TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer

Berkay Berabi [1 2] **Jingxuan He** [1] **Veselin Raychev** [1 2] **Martin Vechev** [1]

Abstract

The problem of fixing errors in programs has attracted substantial interest over the years. The key challenge for building an effective code fixing tool is to capture a wide range of errors and meanwhile maintain high accuracy. In this paper, we address this challenge and present a new learning-based system, called TFix. TFix works directly on program text and phrases the problem of code fixing as a text-to-text task. In turn, this enables it to leverage a powerful Transformer based model pre-trained on natural language and fine-tuned to generate code fixes (via a large, high-quality dataset obtained from GitHub commits). TFix is not specific to a particular programming language or class of defects and, in fact, improved its precision by simultaneously fine-tuning on 52 different error types reported by a popular static analyzer. Our evaluation on a massive dataset of JavaScript programs shows that TFix is practically effective: it is able to synthesize code that fixes the error in ∼67 percent of cases and significantly outperforms existing learning-based approaches.

1. Introduction

The high complexity and large size of modern code repositories have led to a substantial amount of coding errors, a serious obstacle to programmer productivity. While powerful static analysis tools can detect errors, a significant amount of manual effort is still exerted on examining the reports and trying to fix these errors correctly. This situation indicates that a tool capable of automatically fixing real-world coding errors would be highly desirable.

An ideal tool should cover a wide range of errors to benefit most developers and come with high fix accuracy. However, this is difficult to achieve as different types of coding errors, such as variable misuses (Allamanis et al., 2018) and integer type errors (Coker & Hafiz, 2013), have varying root causes. To correctly fix these, it is necessary that the tool is able to capture a range of different program behaviors. Indeed, existing learning-based models either only fix specific kinds of errors (Vasic et al., 2019; Hellendoorn et al., 2020; Pradel & Sen, 2018; Cornu et al., 2015) or become very inaccurate (e.g., 25% accuracy) when extended to handling a more diverse, but still limited, set of errors (Dinella et al., 2020; Chen et al., 2019; Lutellier et al., 2020).

This work: TFix To address this challenge, we present a new learning-based tool, called TFix[1], that can accurately synthesize fixes to a wide range of errors covered by ESLint (esl, 2021), the most popular static analyzer for JavaScript. TFix formulates the problem of fixing coding errors as a text-to-text prediction task. That is, given a coding error as text, TFix predicts a new text representing the code that fixes the error. Such a formulation benefits TFix in three ways: (a) it allows TFix to capture various error types in the same text format, (b) frees TFix from the burden of creating a complicated code representation such as graphs (Allamanis et al., 2018; Dinella et al., 2020), and most importantly, (c) enables TFix to utilize a powerful model called Text-to-Text Transfer Transformer (T5) (Raffel et al., 2020). T5 has been demonstrated to generalize across various natural language problems in the text-to-text format and is thus well-suited for our task.

Leveraging new types of knowledge transfer TFix exploits two kinds of knowledge transfer that are not commonly adopted. First, our T5 model is pre-trained on natural language and then fine-tuned on the programming task of code fixing. This enables knowledge transfer between natural and programming languages (Feng et al., 2020). Further, unlike prior works that train models for each individual error type (Pradel & Sen, 2018; Bader et al., 2019), we fine-tune various error types together (Dinella et al., 2020; Tarlow et al., 2020; Yasunaga & Liang, 2020), which enables knowledge transfer between error types and can be seen as a form of multi-task learning (fixing each error type is an individual task). These two features are key factors for

[1]Department of Computer Science, ETH Zurich, Switzerland [2]Snyk, Switzerland. Correspondence to: Berkay Berabi <berkay.berabi@gmail.com>, Jingxuan He <jingxuan.he@inf.ethz.ch>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

[1]The code, trained model, and dataset can be found at https://github.com/eth-sri/TFix.

TFix to learn a deep understanding of program semantics and thus achieve high accuracy for fixing coding errors.

Fine-tuning with a large high-quality dataset To fine-tune T5, a large model with millions of parameters, we extract a large-scale dataset of ∼100k aligned pairs of coding errors and fixes from 5.5 million GitHub commits. The extraction process invokes the static analyzer and goes through a combination of greedy bipartite matching and Myers diff algorithm (Myers, 1986), resulting in a dataset of significantly higher-quality than existing datasets created by injecting artificial bugs (Vasic et al., 2019; Hellendoorn et al., 2020; Pradel & Sen, 2018) or applying tree transformation rules (Bader et al., 2019; Dinella et al., 2020).

Evaluation of TFix TFix's design is not specific to a particular programming language or error type as it represents code as pure text. We conduct an extensive evaluation of TFix on a massive dataset of JavaScript programs to fix 52 error types detected by ESLint. The results demonstrate that TFix is practically effective: it generates code that fixes the error in 67% of cases. Moreover, we compare TFix to SequenceR (Chen et al., 2019), CoCoNuT (Lutellier et al., 2020), and Hoppity (Dinella et al., 2020), state-of-the-art machine learning tools for code error fixing. TFix significantly outperforms these tools. Our case studies show that TFix can generate correct fixes for complex coding errors.

2. Overview of TFix

In this section, we provide an overview of TFix on a motivating example. Figure 1 shows a JavaScript code snippet that copies properties from one object to another. Besides the desired properties, the code incorrectly copies the properties that belong to the prototype chain (pro, 2020). One way to avoid copying the undesired properties is to call the function `hasOwnProperty`, which ensures that the copied properties are directly in the object instance and are not inherited from the prototype chain.

The defect in Figure 1 can be found by existing error detectors such as ESLint. However, ESLint cannot automatically fix the defect and leaves the task to the developer. Our proposed tool, TFix, can synthesize the snippet in Figure 2 used to replace the corresponding code in the input program to correctly fixes the error. This is achieved by learning from fixes made by human developers in open-source projects. Next, we show step-by-step how TFix fixes this error.

The pipeline of TFix is shown in Figure 3. TFix first sends the input program to ESLint, which identifies a set of errors in the program. For the example in Figure 1, ESLint detects the defect and yields the following information:

```
line 670 error: guard-for-in. error message: the body of
a for-in should be wrapped in an if statement to filter
```

```
if (value != null && fieldData.type != null) {
    var type = null;
    for (var typeEntry in types) {
        var typeNames = types[typeEntry];
        if (typeNames.indexOf(fieldData.type) >= 0) {
            type = typeEntry;
```

Figure 1. An example code snippet with an error

```
var type = null;
for(var typeEntry in types) {
    if(!types.hasOwnProperty(typeEntry)) continue;
    var typeNames = types[typeEnrty];
```

Figure 2. The output of TFix which fixes the error

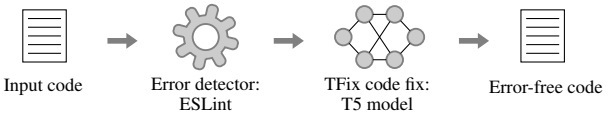

Figure 3. TFix's pipeline for fixing code errors.

`unwanted properties from prototype`.

ESLint locates the error line (highlighted in blue in Figure 1) and outputs the error type and the error message.

The way TFix proposes the correct fix is by first extracting the error context consisting of the error line and the two neighboring lines. Next, TFix represents all information about the error as a single piece of text:

fix error type error message error line : error context

Then TFix uses this text to query a machine learning model called Text-to-Text Transfer Transformer (T5) (Raffel et al., 2020) which generates new text representing the code that fixes the error, as shown in Figure 2. The output text is the same as the error context, except for the inserted if-statement calling `hasOwnProperty`, shown in green. Indeed, the error is fixed after the error context in the original program is replaced by the generated code.

Insights of TFix To generate this non-trivial code, our T5 model has to understand the semantics of the erroneous code and correctly apply `hasOwnProperty`, considering JavaScript syntax and context information such as local variables. Towards that, we leverage a T5 model pre-trained on a large corpus of NLP tasks and then fine-tune it for the specific task of fixing coding errors. This stimulates knowledge transfer between natural language and code: the pre-training drives the model to comprehend natural language fragments in code such as identifier names and then the fine-tuning quickly adapts the model to code fixing.

To enable fine-tuning, we construct a new high-quality dataset consisting of aligned pairs of errors and fixes from 5.5 million GitHub commits (discussed in Section 3.3) –

the largest dataset that captures fixes of coding errors we are aware of. Importantly, fine-tuning is done *jointly* on all 52 error types in the dataset, producing a single model. We experimentally show that this single model outperforms 52 individual models trained for each error type. Indeed, knowledge transfer between error types is a key factor for TFix' ability to generate correct fixes.

Effectiveness of TFix We note that although, in hindsight, the design of TFix may seem simple, it was clearly not the obvious choice. Many recent works apply neural models of code for various tasks (Brody et al., 2020; Allamanis et al., 2020; Wei et al., 2020; Alon et al., 2020) but the practice of leveraging valuable knowledge from a model pre-trained with natural language is not yet common (Feng et al., 2020). Further, most models for code fixing only deal with one specific error and do not benefit from the knowledge transfer between error types (Hellendoorn et al., 2020; Allamanis et al., 2018; Vasic et al., 2019; Pradel & Sen, 2018). Some works (Dinella et al., 2020; Tarlow et al., 2020; Yasunaga & Liang, 2020) implicitly learn a single model for multiple types of errors but do not investigate the effectiveness of this design choice.

We ran SequenceR (Chen et al., 2019), CoCoNuT (Lutellier et al., 2020), and Hoppity (Dinella et al., 2020), three state-of-the-art learning-based tools for code fixing, on the example in Figure 1. None could learn the non-trivial behavior of inserting a new statement and thus failed to generate a correct fix. We also evaluated TFix against the three tools and show that TFix's accuracy is significantly better. Finally, the techniques of TFix are not specific to a particular language or error type and hence open up the possibility of future work for other languages and types of errors.

3. The TFix Approach

In this section, we describe the approach behind TFix, and how it is used to generate code fixes.

3.1. Applying an External Error Detector

The input to TFix is a set of coding errors found by a code analysis tool which we call `detector`. In our case, `detector` parses the input program into an abstract syntax tree (AST), then performs program analysis techniques to detect and report coding errors. Detecting different errors requires various program analyses. As a simple example, finding unused variables requires computing scopes of variables in an AST. TFix is modular as it does not depend on the complex inner logic of `detector` and can directly benefit from advances in bug finding. The inputs to TFix are the error reports made by `detector`. Each report consists of an error type, an error message, and one location.

More formally, given a program p with N lines $[l_i]_{i=1}^{N}$, `detector` identifies a list of M errors $\mathcal{E} = [e_i]_{i=1}^{M}$ in p (if any). The code analyzer `detector` usually supports multiple error types denoted by the set $\mathcal{T}$. Each error e is a tuple $(l_k, \mathcal{L}, t, m)$ where l_k is the k-th line in the program p reported by `detector` for introducing the error, $\mathcal{L} = [l_{k-w}, ..., l_{k-1}, l_k, l_{k+1}, ..., l_{k+w}]$ is the error context, i.e., l_k and the lines surrounding l_k with a window size of w, $t \in \mathcal{T}$ is the error type, and m is an error message. We set $w = 1$ in our evaluation. The input to TFix is the set of errors $\mathcal{E}$. TFix processes each error in $\mathcal{E}$ individually.

Instantiation with ESLint In our work, we instantiate `detector` with ESLint – a static analyzer for JavaScript covering a wide range of errors (esl, 2021). We focused on ESLint due to its popularity: it is adopted by major companies (com, 2021) and has 16 million weekly downloads (dow, 2021). ESLint allows configuring the set $\mathcal{T}$ of returned error types. We used its default configuration, which reports coding errors and best practices, but no formatting issues. ESLint internally includes a manually crafted capability to fix a limited set of stylistic issues concerning whitespaces in code. These whitespace formatting issues are disabled in the default ESLint configuration and were not included in TFix. There exist tools like ESLint for other languages, such as Pylint for Python (pyl, 2021), which can be used when extending TFix to other languages.

3.2. Generating Code Fixes with T5

The goal of TFix is to synthesize a fix for a detected error. We formulate this task as a text-to-text prediction, i.e., given a code error as text, TFix generates text representing new code that has the specific error fixed. Formally, given an error $e = (l_k, \mathcal{L}, t, m)$, TFix represents it as a single piece of text by filling the following template:

$$\texttt{text}(e) = \text{“fix”} \; \textvisiblespace \; t \; \textvisiblespace \; m \; \textvisiblespace \; l_k \; \textvisiblespace \; \text{“:”} \; \textvisiblespace \; \mathcal{L}$$

where “fix” and “:” are raw strings, and $\textvisiblespace$ is a space. Then, TFix queries a text-to-text encoder-decoder model which outputs $\mathcal{L}'$ as text. $\mathcal{L}'$ is used to replace the lines $\mathcal{L}$ in the code to fix the error e (assuming the fix of e lies in $\mathcal{L}$). Note that TFix can be further improved by sampling from the model multiple times until a fix passes `detector`'s check. We leave that as future work.

We leverage the Text-to-Text Transfer Transformer (T5) (Raffel et al., 2020) as our text-to-text model because our formulation is in line with its design. T5 is a generic model that unifies different tasks into the text-to-text format and is pre-trained on NLP tasks. We discuss how to fine-tune T5 for the programming task of generating fixes for code errors in Section 3.3.

To deal with large vocabulary size and out-of-vocabulary

(OOV) tokens, our work represents the input and output code text with Byte Pair Encoding (BPE) (Sennrich et al., 2016), as with (Karampatsis et al., 2020) and the T5 model. Initialized with single characters, the BPE algorithm builds a vocabulary of sub-word tokens by iteratively merging the most frequently co-occurring sub-words. BPE is well-suited for text-to-text tasks and can generalize better to new words unseen at training time than copy mechanism (Chen et al., 2019; Gu et al., 2016) and names splitting based on common naming styles (Lutellier et al., 2020; Allamanis et al., 2018; Alon et al., 2020).

We note that our formulation allows TFix to capture more errors than existing works. Generating an l'_k to replace l_k, as done in (Lutellier et al., 2020; Chen et al., 2019), is not enough for fixing many errors. This is because a correct fix usually involves modifying other lines in the context of l_k. Hoppity (Dinella et al., 2020) suffers from the classic issue of limited vocabulary (Karampatsis et al., 2020). When the desired node value does not exist in the vocabulary, Hoppity can neither express nor generate the desired fix.

3.3. Fine-tuning T5 for Synthesizing Code Fixes

We now discuss our techniques for fine-tuning the pre-trained T5 model for the task of generating code fixes.

Fine-tuning objectives We assume a fine-tuning dataset $\mathcal{D} = \{(e, \mathcal{L}')\}$ consisting of d pairs of error $e = \{l_k, \mathcal{L}, t, m\}$ and its corresponding fix $\mathcal{L}'$ proposed by human developers. The fine-tuning objective is to minimize the cross-entropy loss:

$$L(D) = \sum_{t' \in \mathcal{T}} \sum_{\substack{(e,\mathcal{L}') \in D \\ t = t'}} \log p(\mathcal{L}' \mid \texttt{text}(e)) \qquad (1)$$

The teacher forcing algorithm (Williams & Zipser, 1989) is used during fine-tuning.

Fine-tuning all error types together Note that our loss function in Equation (1) sums over all error types in $\mathcal{T}$, i.e., we fine-tune all error types together, which can also be viewed as a form of multi-task learning that significantly enlarges the dataset and exploits the knowledge transfer between error types. We show in Section 4 that this technique significantly increases TFix's accuracy. A future work item is to fine-tune multiple languages together to obtain a multilingual code fixing model and benefit from the knowledge transfer between languages (Zügner et al., 2021).

3.4. Obtaining a Fine-tuning Dataset

To obtain the dataset $\mathcal{D}$, we analyze a large volume of commits $\mathcal{C} = \{(p, p')\}$ from GitHub where p and p' are the two versions of the program before and after the commit, respectively. A challenge in obtaining a high-quality dataset is to separate error fix commits from the many non-relevant commits that remove an error by completely deleting code or other means, and to extract the parts of code that correspond to errors and fixes in the error fix commits.

Algorithm 1 Procedure for extracting a fine-tuning dataset.

Input : $\mathcal{C} = \{(p, p')\}$, a dataset of GitHub commits.
Output : $\mathcal{D} = \{(e, \mathcal{L}')\}$, a fine-tuning dataset.
1: $\mathcal{D} = \text{emptyset}()$
2: **for** (p, p') **in** $\mathcal{C}$ **do**
3: $\quad \mathcal{E} = \texttt{detector}(p)$; $\mathcal{E}' = \texttt{detector}(p')$
4: $\quad$ **if** $|\mathcal{E}| > |\mathcal{E}'|$ **then**
5: $\quad\quad \mathcal{E}_{\text{fixed}} = \texttt{findFixedErrors}(\mathcal{E}, \mathcal{E}')$
6: $\quad\quad$ **for** e **in** $\mathcal{E}_{\text{fixed}}$ **do**
7: $\quad\quad\quad \mathcal{L}' = \texttt{computeFix}(e, p, p')$
8: $\quad\quad\quad \mathcal{D}.\text{add}((e, \mathcal{L}'))$
9: **return** $\texttt{clean}(\mathcal{D})$

We present the data extraction and cleaning procedure of TFix in Algorithm 1. The algorithm starts with an empty set $\mathcal{D}$ (Line 1), iterates over the input commits $\mathcal{C}$ (Line 2), and runs `detector` to obtain $\mathcal{E}$ and $\mathcal{E}'$ on the pair of files p and p' in each commit. Then it checks if the number of errors in $\mathcal{E}$ is larger than that in $\mathcal{E}'$. If so, it is very likely that p' fixes some errors in p. Therefore, the commit is considered to contain error fixes. We note that we found this criterion for determining an error fix commit to be significantly more accurate in practice than previous approaches based on keywords in the commit message (Lutellier et al., 2020) or the number of tree edits (Dinella et al., 2020) as it leverages the error detector in the decision process.

TFix then calls the `findFixedErrors` function to identify a set of fixed errors $\mathcal{E}_{\text{fixed}} \subseteq \mathcal{E}$ by comparing $\mathcal{E}$ and $\mathcal{E}'$. To achieve this, we leverage a bipartite matching algorithm on the errors in the sets $\mathcal{E}$ and $\mathcal{E}'$. For each fixed error $e \in \mathcal{E}_{\text{fixed}}$, TFix invokes the `computeFix` function to extract the target fix $\mathcal{L}'$ in p' and to obtain a sample $(e, \mathcal{L}')$ to be added to the dataset $\mathcal{D}$ (Line 6 to 8). Finally, `clean` (Line 9) removes noisy samples that contain misaligned error fixes.

In Appendix B, we include an illustrative example of running Algorithm 1. Next we discuss `findFixedErrors`, `computeFix` and `clean` in detail.

Finding fixed errors with bipartite matching The `findFixedErrors` function first invokes a greedy bipartite matching procedure between $\mathcal{E}$ and $\mathcal{E}'$ to identify a set of errors $\mathcal{E}_{\text{unfixed}} \subseteq \mathcal{E}$ that remain unfixed after the commit. To compute the bipartite matching, we iterate all pairs of errors $e = (l_k, \mathcal{L}, t, m) \in \mathcal{E}$ and $e' = (l_{k'}, \mathcal{L}', t', m') \in \mathcal{E}'$, and accept (e, e') as a match if $t = t'$, $m = m'$ and the normalized Levenshtein edit distance between the lines of code at l_k and $l_{k'}$ is small (we used a threshold 0.3). Intuitively,

Table 1. The number of samples and the accuracy on exact match (TFix, T5-large-no-pre-train, and T5-large-per-type) for each error type.

Error type	#Samples	TFix (T5-large)	T5-large-no-pre-train	T5-large-per-type	Error type	#Samples	TFix (T5-large)	T5-large-no-pre-train	T5-large-per-type
no-new-symbol	10	100.0	0.0	0.0	no-extra-bind	684	71.0	21.7	55.1
no-compare-neg-zero	13	0.0	0.0	0.0	no-case-declarations	725	58.9	0.0	47.9
no-ex-assign	40	25.0	0.0	0.0	no-fallthrough	743	76.0	4.0	62.7
for-direction	50	40.0	0.0	40.0	no-inner-declarations	831	38.1	3.6	23.8
no-unsafe-finally	63	42.9	0.0	28.6	no-array-constructor	980	86.7	10.2	56.1
use-isnan	71	37.5	0.0	25.0	no-constant-condition	1284	51.2	12.4	36.4
no-class-assign	111	41.7	0.0	25.0	generator-star-spacing	1396	67.9	23.6	61.4
no-dupe-class-members	117	8.3	0.0	8.3	no-extra-boolean-cast	1458	54.1	2.7	46.6
no-func-assign	147	46.7	0.0	40.0	no-cond-assign	1512	47.4	11.2	33.6
no-empty-pattern	178	27.8	5.6	16.7	no-process-exit	1514	32.9	10.5	20.4
no-unused-labels	187	52.6	5.3	15.8	no-empty	2063	27.1	6.3	15.5
no-duplicate-case	195	65.0	5.0	55.0	no-dupe-keys	2181	53.4	1.8	48.4
getter-return	203	52.4	33.3	47.6	prefer-spread	2496	46.0	12.8	34.4
no-sparse-arrays	237	25.0	0.0	20.8	no-useless-escape	2923	35.2	0.0	11.9
no-const-assign	277	35.7	3.6	17.9	no-console	3067	73.6	4.9	69.4
no-global-assign	318	59.4	9.4	37.5	guard-for-in	3232	41.7	2.8	28.4
no-new-wrappers	360	27.8	2.8	19.4	no-throw-literal	4075	72.1	11.5	68.9
no-this-before-super	413	47.6	9.5	26.2	no-debugger	4164	94.5	15.1	89.4
no-unsafe-negation	423	72.1	7.0	60.5	prefer-rest-params	4582	35.9	8.9	22.0
require-yield	429	72.1	14.0	39.5	no-unreachable	4727	63.8	16.5	58.4
no-extend-native	443	31.1	0.0	13.3	no-extra-semi	5973	82.6	23.6	76.1
no-new-object	446	71.1	6.7	53.3	no-redeclare	6381	49.5	2.3	45.4
no-caller	449	20.0	4.4	20.0	comma-style	6398	46.2	7.0	38.8
constructor-super	464	59.6	10.6	63.8	no-unused-vars	7768	51.9	1.8	47.0
valid-typeof	539	51.9	7.4	37.0	no-undef	10,637	22.4	0.9	16.5
no-self-assign	610	34.4	4.9	37.7	no-invalid-this	16,217	37.7	5.2	25.2
						Sum: 104,804	Avg: 49.3	Avg: 6.7	Avg: 36.3

this means e and e' are very likely to be the same error, so e remains unfixed after the commit. After the iteration finishes, all matched errors in $\mathcal{E}$ form the set $\mathcal{E}_{\text{unfixed}}$. It could happen rarely that an error is matched more than once, in which case we simply discarded the commit. At the end, the set of fixed errors can be obtained by $\mathcal{E}_{\text{fixed}} = \mathcal{E} - \mathcal{E}_{\text{unfixed}}$.

Computing target fix To compute the target fix $\mathcal{L}'$ with `computeFix`, we first leverage the Myers diff algorithm (Myers, 1986) to compute a series of edit operations, which can be used to transform p into p'. Each edit operation inserts or deletes a piece of text (up to one line) from the program. We apply the edit operations and meanwhile track how l_k is shifted in the program. We locate $l_{k'}$ at the final position of l_k after all edit operations are performed. Finally, $\mathcal{L}'$ is obtained by taking the context of $l_{k'}$.

Note that $\mathcal{L}_{k'}$ may contain a newly added line and become badly aligned with $\mathcal{L}_k$ due to the fixed window size. For these cases, we apply a heuristic line-based search in the surrounding of l_k and l'_k. As a result, $\mathcal{L}_k$ and $\mathcal{L}_{k'}$ can be extended by few lines.

Cleaning the dataset To obtain a high-quality development dataset for our T5 model, we perform a cleaning step that removes potential noisy samples from the dataset $\mathcal{D}$. We keep only samples where less than six edit operations are needed to obtain p' from p. Note that since each edit operation can change an entire line, the samples in $\mathcal{D}$ can still be very complicated. The reasoning is that in the filtered samples, the errors are fixed by the commit, but the target fix cannot be confidently computed to obtain a clean training sample. It is intriguing to check if TFix can fix even these errors that have longer or unaligned fixes, so we used the unfiltered errors as an additional test set in our evaluation.

4. Evaluation

In this section, we present our extensive evaluation of TFix and show its effectiveness in fixing real-world errors.

4.1. Experimental Setup

We first present the details of our experimental setup.

Dataset We ran Algorithm 1 on 5.5 million commits obtained from the top 500k GitHub public repositories based on the number of stars and extracted a dataset of fixes for 52 error types detected by ESLint. We found that there were too many samples for some error types, which would make our dataset heavily skewed. Therefore, we performed random downsampling for those error types. Our final dataset consists of 104,804 samples. Detailed statistics are shown in Table 1, and the description for each error type can be found in (esl, 2021). To a create train-test split, we randomly selected 10% of the samples for each error type as the test set (we call it *clean test*). The remaining was further split into 90% for fine-tuning and 10% for validation.

To measure the dataset quality, we manually inspected 100 samples randomly drawn from the whole dataset and checked if they are indeed error fixes. We found that 96 of them are true error fixes and the true error-fix rates lie in the interval $[90.1, 98.9]$ with a confidence level of 95%, meaning that our data construction process yielded a high-quality dataset with a tiny amount of noise.

Besides *clean test*, we test how TFix generalizes to errors where the original fix was complex or noisy and therefore dropped in our data collection procedure. To this end, we assembled another test set called *random test* consisting of all fixable errors in the GitHub commits except for the ones used for fine-tuning and validation. This is done by considering all the errors in $\mathcal{D}$ before Line 9 of Algorithm 1 and excluding the ones used for fine-tuning and validation. *random test* consists of $243,054$ samples.

We made great efforts to remove duplicates in our dataset. For the GitHub repositories, we removed duplicate files that parse to the same abstract syntax tree and further applied a deduplication process on the commit level: for each file, we detected and discarded the commits that exactly repeat the changes in other commits. For instance, if there are commits `A`, `B` and `C`, where `C` performs the changes of `A` and `B` together, we drop `C` even if it might contain changes in other files. We also ensured that there are no identical samples in the final dataset.

Metrics We propose two metrics for measuring the accuracy of TFix. *Exact match* considers a prediction to be correct if and only if the fix perfectly matches the human fix done in the commit. Note that this metric presents a lower bound on TFix's precision as an error can be fixed in multiple ways, and TFix may propose a correct fix different from the human fix. *Error removal* counts a prediction as correct if the error is removed and no new error is introduced after the erroneous code is replaced by the prediction. The average accuracy is computed by averaging the accuracy per error type. Exact match is a more strict metric than error removal as it requires that not only the error is removed but also the fix is the same as the one in the real commit.

We report *exact match* and *error removal* accuracy for *clean test*. For *random test*, we only report *error removal*, as we do not have aligned human fixes for the error inputs.

Hyperparameters and fine-tuning details We chose T5-large as our model for TFix and implemented it with the transformers library (Wolf et al., 2020). In total, the model has 770 million parameters. For details on the architecture, please refer to (Raffel et al., 2020). We initialized the weights with the pre-trained model (t5l, 2021). For fine-tuning, we used Adam (Kingma & Ba, 2015) with the learning rate initialized to 10^{-4}. We set warm-up iterations to 500 and applied linear learning rate scheduling. TFix was fine-tuned on 8 GPUs (GeForce RTX 2080 Ti) with a batch size of 32. The fine-tuning ran for 30 epochs, which took 3-4 days, and applied validation after each epoch. We note that due to constraints on GPU resources, T5-large is the largest T5 model we could run. We leave it as future work item to run TFix with larger T5 models. During inference, we used beam search with a beam size of five.

Table 2. Accuracy of T5 variations on *clean test* and *random test*.

Model	*clean test*		*random test*
	Exact match	Error removal	Error removal
TFix (T5-large)	49.3	67.8	52.5
T5-large-no-pre-train	6.7	48.1	27.4
T5-large-per-type	36.3	52.0	34.2
T5-base	48.5	68.6	52.5
T5-small	39.2	67.7	54.2

Model variants To investigate the effectiveness of our design choices, we investigate the following T5 baselines:

T5-large-no-pre-train: we initialized a T5-large model with no natural language pre-training and trained it from scratch with our dataset.

T5-large-per-type: we performed fine-tuning and testing separately for each error type.

T5-base & *T5-small*: we fine-tuned two smaller T5 models. T5-base and T5-small have 220 million and 60 million parameters, respectively.

Other important conditions (e.g., dataset, optimizer, etc.) were held the same for those models.

4.2. Accuracy on Fixing Coding Errors

We present the accuracy results in Table 2. For *clean test*, TFix achieves 49.3% accuracy on *exact match* (accuracy per error type is shown in Table 1) and 67.8% accuracy on *error removal*. For *random test*, TFix removes the error in 52.5% of the cases. Next, we provide an ablation study of TFix with each model variant.

Effect of pre-training with natural languages The accuracy of the T5-large-no-pre-train model trails that of the pre-trained model. The interesting result for this experiment is the significantly worse result of 6.7% on exact match. This demonstrates that the natural language pre-training is the key for TFix to generate correct fixes, but also points to an insight that relying only on code commits may cause the model to learn how to remove errors in ways a human would not do it. Such problems are reported also in prior

systems (Bader et al., 2019) where the proposed fix is frequently not accepted by a human.

Effect of fine-tuning all error types together We compare TFix and T5-large-per-type to quantify the effect of fine-tuning all error types together. From Table 2, we can see that TFix achieves a significantly (>13%) higher accuracy than T5-large-per-type on all metrics. Table 1 shows that TFix significantly improves upon T5-large-per-type for almost all error types. These results confirm the importance of fine-tuning all error types together and the existence of knowledge transfer between different types of errors. Intuitively, one instance of such knowledge transfer happens when different types of errors target similar program statements. Fine-tuning all error types together increases the size of the samples containing the target statements, helps the model learn better the semantics of those statements, and thus results in higher accuracy.

Effect of model size We compare TFix with T5-base and T5-small to investigate the effect of model size. For exact match on *clean test*, TFix achieves 0.8% higher accuracy than T5-base and 10.1% higher accuracy than T5-small. Therefore, model size is an important factor from T5-small to T5-base but becomes marginal from T5-base to T5-large. For the error removal metric, none of the three models is significantly better than the others. We pick T5-large as the model for TFix mainly because T5-large achieves the highest accuracy on exact match, meaning that it is the best at synthesizing human-like fixes.

4.3. Comparison with State-of-the-art Approaches

We compare TFix with three state-of-the-art approaches: SequenceR (Chen et al., 2019) and CoCoNuT (Lutellier et al., 2020) based on seq2seq learning, and Hoppity (Dinella et al., 2020) based on graph neural networks.

Comparing with SequenceR and CoCoNuT SequenceR is based on an LSTM encoder-decoder with copy mechanism (Gu et al., 2016). CoCoNuT is based on a convolutional encoder-decoder with global attention[2].

In the original papers, SequenceR and CoCoNuT use more restricted datasets requiring that the error and the fix are at the same line. Therefore, we extracted 42,394 samples satisfying this requirement from our dataset and split them in the same way as described in Section 4.1, resulting in a new test set called *restricted test*. We compare their exact match accuracy with TFix on both *restricted test* and *clean test*. For each comparison, all three models were trained on the same dataset. We also gave the same information to all three models, i.e., the input and output to these models were all encoded in the way described in Section 3.2. To be fair and measure the real learnability of the models, we set a reasonable beam size of five to generate one fix per error and did not perform ensemble learning.

[2]We implemented the architecture in OpenNMT (Klein et al., 2017) as the authors confirmed that the released source code currently needs an update and fixing (coc, 2021).

Table 3. Accuracy on exact match for three compared tools.

Tool	*restricted test*	*clean test*
SequenceR	23.6	17.9
CoCoNuT	16.4	11.7
TFix	**46.3**	**49.3**

The results are presented in Table 3 showing that TFix significantly outperforms both SequenceR and CoCoNuT. SequenceR performs much worse than TFix because the LSTM encoder-decoder model is small and could only capture the behaviors of a limited set of errors. CoCoNuT performs even worse than SequenceR. The original paper of CoCoNuT learned ensembles and used a beam size of 1k to generate 20k fixes per error. The error is considered fixed as long as any of the 20k fixes passes the error check. We believe such a large candidate set is more important than the learnability of the model to make CoCoNuT effective.

Comparing with Hoppity Hoppity represents a buggy program as a graph and predicts graph edits with graph neural networks to fix the bugs. We compare TFix with Hoppity showing that TFix significantly outperforms Hoppity in dataset quality, expressivity, and accuracy.

Dataset quality We first measured the noise level of Hoppity's dataset in the same way as we did for our dataset at the start of Section 4.1. We manually inspected 100 samples randomly drawn from Hoppity's OneDiff dataset. We chose the OneDiff dataset as it was the main dataset used in (Dinella et al., 2020). We found that 34 samples were non-bug changes, including version changes, name changes, string changes, etc. The true bug-fix rate of the OneDiff dataset lies in the interval $[55.8, 75.2]$ with a confidence level of 95%, which is significantly lower than ours.

We also checked if our dataset considers more complicated error fixes than Hoppity's, i.e., if more graph edits are needed to represent the samples in our dataset. To achieve this, we converted our dataset to Hoppity's format. During the conversion, we retained the relevant information in our original dataset: we kept the error localization by extracting the smallest AST subtree containing the error context and added the error type as the root of the extracted subtrees. We did not include the error messages because they are long string values not fit in Hoppity's vocabulary (see the next

```
constructor(location, parameterNameAndValues) {
    this.location = location;
    this.parameterNameAndValues = parameterNameAndValues;
```

```
constructor(location, parameterNameAndValues) {
    super(location);
    this.parameterNameAndValues = parameterNameAndValues;
```

Figure 4. TFix fixes a `no-this-before-super` error by converting an assignment into a call to the parent constructor.

```
    break;
case typeof type === 'array' && typeof type[0] === 'object':
    if (isMethod) {
```

```
    break;
case Array.isArray(type) && typeof type[0] === 'object':
    if (isMethod) {
```

Figure 5. TFix fixes a `valid-typeof` error by calling `isArray` function from the `Array` prototype.

```
scout.NumberField.parent.prototype._init.call(this, model);
if (!this.decimalFormat instanceof scout.DecimalFormat) {
    this.decimalFormat = new scout.DecimalFormat(...
```

```
scout.NumberField.parent.prototype._init.call(this, model);
if (!(this.decimalFormat instanceof scout.DecimalFormat)) {
    this.decimalFormat = new scout.DecimalFormat(...
```

Figure 6. TFix fixes a `no-unsafe-negation` error by adding parantheses to change the order of the operations.

paragraph for more details). On average, >8 graph edits were needed to represent our error fixes, while Hoppity's datasets only contain up to 3 edits.

Given that our dataset contains significantly less noise and more complex error fixes than Hoppity's, we used our dataset for quantitatively comparing the expressivity and accuracy of TFix and Hoppity.

Expressivity For some graph edits, Hoppity predicts node values from a vocabulary consisting of values from other nodes and values frequently seen in training. Hoppity's expressivity is limited when the desired value does not exist in the vocabulary, i.e., it can neither express nor predict the desired change (and eventually the desired fix), but puts a special `UNK` token. Out of the 36,361 samples in the OneDiff test set, 20,491 samples involved edits with `UNK`s. In fact, according to the repository (hop, 2020), Hoppity deemed a fix as correct when `UNK`s are predicted for out-of-vocabulary values. Therefore, Hoppity actually considered an easier learning task than generating a complete fix. We followed this for Hoppity in our comparison, giving favor to Hoppity. On the contrary, TFix does not have such a limitation in vocabulary and can express any fix thanks to the Byte Pair Encoding. After converting our dataset to Hoppity's format, 1,300 of the 10,504 test samples had edits with `UNK`s. TFix output correct fixes for 393 of them, while Hoppity output 61 correct fixes. We note that adapting BPE to Hoppity would require non-trivial changes of its learning framework, which we do not consider in our comparison.

Accuracy We trained and tested Hoppity on our converted dataset (using the same split) with the hyperparameters provided in their paper and code. The exact match accuracy of Hoppity was only 7.9% (for generating non-complete fixes with `UNK`s), significantly lower than TFix (49.3% accuracy for predicting a complete fix). This is non-surprising because even for the much simpler OneDiff dataset, Hoppity only had 14.2% Top-1 accuracy. If we consider generating complete fixes for Hoppity, its accuracy would be even lower. Moreover, Hoppity's accuracy drops significantly with an increasing number of required graph edits due to the larger search space for the correct edit sequence: for 1 to 10 edits, its accuracy was 47.2%, 26.1%, 7.6%, 9.7%, 15.6%, 4.8%, 10.6%, 0.7%, 1.4%, 0.3%. For >10 edits, its accuracy was 0%. TFix does not have such a limitation with a text-to-text format.

4.4. Case Studies

We present three case studies to show how TFix fixes coding errors. More cases can be found in Appendix A. The first case is shown in Figure 4. ESLint finds a `no-this-before-super` error in the code on the left-hand side because the programmer tried to access a `location` field of the child object without having initialized its parent class. TFix generates a correct fix on the right, calling the constructor of the parent class where the `location` field is initialized with the `location` variable.

The second case is shown in Figure 5 and is about an incorrect application of JavaScript's `typeof` operator captured by the `valid-typeof` error type. The programmer intended to check if the variable `type` is an array via the `typeof` operator. However, for arrays, the `typeof` operator just returns `'object'`, making the check inconclusive. TFix successfully synthesizes a fix that calls the correct function `isArrary` from the `Array` prototype to perform the check.

We show the final case in Figure 6. It is about a very common bug related to operator precedence. With the if statement, the developer intended to check that `this.decimalFormat` is not an instance of `scout.DecimalFormat`. However, the negation is actually performed on `this.decimalFormat` instead of the result of the `instanceof` operation, resulting in a `no-unsafe-negation` error. As a result, the if condition is always false. TFix adds parentheses to ensure the operators are applied in the correct order.

These case studies show that TFix can generate correct fixes

even in the presence of non-trivial JavaScript semantics (e.g., inheritance) and can produce complex changes such as converting an assignment to a function call.

5. Related Work

We now discuss works more closely related to ours.

Automated program repair Automated program repair (APR) has been a classic research topic in software engineering (Gazzola et al., 2019; Monperrus, 2018). Traditional APR approaches require a specification of correctness, usually in the form of a test suite or logic assertions. Given a program violating the specification, APR approaches output a program satisfying the specification. The output can be obtained by applying edit patterns on abstract syntax trees (Rolim et al., 2018; Hua et al., 2018) or solving symbolic constraints (Mechtaev et al., 2016; Nguyen et al., 2013; Xuan et al., 2017). Our work is significantly different from this line of research. APR works typically suffer from overfitting issues (Ye et al., 2021) as the specification is specific to individual programs and is often incomplete (Qi et al., 2015; Smith et al., 2015). Learning-based APR techniques (Long & Rinard, 2016) are often limited to small and manually curated datasets (Sobreira et al., 2018) and do not lead to significant improvements in fix rates. On the contrary, TFix can generalize across programs by learning from a large dataset. Moreover, APR approaches require a significant amount of effort on modeling program semantics while TFix simply represents code as text.

A notable APR work with machine learning that was hyped as a useful internal tool at Facebook (get, 2018) is GetaFix (Bader et al., 2019). GetaFix is similar to TFix in the way it learns from commits removing static analyzer warnings. While neither the tool nor the training data of GetaFix are available, their training data is much smaller, no learning is done across bugs, and no natural language pre-training is taken. Based on the amount of training data and their low accuracy for relatively simple bugs, we believe that TFix is much more performant and useful in practice.

Learning-based code error fixing Recent years witnessed an increasing interest in applying learning-based methods for detecting and fixing bugs. Graph neural networks (Allamanis et al., 2018), LSTM (Vasic et al., 2019), and Transformer based models (Hellendoorn et al., 2020) have been used on the task of detecting and fixing VarMisuse errors. DeepBugs detects three specific types of bugs for JavaScript by learning code embeddings and a binary classifier for each bug type (Pradel & Sen, 2018). Unfortunately, they are fundamentally non-competitive to static analysis tools for bug detection in terms of both accuracy and popularity. The main reason is that their models are trained with artificially injected bugs that cannot capture real bug distribution. Hoppity (Dinella et al., 2020) learns graph changes to detect and fix bugs. We showed that TFix is superior to Hoppity in Section 4.3.

Another line of research focuses on fixing compilation errors for introductory programming assignments. DeepFix (Gupta et al., 2017) and sk_p (Pu et al., 2016) leverage a RNN model to translate programs with compilation errors to compiled ones. Reinforcement learning (Gupta et al., 2019) and dynamic program embeddings (Wang et al., 2018) are used to improve the fix rate. DrRepair (Yasunaga & Liang, 2020) utilizes a graph neural network and a self-supervised learning paradigm. Instead of compilation errors, TFix fixes bugs coming from a static analyzer. It is an interesting future research item to extend TFix to fix compilation errors where accurate error localization is often unavailable.

Neural models of code Apart from the models for detecting and fixing coding errors, neural models have been proposed for other tasks, such as code editing (Brody et al., 2020; Yin et al., 2019), type inference (Allamanis et al., 2020; Wei et al., 2020), and code completion (Alon et al., 2020; Brockschmidt et al., 2019). Several works focus on learning general purpose code embeddings (Alon et al., 2019; Sui et al., 2020; Wang & Su, 2020). All of these models require complicated code representations, e.g., program trees or graphs, and none benefit from pre-training on natural language. Natural language fragments are used in (Kanade et al., 2020; Feng et al., 2020) to train general-purpose code embeddings for code. The authors of (Zügner et al., 2021) learn a multilingual representation for code. One possible future research item is to find out how the above models of code can be used in code error fixing.

6. Conclusion

We presented a new learning-based system, called TFix, for automatically fixing coding errors. The key idea behind TFix is to formulate the problem of synthesizing fixes for code errors as a text-to-text prediction. This enables TFix to leverage T5, a Transformer model pre-trained on NLP tasks and to fine-tune it for the task of generating code fixes, on a high-quality dataset of errors and fixes that we extracted from millions of GitHub commits. The accuracy of our T5 model was boosted further by considering various error types together during the fine-tuning process. Our extensive evaluation shows that TFix is practically effective: among the fixes generated by TFix, about half perfectly match human fixes, and about two-thirds remove the original error.

We believe our work is an important step in transfer learning for code. It opens up new directions, e.g., directly applying large pre-trained models from NLP to programming tasks.

References

Getafix: How facebook tools learn to fix bugs automatically, 2018. URL https://engineering.fb.com/2018/11/06/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically/.

Github – hoppity, 2020. URL https://github.com/AI-nstein/hoppity.

Copying objects in javascript, 2020. URL https://www.digitalocean.com/community/tutorials/copying-objects-in-javascript/.

Github issues – coconut, 2021. URL https://github.com/lin-tan/CoCoNut-Artifact/issues/3.

Who's using eslint?, 2021. URL https://eslint.org/users.

eslint – npm, 2021. URL https://www.npmjs.com/package/eslint.

List of rules supported by ESLint, 2021. URL https://eslint.org/docs/rules/.

Pylint – code analysis for python, 2021. URL https://www.pylint.org/.

T5-large pre-trained model, 2021. URL https://github.com/google-research/text-to-text-transfer-transformer.

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In *ICLR*, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

Allamanis, M., Barr, E. T., Ducousso, S., and Gao, Z. Typilus: neural type hints. In *PLDI*, 2020. URL https://doi.org/10.1145/3385412.3385997.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019. URL https://doi.org/10.1145/3290353.

Alon, U., Sadaka, R., Levy, O., and Yahav, E. Structural language models of code. In *ICML*, 2020. URL http://proceedings.mlr.press/v119/alon20a.html.

Bader, J., Scott, A., Pradel, M., and Chandra, S. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA):159:1–159:27, 2019. URL https://doi.org/10.1145/3360585.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. In *ICLR*, 2019. URL https://openreview.net/forum?id=Bke4KsA5FX.

Brody, S., Alon, U., and Yahav, E. A structural model for contextual code changes. *Proc. ACM Program. Lang.*, 4 (OOPSLA):215:1–215:28, 2020. URL https://doi.org/10.1145/3428283.

Chen, Z., Kommrusch, S. J., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

Coker, Z. and Hafiz, M. Program transformations to fix C integers. In *ICSE*, 2013. URL https://doi.org/10.1109/ICSE.2013.6606625.

Cornu, B., Durieux, T., Seinturier, L., and Monperrus, M. Npefix: Automatic runtime repair of null pointer exceptions in java. *CoRR*, abs/1512.07423, 2015. URL http://arxiv.org/abs/1512.07423.

Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *ICLR*, 2020. URL https://openreview.net/forum?id=SJeqs6EFvB.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages. In *EMNLP Findings*, 2020. URL https://doi.org/10.18653/v1/2020.findings-emnlp.139.

Gazzola, L., Micucci, D., and Mariani, L. Automatic software repair: A survey. *IEEE Trans. Software Eng.*, 45(1): 34–67, 2019. URL https://doi.org/10.1109/TSE.2017.2755013.

Gu, J., Lu, Z., Li, H., and Li, V. O. K. Incorporating copying mechanism in sequence-to-sequence learning. In *ACL*, 2016. URL https://doi.org/10.18653/v1/p16-1154.

Gupta, R., Pal, S., Kanade, A., and Shevade, S. K. Deepfix: Fixing common C language errors by deep learning. In *AAAI*, 2017. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603.

Gupta, R., Kanade, A., and Shevade, S. K. Deep reinforcement learning for programming language correction. In *AAAI*, 2019. URL https://doi.org/10.1609/aaai.v33i01.3301930.

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global relational models of source code. In *ICLR*, 2020. URL https://openreview.net/forum?id=B1lnbRNtwr.

Hua, J., Zhang, M., Wang, K., and Khurshid, S. Towards practical program repair with on-demand candidate generation. In *ICSE*, 2018. URL https://doi.org/10.1145/3180155.3180245.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. In *ICML*, 2020. URL http://proceedings.mlr.press/v119/kanade20a.html.

Karampatsis, R., Babii, H., Robbes, R., Sutton, C., and Janes, A. Big code != big vocabulary: open-vocabulary models for source code. In *ICSE*, 2020. URL https://doi.org/10.1145/3377811.3380342.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2015. URL http://arxiv.org/abs/1412.6980.

Klein, G., Kim, Y., Deng, Y., Senellart, J., and Rush, A. M. Opennmt: Open-source toolkit for neural machine translation. In *ACL System Demonstrations*, 2017. URL https://doi.org/10.18653/v1/P17-4012.

Long, F. and Rinard, M. Automatic patch generation by learning correct code. In *POPL*, 2016. URL https://doi.org/10.1145/2837614.2837617.

Lutellier, T., Pham, H. V., Pang, L., Li, Y., Wei, M., and Tan, L. Coconut: combining context-aware neural translation models using ensemble for program repair. In *ISSTA*, 2020. URL https://doi.org/10.1145/3395363.3397369.

Mechtaev, S., Yi, J., and Roychoudhury, A. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, 2016. URL https://doi.org/10.1145/2884781.2884807.

Monperrus, M. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, 2018. URL https://doi.org/10.1145/3105906.

Myers, E. W. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986. URL https://doi.org/10.1007/BF01840446.

Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. Semfix: program repair via semantic analysis. In *ICSE*, 2013. URL https://doi.org/10.1109/ICSE.2013.6606623.

Pradel, M. and Sen, K. Deepbugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25, 2018. URL https://doi.org/10.1145/3276517.

Pu, Y., Narasimhan, K., Solar-Lezama, A., and Barzilay, R. sk_p: a neural program corrector for moocs. In *Software for Humanity, SPLASH 2016*, 2016. URL https://doi.org/10.1145/2984043.2989222.

Qi, Z., Long, F., Achour, S., and Rinard, M. C. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, 2015. URL https://doi.org/10.1145/2771783.2771791.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *JMLR*, 21:140:1–140:67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Rolim, R., Soares, G., Gheyi, R., and D'Antoni, L. Learning quick fixes from code repositories. *CoRR*, abs/1803.03806, 2018. URL http://arxiv.org/abs/1803.03806.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. In *ACL*, 2016. URL https://doi.org/10.18653/v1/p16-1162.

Smith, E. K., Barr, E. T., Goues, C. L., and Brun, Y. Is the cure worse than the disease? overfitting in automated program repair. In *FSE*, 2015. URL https://doi.org/10.1145/2786805.2786825.

Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., and de Almeida Maia, M. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *SANER*, 2018. URL https://doi.org/10.1109/SANER.2018.8330203.

Sui, Y., Cheng, X., Zhang, G., and Wang, H. Flow2vec: value-flow-based precise code embedding. *Proc. ACM Program. Lang.*, 4(OOPSLA):233:1–233:27, 2020. URL https://doi.org/10.1145/3428301.

Tarlow, D., Moitra, S., Rice, A., Chen, Z., Manzagol, P., Sutton, C., and Aftandilian, E. Learning to fix build errors with graph2diff neural networks. In *ICSE Workshops*, 2020. URL https://doi.org/10.1145/3387940.3392181.

Vasic, M., Kanade, A., Maniatis, P., Bieber, D., and Singh, R. Neural program repair by jointly learning to localize and repair. In *ICLR*, 2019. URL https://openreview.net/forum?id=ByloJ20qtm.

Wang, K. and Su, Z. Blended, precise semantic program embeddings. In *PLDI*, 2020. URL https://doi.org/10.1145/3385412.3385999.

Wang, K., Singh, R., and Su, Z. Dynamic neural program embeddings for program repair. In *ICLR*, 2018. URL https://openreview.net/forum?id=BJuWrGW0Z.

Wei, J., Goyal, M., Durrett, G., and Dillig, I. Lambdanet: Probabilistic type inference using graph neural networks. In *ICLR*, 2020. URL https://openreview.net/forum?id=Hkx6hANtwH.

Williams, R. J. and Zipser, D. A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280, 1989. URL https://doi.org/10.1162/neco.1989.1.2.270.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *EMNLP Demos*, 2020. URL https://doi.org/10.18653/v1/2020.emnlp-demos.6.

Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S. R. L., Durieux, T., Berre, D. L., and Monperrus, M. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Software Eng.*, 43:34–55, 2017. URL https://doi.org/10.1109/TSE.2016.2560811.

Yasunaga, M. and Liang, P. Graph-based, self-supervised program repair from diagnostic feedback. In *ICML*, 2020. URL http://proceedings.mlr.press/v119/yasunaga20a.html.

Ye, H., Martinez, M., Durieux, T., and Monperrus, M. A comprehensive study of automatic program repair on the quixbugs benchmark. *J. Syst. Softw.*, 171:110825, 2021. URL https://doi.org/10.1016/j.jss.2020.110825.

Yin, P., Neubig, G., Allamanis, M., Brockschmidt, M., and Gaunt, A. L. Learning to represent edits. In *ICLR*, 2019. URL https://openreview.net/forum?id=BJl6AjC5F7.

Zügner, D., Kirschstein, T., Catasta, M., Leskovec, J., and Günnemann, S. Language-agnostic representation learning of source code from structure and context. In *ICLR*, 2021. URL https://openreview.net/forum?id=Xh5eMZVONGF.