in Figure 3, the nodes represented as rectangles are the AST nodes and are connected by several directed edges.

2) Variable Node: The second part of the nodes is variable nodes, $\mathcal{V}_{\text{var}}$, named as *Vnode*. Each Vnode represents a user-defined element (i.e. variable and parameter) in the context. We use the name of the variable as the ID in T-Graph. Each Vnode has two attributes. The first attribute indicates whether this variable can be accessed from the faulty location. The second attribute is the type of the variable. As shown in Figure 3, the nodes shown as ellipses are the variable nodes and represent the local variables in the context. Each node also is attached with a corresponding type attribute.

B. Edges

To represent the relations of the typing rule, we define a set of the predefined edge types, $\mathcal{R}$. Each edge has a type in $\mathcal{R}$ to represent the relation. There are four subsets of edges, Anode-Anode, Vnode-Anode, Anode-Vnode, and Vnode-Vnode. Formally, the set of the edges can be defined as $\mathcal{E} = \mathcal{E}_{\text{A-A}} \cup \mathcal{E}_{\text{A-V}} \cup \mathcal{E}_{\text{V-A}} \cup \mathcal{E}_{\text{V-V}}$. Table I shows the detailed information of these edges.

1) Anode-Anode: The first subset is the original edges in T-AST, which contains grammatical constraints of the code. Since the edges in the tree are directed, we define four edge types to represent these relations. As shown in Table I, they are *Parent-Child*, *Child-Parent*, *Left-Sibling*, and *Right-Sibling*, respectively. The former two denote the depth information while the latter two denote the breath information. Formally, these edges can be defined as $\mathcal{E}^{\text{plus}}_{\text{ast}} = \{\langle v^i_a, v^j_a\rangle | v^i_a \text{ is the left or right sibling of } v^j_a \vee \langle v^i_a, v^j_a\rangle \in \mathcal{E}_{\text{ast}} \vee \langle v^j_a, v^i_a\rangle \in \mathcal{E}_{\text{ast}}\}$. As shown in Figure 3, Anode "L-Expr_N" and Anode "loc1" has the edges (*Parent-Child*, *Child-Parent*) between them in Figure 4.

Except these edges, inspired by existing work [19], [20], we also integrate $\mathcal{G}$ with the data flow graph (DFG) of the program. The graph represents the dependency between the variables, in which nodes represent variables and edges represent where the value of each variable comes from. Such code structure provides crucial code semantic information for the model to extract the constraints in the code. We define the DFG of the program as $\mathcal{G}_{\text{dfg}} = \langle \mathcal{V}_{\text{dfg}}, \mathcal{E}_{\text{dfg}} \rangle$, especially $\mathcal{V}_{\text{dfg}} \subseteq \mathcal{V}_{\text{ast}}$. Since DFG is a directed graph, we define two types of edges for $\mathcal{G}$. If node x has a directed edge pointing to node y, x will have a *Variable-Use* edge connecting to y in $\mathcal{G}$. In the meantime, y will have a *Variable-Def* edge connecting to x. Formally, these edges can be defined as $\mathcal{E}^{\text{plus}}_{\text{dfg}} = \{\langle v^i_a, v^j_a\rangle | \langle v^i_a, v^j_a\rangle \in \mathcal{E}_{\text{dfg}} \vee \langle v^j_a, v^i_a\rangle \in \mathcal{E}_{\text{dfg}}\}$. Finally, the edges between Anodes can be defined as $\mathcal{E}_{\text{A-A}} = \mathcal{E}^{\text{plus}}_{\text{ast}} \cup \mathcal{E}^{\text{plus}}_{\text{dfg}}$. For example, the edges between Anode "loc1" and Anode "loc2" denote that the variable "loc1" use the value of the variable "loc2" after this assignment, which come from the DFG of the faulty statement.

2) Vnode-Anode: This subset of edges represents the relations between AST nodes and variables in the context. We assume that the root of the subtree of the declaration statement of variable y is x. Then, x will have a *Var-Declaration* edge connecting to y in $\mathcal{G}$. Similarly, x has a *Use-Var* connecting to y when the terminal x in T-AST uses variable y. Formally, this subset can defined as $\mathcal{E}_{\text{V-A}} = \{\langle v^i_a, v^j_{\text{var}}\rangle | v^i_a \text{ declares or uses variable } v^j_{var}\}$. As shown in Figure 3, Vnode "loc1" has an edge pointing to Anode "loc1" since the Anode uses the corresponding variable.

3) Anode-Vnode: This subset of edges is the reverse of *Vnode-Anode*. Thus, the set can be defined as $\mathcal{E}_{\text{A-V}} = \{\langle v^i_{var}, v^j_a\rangle | \langle v^i_a, v^j_{\text{var}}\rangle \in \mathcal{E}_{\text{V-A}}\}$. As shown in Figure 3, Anode "loc1" also has an edge pointing to Vnode "loc1".

4) Vnode-Vnode: This type represents the subtyping relations between variables. Specifically, for all variables, we define three types of relations: (1) the variables have the same type, (2) the variables have unidirectional compatible types but not the same type (3) the variables are unidirectional incompatible. Thus, each two variable nodes are connected by one of these edges. As Figure 3 shown, there are 6 edges connecting variables nodes in the graph, which contain the type compatible information between these variables. Formally, the set can be defined as $\mathcal{E}_{\text{V-V}} = \{\langle v^i_{\text{var}}, v^j_{\text{var}}\rangle | \langle v^i_{\text{var}}, v^j_{\text{var}}\rangle \in \mathcal{V}_{\text{var}} \times \mathcal{V}_{\text{var}} \wedge i \neq j\}$.

V. T-Graph Encoder

In this section, we will introduce the detailed structure of the proposed T-Graph encoder, which is shown in Figure 5. As shown, the encoder consists of a relation-aware attention layer and a gating layer to process T-Graph. First, T-Graph will be converted into three parts, node sequence, attribute sequence, and relation matrix, for the encoder during preprocessing. Then, the relation-aware attention layer integrates the node embeddings with the relation matrix to learn the typing relation in the faulty code. Finally, the gating layer incorporates the attribute embeddings with the corresponding node embeddings processed by the first layer. We will first describe the detail of the preprocessing for T-Graph.

Preprocessing. To encode the graph-shaped input, it first needs to be converted into sequences with a linearization method. Here, we adopts Pre-order Traversal [21] linearization for Tare, which significantly shortens the length of the sequences. Under this setting, T-Graph $\mathcal{G}$ is represented as three sequences.

- Node traverse sequence. To encode the node tokens, we first adopt the node traverse sequence to represent the

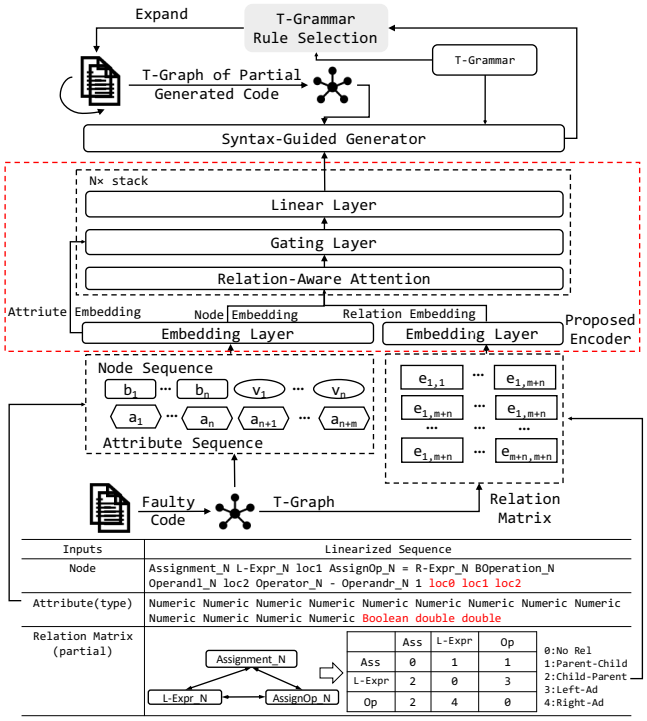

Fig. 5: Neural Model of Tare

sequential information. It consists of two parts. The first part is the pre-order traverse of T-AST and the second part is the sequence of the variables. We assume $\mathcal{G}$ has m AST nodes and n variable nodes. Thus, the length of this sequence is $m+n$.

- Attribute sequence. As described in section IV-A, each node in T-Graph is annotated with several attributes. To integrate this information, we represent these attributes as a sequence of vectors following the order of the node traverse sequence.
- Relation matrix. We represent the edges as the adjacency matrix, $E \in R^{(m+n)\times(m+n)}$. We assign each type of edge listed in Table I with an unique ID in E. For example, $e_{i,j}$ denotes the ID of the edge from the i-th node to the j-th node. Especially, if there are no edge from the i-th node to the j-th node, we set $e_{i,j} = 0$.

Figure 5 shows the input sequences of T-Graph in Figure 3. Given these sequences, we then adopt a relation-aware attention based encoder to process.

Relation Encoding. Inspired by Wang et al. [14], we adopt the relation-aware attention based on the standard self-attention [22] to encode the relation matrix. The relation-aware attention layer first uses multi-head attention to capture the long dependency of the sequence. To integrate the relations between nodes, this layer computes the attention weights based on both node embeddings and edge embedings. For the input node token embeddings, $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_n$, this component outputs a sequence of output vectors with sequential information, $\boldsymbol{z}_1, \boldsymbol{z}_2, \cdots, \boldsymbol{z}_n$. The computation of the h-th head can be represented as

$$
\begin{aligned}
w_{i,j}^{(h)} &= \frac{\boldsymbol{q}_i W_Q^{(h)} (\boldsymbol{k}_j W_K^{(h)} + \boldsymbol{e}_{i,j})^T}{\sqrt{d/H}} \\
\boldsymbol{z}_{i,j}^{(h)} &= \sum_{j=1}^{n} \sigma(w_{i,j}^{(h)})(\boldsymbol{v}_j W_V^{(h)} + \boldsymbol{e}_{i,j})
\end{aligned}
\tag{1}
$$

where $W_Q, W_K, W_V \in R^d$ are the parameters of three full-connected layers, d denotes the embedding size, σ denotes the scoring function (e.g. softmax or hardmax), H is the number of the heads, and the $\boldsymbol{e}_{i,j}$ term denotes the embedding of the corresponding edge. With the layer, the encoder can learn to represent the typing relations via trainable parameters.

Attribute Encoding. To incorporate the attribute information of each node with the node embeddings, we use a Gating Mechanism [15], [23]. First, since each node in T-Graph has multiple attributes, we combine the embeddings of the attributes into a real-valued vector via a full-connected layer. In particular, we use separate layers to process the AST nodes and the variable nodes. Then, as used in the existing work [6], [15], [23], we use the gating layer to incorporate the attribute embeddings with the node embeddings. Here, we use the node embeddings to decide the weights of these two embeddings. The computation of the gating layer in i-th head can be represented as:

$$
\begin{aligned}
\alpha_i^o &= exp(\boldsymbol{q}_i^T \boldsymbol{k}_i^o)/\sqrt{d} \\
\alpha_i^c &= exp(\boldsymbol{q}_i^T \boldsymbol{k}_i^c)/\sqrt{d} \\
\boldsymbol{h}_i &= (\alpha_i^o \cdot \boldsymbol{v}_i^o + \alpha_i^c \cdot \boldsymbol{v}_i^c)/(\alpha_i^o + \alpha_i^c)
\end{aligned}
\tag{2}
$$

where $\boldsymbol{q}_i, \boldsymbol{k}_i^o, \boldsymbol{v}_i^o$ are all computed by a fully-connected layer over the node embeddings and $\boldsymbol{k}_i^c$, $\boldsymbol{v}_i^c$ are computed by another full-connected layer over the attribute embedding. Then, $\boldsymbol{h}_i$ is combined with output of other heads via a full-connected layer.

1) Linear Layer: Finally, following the structure of Transformer [22], we feed the embedding of T-Graph, into two full-connected layers for linear transformation. The component yields the output of the mechanism.

In summary, the encoder has N blocks of these three sub-layers. For the first mechanism, it takes the embeddings of three sequences as input. For the rest of N-1 mechanisms, they take the output of the previous mechanism as input.

VI. Experiment Setup

In this section, we will introduce the setup of the experiment for evaluating the effectiveness of Tare. We currently have implemented Tare in Java programming language.

A. Research Questions

Our experiment aims to answer these research questions:
RQ1: How well does Tare perform compared with the existing APR tools? To answer this question, we compared Tare with the existing APR approaches on the widely used benchmarks, *Defects4J v1.2*, containing 393 bugs of 6 projects.
RQ2: How well does Tare perform on other APR benchmarks? To show the generalizability of Tare, we evaluate

TABLE II: Statistics of Dataset

Project	Version	Bugs	Description
Chart	V1.0	26	A 2D chart library for Java applications.
Closure	V1.0	133	A JavaScript checker and optimizer.
Lang	V1.0	64	A host of helper utilities for the java.lang API.
Math	V1.0	106	Miscellaneous math-related utilities
Time	V1.0	64	Joda-Time is the widely used Java date and time classes.
Mockito	V1.2	38	Most popular Mocking framework for unit tests.
Cli	V2.0	39	A simple API for presenting a Command Line Interface.
JackSonCore	V2.0	26	Core part of Jackson that defines Streaming API.
JacksonDatabind	V2.0	112	General data-binding package for Jackson.
JacksonXml	V2.0	6	Extension for Jackson JSON processor.
Compress	V2.0	47	An API for working with compression and archive formats.
Codec	V2.0	18	Simple encoder and decoders for various formats.
Jsoup	V2.0	93	The Java HTML parser.
JxPath	V2.0	22	A Java-based implementation of XPath 1.0.
Gson	V2.0	18	A Java library used to convert Java Objects into JSON.
Csv	V2.0	16	A simple interface for reading and writing CSV files.
QuixBugs	-	40	A benchmark set based on the Quixey Challenge.

Tare on two additional benchmarks compared with other APR approaches. One is 444 additional bugs from *Defects4J v2.0*. The other one is 40 bugs from *QuixBugs*.

RQ3: Does Tare improve the accuracy of the generated patches? To answer this question, we compare Tare with other APR tools in terms of the ranking of the correct patch on *Defects4J v1.2*.

RQ4: Does Tare improve the compilability of generated patches? To answer this question, we calculate the compilable rate of the patches generated by several DL-based APR tools on *Defects4J v1.2* and *QuixBugs*.

B. Dataset

1) Training Dataset: Tare adopts a neural model to generate the patch and thus needs history program patches to train the parameters. For fair comparison, we used the dataset collected by Zhu et al. [6], which contains 103,585 valid patches in Java. We randomly split the dataset into two parts: 80% for training, 20% for validation following Recoder.

2) Test Dataset: To evaluate the effectiveness of Tare, we conducted an experiment on three benchmarks, including *Defects4J v1.2* [24], additional bugs from *Defects4J v2.0* [24], and *QuixBugs* [25]. *Defects4J v1.2* contains 393 real-world bugs from 6 widely-used open-source projects, which is a commonly used benchmark to evaluate the performance of the APR tools. *Defects4J v2.0* introduces 444 additional bugs from 12 projects. *QuixBugs* contains 40 programs from the Quixey Challenge translated into Java. Each contains a one-line defect, along with passing and failing test cases. Table II shows the details of these benchmarks.

C. Independent Variables

1) Fault Localization: We evaluate Tare under two fault localization settings. In the first setting, we used a spectrum-based algorithm, Ochiai, implemented by GZoltar [26] following the previous approaches [6], [27]–[29]. In the second setting, we give the real faulty location to the APR tools known as perfect fault localization. This setting aims to figure out the real performance of the tools without the influence of fault localization techniques, which is extensively used in existing work [2]–[4], [6], [8].

2) Compared techniques: We compare Tare with several state-of-the-art APR approaches. (1) *Traditional APR tools.* We select 4 commonly-used existing APR tools based on traditional techniques to compare: CapGen [30], TBar [27], SimFix [28], Hanabi [12]. (2) *DL-based APR tools.* With the development of DL techniques, lots of DL-based APR tools have been proposed recently. We consider 5 recent models which have the best performance: CoCoNuT [2], CURE [5], RewardRepair (RRepair) [7], DLFix [4], and Recoder [6]. In particular, the former four models all adopt a token-based decoder structure to generate the patches, while Recoder and DLFix use a syntax-guided decoder. In addition, Recoder correctly repairs the highest number of bugs on Defects4J v1.2 as far as we know. Following the common practice of the existing approaches [6], [27], [28], all the performances of the baselines were collected from the existing papers. Since several tools were only evaluated under one or two settings used in our paper, for each setting, we select the state-of-the-art APR tools (with the best recall or precision) which have been evaluated in the corresponding setting. Furthermore, we consider two additional baselines. The first one is Recoder-F, which directly filters out the untypable candidates with basic type checking during beam search for Recoder. The other one is Recoder-T, which only replace the orginal grammar of Recoder with T-Grammar to show the effectiveness of T-Graph.

3) Patch Validation and Correctness: In our experiment, Tare generates the patches based on the result of the fault localization technique. For each suspicious faulty statement, Tare adopts beam search strategy with size 100 to generate candidate patches. Thus, we generate 100 patches for each suspicious statement based on the score. Due to the running-time limit, we only generate patches for the top-500 suspicious faulty statements given by the fault localization technique. After the patches are generated, we execute the patches with the test suite written by developers until one plausible patch [1] is found. Following the previous work [6], [27], [28], we set a 5-hour running-time limit for Tare.

The plausible patch is considered as correct when it is identical or semantically equivalent to the developer-written patch judged by two authors individually. To alleviate the potential error in this processing, we also publish all generated patches for public judgment (details are in Section XI).

4) Implementation: In our current implementation, we directly adopt the syntax-guided generator, Recoder [6] which is one of the state-of-the-art APR approaches on several benchmarks, as the decoder of Tare. We only replace the neural components for encoding partial code in Recoder with the T-Graph encoder.

5) Hyperparameter: For the hyperparameters of our model, we set the number of the encoder iterations N = 5, i.e., the encoder contains a stack of 5 blocks. The hyperparameters of other neural components were set following the configuration of Recoder. We also applied dropout after each block of

[1] A patch that passes all the test cases.

TABLE III: Comparison without Perfect Fault Localization

Project	Bugs	CapGen	SimFix	TBar	DLFix	Hanabi	Recoder	Recoder-F	Recoder-T	Tare
Chart	26	4/4	4/8	9/14	5/12	3/5	8/14	9/15	8/16	**11/16**
Closure	133	0/0	6/8	8/12	6/10	-/-	13/33	14/36	15/31	**15/29**
Lang	64	5/5	9/13	5/14	5/12	4/4	9/15	9/15	11/23	**13/22**
Math	106	12/16	14/26	18/36	12/28	19/22	15/30	16/31	16/40	**19/42**
Time	26	0/0	1/1	1/3	1/2	2/2	**2/2**	**2/2**	**2/4**	**2/4**
Mockito	38	0/0	0/0	1/2	1/1	-/-	**2/2**	**2/2**	**2/2**	**2/2**
Total	393	21/25	34/56	42/81	30/65	28/33	49/96	52/101	54/116	**62/115**
P(%)	-	84.0	60.7	51.9	46.2	**84.8**	52.5	51.5	46.6	53.9

In the cells, x/y:x denotes the number of correct patches, and y denotes the number of patches that can pass all the test cases.

TABLE IV: Comparison with Perfect Fault Localization

Project	CoCoNuT	CURE	RRepair	Recoder	Recoder-F	Recoder-T	Tare
Chart	7	10	5	10	10	9	**11**
Closure	9	14	12	23	24	**25**	**25**
Lang	7	9	7	9	10	12	**14**
Math	16	19	18	19	20	20	**22**
Time	1	1	1	**3**	**3**	**3**	**3**
Mockito	**4**	**4**	2	2	2	2	2
Total	44	47	45	66	69	71	**77**

Since several state-of-the-art DL-based APR tools only provide the correct patches with perfect localization, we do not list the corresponding plausible patches here.

TABLE V: Comparison on additional benchmarks without Perfect Fault Localization

Project	Bugs	TBar	Recoder	Recoder-F	Recoder-T	RRepair	Tare
Cli	39	1/7	3/3	3/3	4/4	6/-	**5/13**
Clousre	43	0/5	0/7	0/8	0/6	**1/-**	0/5
JacksonDatabind	112	0/0	0/0	0/0	0/0	3/-	0/4
Codec	18	2/6	2/2	3/3	3/5	3/-	**3/7**
Collections	4	0/1	0/0	0/0	0/0	0/-	0/0
Compress	47	1/13	3/9	3/9	**4/12**	0/-	**4/13**
Csv	16	0/2	4/4	4/4	4/4	2/	**5/7**
JacksonCore	26	0/6	0/4	0/5	0/5	1/-	**2/7**
Jsoup	93	3/7	7/13	7/13	7/15	4/-	**10/16**
JxPath	22	0/0	0/4	0/4	0/6	**3/-**	2/10
Gson	18	0/0	0/0	**1/1**	**1/1**	**1/-**	**1/1**
JacksonXml	6	0/0	0	0/0	0/0	0/-	0/1
Total	444	8/50	19/46	21/50	23/58	24/-	**32/84**
Quixbugs	40	-/-	17/17	19/19	19/19	19/-	**27/27**

The publication of RRepair only reports the correct patches of the addtional bugs in Defects4J v2.0. Thus we use '-' here.

the attention layer, where the drop rate is 0.1. The model was optimized by Adam with an initial learning rate lr = 0.0001, and scheduled by the early-stop policy. Furthermore, all experiments are conducted with fixed random seeds to avoid randomness and guarantee reproducibility.

```
-  int j = 4 * n - 1;
+  int j = 4 * n - false;                    Recoder
+  int j = 4 * n - 4;                        Tare
```

Fig. 6: The Patch for Math-80 in Defects4J.

```
return foundDigit; }
+if((hasDecPoint || hasExp)){ +return false;}      Tare
```

Fig. 7: The Patch for Lang-24 in Defects4J.

VII. Experiment Result

A. RQ1: Effectiveness of Tare

1) Performance without perfect localization: Table III shows the performance of Tare without perfect fault localization. As shown, Tare substantially outperforms the compared APR approaches on *Defects4J v1.2*. Overall, Tare successfully repairs 62 bugs, 14.8% (8 bugs) more than the runner-up (Recoder-T). In particular, Tare also achieves an uptick (7.3%) in terms of the precision compared with Recoder-T. These results indicate that Tare successfully takes the advantage of the typing relations to enhance the quality of patches while Recoder-T just implements the T-Grammar. Figure 6 shows a unique bug repaired by Tare. Recoder generates an incompatible variable and excludes the correct patch out of the beam. On the contrary, Tare would not generate the *boolean* literal when expanding the non-terminal with *Numeric* type. Figure 7 also shows a bug repaired by Tare but not Recoder-T. As shown, without the type information of the local variables "hasDecPoint" and "hasExp", Recoder is not able to generate the correct conditional statement. The result denotes the effectiveness of T-Graph of Tare. In addition, we also can observe that Tare has a lower precision than some APRs (Capgen, Hanabi). We assume the reason is that Tare focuses on improving the recall of the current DL-based APR, while they focuses on improving the precision. Furthermore, as shown in the Table II, Tare has a similar precision to other APR tools that repair more than 30 bugs on Defects4J v1.2. We believe such precision is acceptable in practice and the small difference between different approaches is not important because many approaches [31]–[33] have been proposed to handle false positives, and could reduce at least half of the false positives.

Figure 8 shows the detailed analysis of the complementary for Tare and other state-of-the-art APR tools, Recoder, DLFix, TBar, and Simfix. As shown, Tare repairs 9 unique bugs compared with these approaches on *Defects4J v1.2*. This

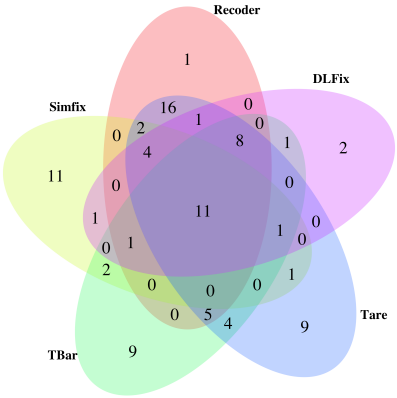

Fig. 8: Degree of Complementary.

TABLE VI: Comparison on Ranking of Correct Patch

Project	Recoder			Tare		
Metric	Max	Min	Avg	Max	Min	Avg
Chart	708	1	172.5	503	1	132.1
Closure	1782	7	545.1	467	18	213.8
Lang	645	2	242.5	338	2	142.3
Math	919	1	251.2	761	1	183.7
Time	1995	846	1420.5	1248	582	915
Mockito	30	16	23	29	3	16
Total	1995	846	368.0	1248	582	203.6

indicates that Tare complements the existing state-of-the-art APR approaches.

2) Performance with perfect localization: Table IV shows the performance of several APR tools on *Defects4J v1.2* with perfect fault localization. As shown, Tare also achieves the best performance, 77 bugs, under this criterion. In particular, we observe that Recoder additionally handles 3 bugs under this setting that were repaired by Tare without perfect fault localization. We assume that Tare gives the correct patch a higher ranking and makes the patch validated before the time limit. This indicates that Tare can somehow alleviate the effect of the fault localization technique.

3) Time Efficiency: We further evaluate the efficiency of our model. It takes 48 mins for an epoch on two Nvidia Titan 3090 with batch size 60 for Tare, whereas 53 mins for Recoder. These two models both are trained for 20 epochs. For inference, Tare takes 90s in average for each faulty location with beam size 100, while Recoder takes 120s.

B. RQ2: Generalizability of Tare

As discovered by Durieux et al. [34], "benchmark overfit" is a common phenomenon for APR tools, especially on *Defects4J v1.2*. To show the generalizability of Tare, we further evaluate Tare on two extra benchmarks, 444 additional bugs from *Defects4J v2.0* and 40 bugs from *Quixbugs*. Furthermore, we also use GZoltar to compute the suspicious score for each line on these benchmarks. Table V presents the results of Tare and other APR tools. We list serveral state-of-the-art baselines that have been evaluated on these benchmarks. From the table, we

TABLE VII: Comparison on Compilable Rate

Model	**Top-30**	**Top-100**	**Top-200**
SequenceR [3]	33%	-	-
CoCoNuT [2]	24%	15%	6%-15%
CURE [5]	39%	28%	14%-28%
RRepair [7]	45.3%	37.5%	33.1%
Recoder [6]	43.5%	36.4%	34.2%
Tare	**54.6%**	**48.6%**	**46.7%**

can observe that Tare outperforms the state-of-the-art APR tools on the benchmarks, with 33.3% (8 bugs) improvement on additional bugs of *Defects4J v2.0* and 42.1% (8 bugs) on *QuixBugs*. These results shed light on the generalizability of Tare. The observation is also expected: Tare is inclined to learn the typing relations defined by the typing rules via T-Graph, which is generally useful for all benchmarks.

C. RQ3: Ranking of Correct Patch

In this RQ, to investigate the reason for the improvement, we calculate the rankings of the correct patch of the bugs repaired both by Recoder and Tare on *Defects4J v1.2*. Since each project has several bugs in Defects4J. Thus, we use the highest, average, and lowest rankings of the correct patches among these bugs of different projects.

The detailed results are presented in Table VI. The results show that Tare almost has better performance than Recoder on all projects for three metrics except the minimum ranking of Closure. For the average rankings, Tare achieves 44.7% improvement over Recoder. Furthermore, we also can observe that Tare outperforms Recoder by 60.8% on Closure, which is the highest among all the projects. We assume the reason is that the code contexts of Closure are more tremendously complex than other projects. Without the type information, Recoder tends to generate more untypable patches.

D. RQ4: Compilable Rate of Patches

Finally, to grasp whether Tare tends to generate more compilable patches, we calculate the compilable rate of the top-k candidates with perfect fault localization, where k denotes the beam size. For sake of fair comparison, we select $k = 30, 100, 200$ in our evaluation following RewardRepair [7] and use the same benchmarks, *Defects4J v1.2* and *Quixbugs*.

Table VII shows the results of the analysis. We directly list the performances reported in the paper [7] for SequenceR, CoCoNuT, CURE, and RewardRepair. For Recoder, we re-run the artifact provided by the authors to calculate the compilable rate. Notably, Tare is inclined to generate more compilable patches than the five DL-based approaches. Overall, Tare achieves 9.3%, 11.1%, 12.5% higher compilable rate than the previous state-of-the-art tools within Top-30, Top-100 and Top-200, respectively. Recall that the key contribution of Recoder is a syntax-guided decoder which integrates the grammatical constraints. This means that Recoder does not embed typing relations knowledge in the neural network. RewardRepair introduces a semantic training approach to

help neural models learn the corresponding knowledge via backpropagation. When the model generates an uncompilable patch, RewardRepair punishes the candidate via decreasing the reward during training. On the contrary, Tare directly encodes the knowledge into the encoder with T-Graph and decodes the patch with constraints of T-Grammar. We can make another observation from the table: with the increases of the beam size, the improvement of Tare increases, too. This also confirms the effectiveness of encoding the typing rules in the encoder directly. Favored by this knowledge, the model estimates higher probabilities for the compilable patches. Thus, the model tends to preserve more typable candidates in the beam with the increasing size.

VIII. Threats to Validity

Threats to internal validity. A threat to internal validity is the potential faults in the implementation of our experiments. To alleviate this threat, we mainly used the reported performance in the paper [2]–[5], [7], [28]–[30], [35]–[37], [37]–[40] for the existing APR approaches. For the ranking and the compilable rate of Recoder [6], we re-run the model provided by the authors. Furthermore, the implementation of Tare is mainly based on two published models [6], [14] to avoid potential errors in re-implementation.

Threats to external validity. A threat to external validity mainly lies in the benchmarks used in our experiment. The training data for Tare is directly derived from the existing work [6]. For testing, we use three widely-used evaluated benchmarks, *Defects4J v1.2*, *Defects4J v2.0*, and *QuixBugs* for Tare. These results show the effectiveness and generalizability of Tare. However, the performance of Tare on other benchmarks [41] is yet unknown. Meanwhile, since our approach was only implemented on Java, further studies are also needed to apply our model to other programming languages.

IX. Related Work

Automated Program Repair. Lots of approaches have been proposed in recent years for automated program repair. Due to the weak test suite and the large search space, various techniques have been used to guide the search processing, including genetic programming [29], manually defined fix patterns [27], [42], [43], automatically mined fix patterns [10], [28], [38], [44]–[49], heuristics [50], learning from code or program synthesis [28], [30], [50]–[53], and semantic analysis [36], [37], [54]–[59].

Apart from these approaches, a more related serie of work [2]–[10] adopts the deep learning models for APR. These approaches mainly use an encoder-decoder architecture and treat the patch generation as natural language translation. SequenceR [3] proposes a sequence-to-sequence NMT to generate the fixed code directly. Different from these approaches, CODIT [8] uses the same model to predict the code edits for the faulty code. DLFix [4], CoCoNuT [2] and Cure [5] take the context of the faulty statement as input and encode it via tree-based LSTM, CNN, GPT, respectively. Recoder [6] proposes a syntax-guided decoder to generate edits with placeholder via the provider/decider architecture.

The recent work, RewardRepair [7] is the most closely related to Tare, focusing on the low compilable rate of the generated patches by DL-based APR approaches. RewardRepair proposes a strategy of semantic training integrating program compilation and test execution information via reinforce learning. Different from RewardRepair, Tare directly learns the typing rules to guide the generation. Thus, Tare also can be combined with semantic training to boost the performance. We leave it as the future work.

Graph Representation for Code. For many software analysis tasks, the graph is the major representation for source code. Learning a good representation for graphs via neural models has been researched in recent years. Inst2Vec [16] proposes to learn the distributed representation of code statements based on contextual data flow. Allamanis et al. [17] uses a Gated Graph Neural Network (GGNN) to embed a program's dataflows for variable renaming and misuses. Lou et al. [18] also adopts the same neural model to encode the coverage-based graph for fault localization. DeepSim [60] and Flow2vec [20] transform the control- and data-flow graph using a GGNN for method naming. Different from these approaches, Tare represents the source code as a heterogeneous graph considering different type relations in code and adopts a relation-aware attention layer to encode such structure.

X. Conclusion

In this paper, we propose Tare, a type-aware neural program repair approach. To integrate the typing relations of an individual typing rule, we represent the faulty code as a heterogeneous graph structure to represent the program. Furthermore, we design a novel grammar, T-Grammar, to combine the typing information into a standard context-free grammar. Generated with T-Grammar, the generator directly predicts the type information of the partial generated programs. Finally, we propose a relation-aware attention-based encoder, T-Graph Encoder, to embed the type information contained in T-Graph. We have conducted an extensive experiment on the widely used benchmarks Defects4J and QuixBugs. The results show that Tare outperforms the existing APR approaches on all benchmarks. The further evaluation of correct patches' ranking and compilable rate suggests that Tare is able to learn the typing relations from the graph and tends to generate more compilable patches.

XI. Data Availability

We have disclosed all the analysis code, generated patches, and data in https://doi.org/10.5281/zenodo.7029404.

ACKNOWLEDGMENTS

This work is sponsored by the National Natural Science Foundation of China under Grant Nos. 62161146003, and a grant from ZTE-PKU Joint Laboratory for Foundation Software.

REFERENCES

[1] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," ser. ISSTA, 2015, pp. 24–36.

[2] T. Lutellier, H. V. Pham, L. Pang, Y. Li, and L. Tan, "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*, 2020.

[3] M. Tufano, C. Watson, G. Bavota, M. di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 832–837.

[4] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 602–614. [Online]. Available: https://doi.org/10.1145/3377811.3380345

[5] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021. [Online]. Available: http://dx.doi.org/10.1109/ICSE43902.2021.00107

[6] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 341–353. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[7] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1506–1518. [Online]. Available: https://doi.org/10.1145/3510003.3510222

[8] S. Chakraborty, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural machine translation," *arXiv preprint arXiv:1810.00314*, 2018.

[9] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 1345–1351.

[10] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781.

[11] C. M. U. C. S. Dept, "Speech understanding systems. summary of results of the five-year research effort at carnegie-mellon university," *cmu*, 1977.

[12] Y. Xiong and B. Wang, "L2s: A framework for synthesizing the most probable program under a specification," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, mar 2022. [Online]. Available: https://doi.org/10.1145/3487570

[13] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, and L. Zhang, "Deep learning based program generation from requirements text: Are we there yet?" *IEEE Trans. Software Eng.*, vol. 48, no. 4, pp. 1268–1289, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3018481

[14] B. Wang, R. Shin, X. Liu, A. Polozov, and M. Richardson, "Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers," in *ACL 2020*, June 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/rat-sql-relation-aware-schema-encoding-and-linking-for-text-to-sql-parsers/

[15] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 8984–8991. [Online]. Available: https://aaai.org/ojs/index.php/AAAI/article/view/6430

[16] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 3589–3601.

[17] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017. [Online]. Available: http://arxiv.org/abs/1711.00740

[18] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 664–676. [Online]. Available: https://doi.org/10.1145/3468264.3468580

[19] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," *ArXiv*, vol. abs/2009.08366, 2021.

[20] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: Value-flow-based precise code embedding," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428301

[21] Z. Tang, X. Shen, C. Li, J. Ge, L. Huang, Z. Zhu, and B. Luo, "Ast-trans: Code summarization with efficient tree-structured attention," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 150–162.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[23] Q. Zhu, Z. Sun, X. Liang, Y. Xiong, and L. Zhang, "Ocor: An overlapping-aware code retriever," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 883–894. [Online]. Available: https://doi.org/10.1145/3324884.3416530

[24] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 437–440.

[25] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge," 10 2017, pp. 55–56.

[26] A. Riboira and R. Abreu, "The gzoltar project: A graphical debugger interface," in *Testing – Practice and Research Techniques*, L. Bottaci and G. Fraser, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 215–218.

[27] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[28] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *International Symposium on Software Testing & Analysis*, 2018, pp. 298–309.

[29] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[30] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 1–11.

[31] X. Liu, M. Zeng, Y. Xiong, L. Zhang, and G. Huang, "Identifying patch correctness in test-based automatic program repair," 06 2017.

[32] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 968–980.

[33] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *Empirical Software Engineering*, vol. 26, no. 2, feb 2021. [Online]. Available: https://doi.org/10.1007%2Fs10664-020-09920-w

[34] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts," in *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*, 2019. [Online]. Available: https://arxiv.org/abs/1905.11973

[35] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[36] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Sketchfix: A tool for automated program repair approach using lazy candidate generation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 888–891.

[37] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.

[38] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, pp. 1–45, 2020.

[39] A. Ghanbari and L. Zhang, "Prapr: Practical program repair via bytecode mutation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1118–1121.

[40] S. Saha *et al.*, "Harnessing evolution for multi-hunk program repair," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 13–24.

[41] Y. Jiang, H. Liu, N. Niu, L. Zhang, and Y. Hu, "Extracting concise bug-fixing patches from human-written patches in version control systems," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 686–698. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00069

[42] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *ASE*. IEEE Press, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155643

[43] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 441–444. [Online]. Available: https://doi.org/10.1145/2931037.2948705

[44] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.

[45] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 255–266.

[46] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[47] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[48] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 404–415.

[49] F. S. Ocariza, Jr., K. Pattabiraman, and A. Mesbah, "Vejovis: Suggesting fixes for javascript faults," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 837–847. [Online]. Available: https://doi.org/10.1145/2568225.2568257

[50] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 660–670.

[51] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.

[52] Y. Xiong, B. Wang, G. Fu, and L. Zang, "Learning to synthesize," in *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*, 2018, pp. 37–44.

[53] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[54] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 448–458.

[55] S. Mechtaev, A. Griggio, A. Cimatti, and A. Roychoudhury, "Symbolic execution with existential second-order constraints," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 389–399.

[56] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–27, 2021.

[57] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 637–647.

[58] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 691–701.

[59] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: Automated synthesis of repair hints," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 266–276.

[60] G. Zhao and J. Huang, "Deepsim: Deep learning code functional similarity," ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 141–151. [Online]. Available: https://doi.org/10.1145/3236024.3236068