electronics

MDPI

Article

Exploring the Potential of Pre-Trained Language Models of Code for Automated Program Repair

Sichong Hao, Xianjun Shi and Hongwei Liu *

Faculty of Computing, Harbin Institute of Technology, Harbin 150001, China; schao@stu.hit.edu.cn (S.H.); shixianjun@hit.edu.cn (X.S.)
* Correspondence: liuhw@hit.edu.cn

Abstract: In the realm of software development, automated program repair (APR) emerges as a pivotal technique, autonomously debugging faulty code to boost productivity. Despite the notable advancements of large pre-trained language models of code (PLMCs) in code generation, their efficacy in complex tasks like APR remains suboptimal. This limitation is attributed to the generic development of PLMCs, whose specialized potential for APR is yet be to fully explored. In this paper, we propose a novel approach designed to enhance PLMCs' APR performance through source code augmentation and curriculum learning. Our approach employs code augmentation operators to generate a spectrum of syntactically varied yet semantically congruent bug-fixing programs, thus enriching the dataset's diversity. Furthermore, we design a curriculum learning strategy, enabling PLMCs to develop a deep understanding of program semantics from these enriched code variants, thereby refining their APR fine-tuning prowess. We apply our approach across different PLMCs and systematically evaluate it on three benchmarks: BFP-small, BFP-medium, and Defects4J. The experimental results show that our approach outperforms both original models and existing baseline methods, demonstrating the promising future of adapting PLMCs for code debugging in practice.

Keywords: large language models of code; automated program repair; code debugging

Citation: Hao, S.; Shi, X.; Liu, H. Exploring the Potential of Pre-Trained Language Models of Code for Automated Program Repair. *Electronics* **2024**, *13*, 1200. https://doi.org/10.3390/electronics13071200

Academic Editor: Josep Silva

Received: 9 February 2024
Revised: 18 March 2024
Accepted: 22 March 2024
Published: 25 March 2024

1. Introduction

In an era where digital technologies permeate every aspect of our daily lives, the ubiquity of software programs and systems has inadvertently led to an increase in software bugs, posing significant challenges to software maintenance and reliability. Automated program repair (APR) [1] has emerged as a pivotal innovation, streamlining the process of software maintenance by automating the generation of patches to rectify software defects, thereby reducing the reliance on labor-intensive manual debugging efforts. The advent of artificial intelligence (AI)-driven approaches has marked a paradigm shift in how we address the challenge of program repair, attracting substantial research interest and investment in recent years. Among these techniques, large pre-trained language models of code (PLMCs) [2–5] stand out for their exceptional code generation capabilities, heralding a new era of AI-powered software engineering. These general-purpose PLMCs, initially pre-trained on extensive corpora of unlabeled data through self-supervised learning paradigms, demonstrate adaptability to specialized tasks like program repair with minimal adjustments during fine-tuning. The success of language models in natural language processing (NLP) [6–8] has laid a robust foundation for the application of PLMCs in APR, suggesting a promising avenue for harnessing the power of advanced AI to enhance software development and reliability efficiency.

Although applying PLMCs to the realm of program repair has shown to surpass existing APR methodologies [9], their efficacy, as measured by success rates, remains suboptimal. As shown in [10], the state-of-the-art PLMC CodeT5 [5] achieves an average bug-fixing success rate of 18.3%, while the best non-PLMC-based APR techniques [11] is

12.7%. As such, the quest to enhance the success rates of PLMCs in program repair tasks is a formidable challenge.

While the integration of PLMCs into the APR domain holds considerable promise, the exploration of the most efficient methodologies for leveraging these models to optimize APR outcomes remains scant. The process of fine-tuning emerges as a pivotal strategy, enabling the adaptation of PLMCs to APR tasks by leveraging specialized datasets tailored for bug fixing. Notably, fine-tuning presents a significantly reduced computational cost compared to the exhaustive demands of training a language model from the ground up [12]. It has been documented that such targeted fine-tuning can amplify the efficacy of PLMCs in APR endeavors by upwards of 30% [13]. Consequently, the quest to improve fine-tuning methodologies, to bolster the APR capabilities of PLMCs, presents a pressing challenge that warrants thorough exploration.

Within the realm of APR, the practice of fine-tuning PLMCs is currently beset by several formidable challenges. Firstly, the foundational pre-training methodologies employed by PLMCs are predominantly derived from the domain of NLP. This transposition is complicated by the differences between the syntax and semantics of programming languages versus natural languages. Programming languages permit the expression of semantically equivalent constructs through a wide array of syntactic variations, a flexibility that natural languages often do not afford. This discrepancy leads to PLMCs' diminished sensitivity to the intricate interplay between code syntax and semantics, subsequently impeding their efficiency in diagnosing and debugging erroneous code [14]. Secondly, the fine-tuning of PLMCs typically relies on a corpus of historical bug-fix records, which are pairs of buggy and corrected code snippets, collected via meticulously designed heuristics. This approach, however, is hampered by the inherent limitations in the scope and depth of available bug-fix datasets. In contrast to the abundant reservoir of open-source code available for training, the datasets comprising bug fixes are not only scarce, but also of variable quality [15]. This scarcity and variability constrain the potential of PLMCs in code debugging.

To effectively navigate the aforementioned challenges, we propose a general fine-tuning framework designed to both enhance the training dataset's diversity for bug fixes and facilitate a more profound semantic comprehension by PLMCs during the fine-tuning phase. Our approach encompasses three pivotal components: a code augmentation operator, a difficulty measurement module, and a dataloader scheduler, working in concert to refine the training process. The framework employs an array of ten distinct code augmentation operators to generate a multitude of syntactically varied yet semantically congruent program variants. It simulates the varied methodologies software developers might employ to achieve identical functional outcomes, thereby broadening the spectrum of bug-fixing scenarios presented in the model. This methodological innovation not only expands the diversity of the bug-fixing dataset, but also presents PLMCs with a richer set of syntactic variations to learn from, thereby enhancing their adaptability and efficacy in addressing a broader array of program repair tasks.

To assimilate the nuanced code semantic insights from the diverse syntactic representations of bug-fixing code variants, we propose an advanced curriculum learning mechanism. This mechanism, inspired by pedagogical principles akin to human learning methodologies, systematically organizes the training process into a structured "curriculum" for PLMCs. Drawing upon the foundational concepts established by Bengio et al. [16], curriculum learning's efficacy in educational settings is adapted to the domain of APR, treating bug-fixing code snippets as "learning materials" for PLMCs. This "bug-fixing curriculum" is meticulously crafted to optimize the fine-tuning phase, enhancing the PLMCs' proficiency in APR tasks. Central to our approach is the integration of a difficulty measurer and a dataloader scheduler, which collectively evaluate and sequence the bug-fixing code samples based on their complexity. Contrary to the conventional random sampling approach, our methodology prioritizes a progressive learning path, presenting PLMCs with simpler tasks initially and gradually introducing more complex challenges. This strategic arrangement

mirrors the curriculum-based learning approach and is poised to significantly improve the PLMCs' learning efficiency and, consequently, their performance in code debugging.

While our approach can be integrated with various PLMCs, we implement it with two representative PLMCs, GraphCodeBERT [4] and CodeT5 [5]. We conduct extensive experiments on three benchmarks with different APR scenarios, including bug-fix commits (BFP_{small} and BFP_{medium} [17]) and real bugs with test cases in open source project (Defect4J [18]). The experimental results demonstrate that, without manual adjustments to the model architectures, PLMCs implemented with our approach outperform both original models and existing baseline methods in generating accurate patches over the baseline metrics. To delve deeper into the efficacy of our approach, we conducted a series of ablation studies to evaluate the contributions of its core components. Additionally, our exploration into various dataloader scheduler configurations further elucidates the adaptability and robustness of our approach, reinforcing its practical utility in enhancing the APR process. Additionally, we further explore the effectiveness of different types of code augmentation operators. The comprehensive analysis and subsequent findings attest to the efficiency and effectiveness of our approach, paving the way for its broader adoption in the field of automated program repair.

To sum up, we make the following contributions:

- We introduce an innovative approach to fine-tuning PLMCs specifically for APR tasks, diverging from traditional methodologies that typically employ a randomized order for training examples. By strategically developing a bug-fixing curriculum, we sequence the training process to progressively transition from simpler to more complex examples.
- We propose a comprehensive curricular fine-tuning framework designed to elevate the APR success rates of PLMCs. This framework incorporates a suite of ten distinct code augmentation operators to enrich the diversity within the bug-fixing dataset. Furthermore, it features an APR-specific difficulty measurer to evaluate the complexity of bug-fixing code samples, alongside an effective dataloader scheduler that optimizes the order of code data presented during training.
- We apply our approach to different PLMCs, and our extensive experiments demonstrate that our approach substantially improves the performance of PLMCs in program repair without manual intervention in model architecture design or excessive time cost. Further analysis proves the generalizability of our approach in fixing real-world bugs.

This work is an extension of our ICSME 2023 paper [19]. Compared to the preliminary version, we explore the effects of different code augmentation operators in Section 3, and the generalizability in addressing real bugs of Defect4J in Section 4. Additional experimental results and discussions of code augmentation operators and the generalizability of our approach are included in Section 5. We also expand the evaluation to investigate the effectiveness of our approach compared with other APR methods in Section 5.

2. Background and Related Work

2.1. Pre-Trained Language Models of Code

The advent of large pre-trained language models has significantly enhanced the capabilities across a broad spectrum of NLP tasks. This monumental success has catalyzed a wave of innovative endeavors aimed at tailoring these pre-training methodologies to the realm of programming languages, leading to the development of numerous PLMCs. Analogous to the categorization of NLP models into encoder-only (e.g., BERT [7]), decoder-only (e.g., GPT [20]), and encoder–decoder architectures (e.g., T5 [8]), PLMCs also follow a similar architectural classification. Models like GraphCodeBERT utilize a bidirectional Transformer encoder, leveraging multiple layers of self-attention to capture the nuanced vector representations of code sequences. On the other hand, decoder-only models, such as Codex [21], employ a Transformer decoder architecture, sequentially generating code by processing all preceding tokens. Meanwhile, encoder–decoder models like CodeT5 [5] jointly train bidirectional encoders and autoregressive decoders for comprehensive model-

ing of source code. The encoder is used to obtain the vectorized embedding of input data, while the decoder is used to generate output code.

It is noteworthy that, despite the architectural variances among these models and their distinct pre-training paradigms, the essence of fine-tuning for APR tasks remains universally applicable. In this paper, we propose a general framework that capitalizes on fine-tuning existing PLMCs to amplify their APR success rates. While our methodology is adaptable to a wide array of PLMCs, we validate its efficacy through empirical studies on GraphCodeBERT and CodeT5. These two models are selected based on the following reasons: (i) they are publicly available. Codex is not included since it is not open-source, and thus cannot be fine-tuned; and (ii) they have proven track records in APR, underscored by their high citation rates and authoritative performance. Through this focused examination, we aim to underscore the applicability and effectiveness of our fine-tuning framework across different PLMC architectures.

2.2. Language Model-Based Automated Program Repair

For APR, researchers have harnessed various language model-based strategies, broadly categorized into universal and specific language model-based APR frameworks.

Universal language model-based APR methods [3–5,22–24] strive to develop versatile PLMCs that cater to a range of coding tasks, including program repair. Given these models are not tailor-made for APR, they necessitate fine-tuning with APR-centric datasets. Our technique offers an enhancement to the fine-tuning efficacy of PLMCs, serving as a valuable supplement to these existing methods.

Specific language model-based APR approaches concentrate exclusively on program repair, utilizing language models to forge innovative APR methodologies. Yuan et al. [25] introduce CIRCLE, an APR method grounded in T5 that progressively learns bug resolution across various programming languages. Jiang et al. [26] pre-train a GPT model on an extensive codebase and amalgamate it with NMT architecture. Distinct from these techniques, our method sidesteps the need for custom language model architecture or additional pre-training phases. It instead focuses on enhancing model performance through the strategic processing of training data in the fine-tuning phase. As a result, our approach can be seamlessly and effectively implemented across different PLMCs, irrespective of their pre-training goals.

To enhance the performance of program repair, researchers have adopted diverse methodologies by refining code data [26–28]. Tufano et al. [17] introduced a code abstraction technique to reduce vocabulary size and ensure models focus on discerning common patterns across various code modifications in bug-fixing instances. SequenceR [29] incorporates class-level context, abstracting the buggy context from the class to infer potential fixes. Lutellier et al. [30] preprocess the buggy code and its contextual surroundings separately, subsequently inputting these sequences into a neural machine translation (NMT) framework. Chakraborty et al. [31] developed a multi-modal NMT-based tool that crafts patches by processing diverse information modalities, including edit locations, code context, and commit messages. Xia et al. [15] manipulate buggy code with mask tokens and encode this alongside the surrounding context for model input. Ye et al. [32] create training samples with bugs via a perturbation model, utilizing these artificially bugged codes as inputs for a Transformer neural network. Our approach diverges from these methods by generating code variants from abstracted bug-fixing pairs, preserving both the semantic integrity and syntactic authenticity of the code. We then fine-tune PLMCs on this preprocessed data in a strategic sequence to improve their APR success rates.

2.3. Curriculum Learning

Curriculum learning is a training paradigm that trains a model with easier samples first and then gradually extends to more complex samples. This methodology mirrors the pedagogical approach of incrementally increasing the complexity of learning materials, starting with a foundational, simplified subset of the training data. As the model's training advances,

it progressively encounters a broader spectrum of examples, introducing higher levels of challenge that simulate the escalating difficulty found in human educational curricula.

Curriculum learning has garnered validation in cognitive science [33,34], and has been adapted for training models in the realm of artificial intelligence. Bengio et al. [16] pioneered the curriculum learning strategy, showcasing its utility in supervised tasks within visual and linguistic domains. This innovation has spurred a wide array of scholars to integrate curriculum learning across diverse model frameworks and applications, including computer vision [35] and natural language processing [36]. Hacohen et al. [37] undertook a comprehensive study on employing curriculum learning for image classification using the convolutional neural networks framework. Platanios et al. [38] introduced a competence-based curriculum learning approach for neural machine translation, confirming its applicability across both recurrent neural network and Transformer architectures. They developed competence functions to regulate the number of training instances accessible to the model, which has influenced the design of our dataloader scheduler. Penha et al. [39] explored curriculum strategies for enhancing conversation response ranking in information retrieval, demonstrating the efficacy of this approach with the BERT model. Wang et al. [40] advanced code comprehension accuracy in code understanding tasks by applying curriculum learning to categorize transformed data, further enriched with test-time augmentation techniques. Different from these studies, we conducted an exploration of applying curriculum learning to automated program repair. We introduce an innovative method for categorizing code samples by difficulty, predicated on code length, to exploit the nuances of bug-fixing data. The process begins with shorter bug-fixing code pairs, progressively incorporating longer samples.

The major challenges of curriculum learning are how to define the relative difficulty of training examples and how to schedule the sequence of data subsets during the training process. Addressing this, we propose a curriculum learning-based framework consisting of a difficulty measurer and a dataloader scheduler. This framework is engineered to optimize the learning trajectory of PLMCs, particularly in the context of code debugging, ensuring they achieve optimal performance in program repair tasks.

3. Approach

In this section, we introduce our approach. We commence with a comprehensive overview of the framework's architecture, laying the groundwork for a deeper exploration. Subsequently, we introduce each constituent element of our approach, encompassing the bug-fixing code augmentation mechanism, the difficulty measurement module, and the dataloader scheduler, providing a detailed exposition of their functionalities and interplay within the framework.

3.1. Architecture Overview

The architecture of our approach is illustrated in Figure 1. We start with a set of buggy code, referred to as B and its corrected version, F (i.e., (B, F)). The initial step in our approach involves leveraging a suite of bug-fix code augmentation operators. These operators are designed to produce a variety of syntactically distinct yet semantically consistent versions of both B and F correspondingly. In the subsequent phase, we design an APR-based difficulty measurer to decide the relative difficulty of the original training samples as well as their corresponding variants. Lastly, we organize the training data using the dataloader scheduler, which arranges the code examples in a way that gradually increases in complexity, making it easier for PLMCs to digest and learn from the expanded set of examples.

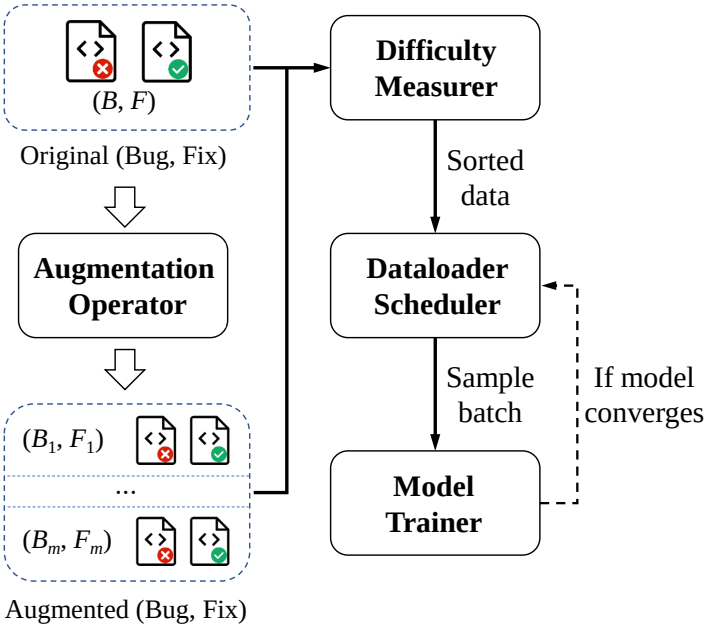

Figure 1. An overview architecture of our approach. The key novel features are the augmentation operator, the difficulty measurer, and the dataloader scheduler.

3.2. Bug-Fixing Code Augmentation

The initial phase of our curriculum learning-oriented framework focuses on preparing high-quality "learning materials". This process is designed to acquaint PLMCs with a series of diverse code features, derived from a variety of bug-fixing programs that, despite their structural differences, maintain functional parity. Consequently, it is essential to generate multiple iterations of each bug-fix pair that preserve the underlying semantics without alteration. Manual collection of such diverse examples is impractical, particularly when dealing with extensive code datasets. Therefore, we employ an automated approach, utilizing a series of code augmentation operators. These operators are meticulously crafted to ensure that the augmented code samples retain their original semantics while exhibiting syntactic naturalness, thereby facilitating an effective learning environment for PLMCs.

In this study, we concentrate on augmenting Java bug-fixing code, due to Java's prominence as the most extensively studied language in prior APR research [41]. There are many methods for augmenting the code [42,43], and the more augmented methods we use, in principle, the more code data we can obtain. To guarantee the efficacy of our augmentation approach specifically for bug-fix pairs, we implement ten distinct code augmentation operators, each designed to enhance the diversity and richness of our dataset. Further, these ten operators can be categorically divided into three types based on the scope and nature of the modifications they introduce:

- Control structure transformations:
  `Change If Statement`: it changes a single if statement into a conditional expression statement.
  `Change Conditional Expression`: it changes a conditional expression statement into a single if statement.
  `Change If Else`: it switches two code blocks in the if statement and the corresponding else statement.
  `Change While to For`: it replaces a while statement with a semantic-equivalent for statement.
- API transformations:
  `Switch Equals Method`: it switches the two string arguments on both sides of the `String.equals()` function.
  `Change Increment Operator`: it changes the assignment `x++` into `x+=1`.

`Change Add Assignment Operator`: it changes the addition assignment operator to assignment operator, i.e., `x+=1` becomes `x=x+1`.

- Declaration transformations:
  `Merge Declaration`: it merges the declaration statements into a single compound declaration statement.
  `Rearrange Statement Order`: it swaps the positions of two adjacent statements within a code block, provided that the first statement does not share any variables with the second.
  `Switch Equality Operator`: it swaps the positions of two expressions on either side of the equality operator in an infix expression.

The first group focuses on alterations to the control structures, the second group pertains to modifications in API usage, and the third group involves changes in variable declarations. Each group is designed to aid the models in identifying and assimilating pertinent features from the code, all the while meticulously preserving the original semantics of the code snippets. The ten augmentation operators we utilize adhere to three fundamental principles: (i) they generate variants for both the buggy and fixed code, maintaining semantic consistency while introducing diverse syntactic structures; (ii) for every bug-fix pair, the syntactic alterations applied to the buggy and fixed code are identical to ensure uniformity; and (iii) all applied transformations are correct, preserving the original semantics of the bug-fix pairs without introducing any inaccuracies. As illustrated in Figure 2, the `Change If Statement` augmentation operator exemplifies this approach by converting a standard if statement in the original bug-fix pair (as depicted in Figure 2a,b) into a conditional expression in the augmented pair (shown in Figure 2c,d). Given that the core semantics of the buggy and corrected code remain intact post-augmentation, the augmented code pairs maintain functional equivalence with their original counterparts, albeit with altered syntactic configurations. By employing ten such operators, we can enrich the bug-fixing dataset with a wide array of syntactically varied yet semantically identical programs. This enhancement allows models to gain a deeper comprehension of different syntactic forms, and to grasp the relationships between syntax and semantics.

```
private void METHOD_1() {
    TYPE_1 VAR_1 = VAR_2.METHOD_2();
    final int count = VAR_1.METHOD_3();
    for (int i = 0; i < count; i++) {
        TYPE_2 VAR_3 = METHOD_4(i);

        java.lang.CharSequence title = VAR_1.METHOD_5(i);
        if (title == null) title = STRING_1;
        VAR_3.setText(title);

    }
    METHOD_6();
}
```

(a) Original Buggy Code

```
private void METHOD_1() {
    TYPE_1 VAR_1 = VAR_2.METHOD_2();
    final int count = VAR_1.METHOD_3();
    for (int i = 0; i < count; i++) {
        TYPE_2 VAR_3 = METHOD_4(i);
        if (VAR_3 != null) {
            java.lang.CharSequence title = VAR_1.METHOD_5(i);
            if (title == null) title = STRING_1;
            VAR_3.setText(title);
        }
    }
    METHOD_6();
}
```

(b) Original Fixed Code

```
private void METHOD_1() {
    TYPE_1 VAR_1 = VAR_2.METHOD_2();
    final int count = VAR_1.METHOD_3();
    for (int i = 0; i < count; i++) {
        TYPE_2 VAR_3 = METHOD_4(i);

        java.lang.CharSequence title = VAR_1.METHOD_5(i);
        title = (title == null) ? STRING_1 : title;
        VAR_3.setText(title);

    }
    METHOD_6();
}
```

(c) Augmented Buggy Code

```
private void METHOD_1() {
    TYPE_1 VAR_1 = VAR_2.METHOD_2();
    final int count = VAR_1.METHOD_3();
    for (int i = 0; i < count; i++) {
        TYPE_2 VAR_3 = METHOD_4(i);
        if (VAR_3 != null) {
            java.lang.CharSequence title = VAR_1.METHOD_5(i);
            title = (title == null) ? STRING_1 : title;
            VAR_3.setText(title);
        }
    }
    METHOD_6();
}
```

(d) Augmented Fixed Code

Figure 2. An example illustrating bug-fixing code augmentation. (**a**) shows a buggy code snippet and (**b**) shows the corresponding fixed code (with the fixed part highlighted in blue), which are both from the Bugs2Fix medium dataset. (**c**,**d**) show the results of applying `Change If Statement` augmentation operator on (**a**,**b**), respectively (the changes are highlighted in red and green backgrounds). They are both augmented by changing a single if statement into a conditional expression statement.

3.3. Difficulty Measurer

Numerous criteria exist for gauging the complexity involved in rectifying a piece of buggy code. In our approach, we draw upon two heuristic principles influenced by elements known to affect the efficacy of PLMCs. These guiding principles are crafted to provide a nuanced understanding of what constitutes 'difficulty' in the context of code repair, taking into account the intricacies that challenge PLMCs.

3.3.1. The Edits of Code

In Section 3.2, we outline a collection of augmentation operators tailored for bug-fixing code datasets, designed to enhance the variety of training examples. However, the code variations produced by these operators might introduce subtle disturbances to the baseline data during the editing process. However, the code variants produced by these operators might introduce minor perturbations to the original data during the editing of code snippets, which could be viewed as a form of adversarial attack against the models [44–47]. As shown in [48], PLMCs are particularly sensitive to these semantics-preserving modifications, which could potentially reduce the performance of the models. The studies by Wang et al. [40] and Liu et al. [49] have further demonstrated that data augmented with semantics-preserving edits pose greater learning challenges for PLMCs than the original data. In light of this, our difficulty measurement incorporates the presence of semantics-preserving edits in the original bug-fixing code data as a key criterion. In our framework, the unaltered dataset is deemed the "easy" scenario, while the dataset enriched with augmentation operators is classified as the "hard" scenario.

3.3.2. The Length of Code

Building on the findings of Tufano et al. [17], which highlight the impact of input data length on model training and efficacy, we set out to assess the influence of code length on the learning challenges faced by PLMCs when tackling bug-fixing tasks. To explore this concept empirically, we designed an experiment, outlined in Algorithm 1, aimed at discerning the effects of code length variations on model outcomes.

The essence of Algorithm 1 lies in its exploration of the relationship between code length diversity and model performance metrics. This exploration involves subjecting the model to a series of test datasets characterized by differing code lengths and scrutinizing the resultant performance disparities. Our methodology involves categorizing the bug-fix pair dataset according to the length of the buggy code snippets. Subsequently, each category is divided into training (80%), validation (10%), and test (10%) subsets in a randomized manner (lines 5–14). Following the model's training phase, which encompasses all length categories within the training and validation sets, we proceed to assess its proficiency on the test subsets, each distinguished by a unique code token count (lines 15–20).

We applied Algorithm 1 to the cutting-edge PLMC, CodeT5 [5], utilizing the Bugs2Fix benchmark dataset [17], which comprises code methods capped at a maximum length of 100 tokens. The entire dataset's training set served as our input. To achieve a balance between the number of categories and the number of bug-fix pairs within each category, we designated N as 4 and K as 10,000. The outcomes depicted in Figure 3 highlight CodeT5's performance across various code length categories. These results reflect the relative challenge posed by each group, with superior model performance correlating with lower difficulty levels, and the reverse being true for diminished performance. As illustrated in Figure 3, the model's performance in program repair declines with increasing code length. This trend underscores the significant influence that code length variability exerts on the model's ability to program repair, with shorter snippets typically being easier to fix than lengthier ones. This observation aligns with our hypothesis that fixes within longer code fragments are often more intricate, entailing a broader array of context variables, identifiers, and literals that challenge the model's interpretative capabilities. This phenomenon has also been observed previously by Chen et al. [29]. Consequently, we

categorize code pairs with shorter lengths as "easy" samples and those with longer lengths as "hard" samples.

Algorithm 1 Code length validation algorithm

Input: Dataset $P = \{(B, F)\}$ consists of bug B and corresponding fix F pairs, model M, count N and K
Output: Experimental results Φ

$\Gamma \leftarrow \{\}$, a set of training data
$\Theta \leftarrow \{\}$, a set of validation data
$\Lambda \leftarrow \{\}$, a set of test data
$\Phi \leftarrow \{\}$, a set of experimental results
Sort P in ascending order based on the code length of B and denote the sorted dataset as P'
Find the maximum length of samples in B and denote it as max_len
$R \leftarrow max\_len / N$
for $i \leftarrow 1 \ \ldots \ N$ **do**
    Randomly pick K pairs of (B, F) from P' and denote the picked set as P_i, where the length of B falls within the range of $((i-1) \cdot R, i \cdot R]$
    Divide P_i into three sets Γ_i, Θ_i, and Λ_i in an 8:1:1 ratio
    $\Gamma \leftarrow \Gamma \cup \Gamma_i$
    $\Theta \leftarrow \Theta \cup \Theta_i$
    $\Lambda \leftarrow \Lambda \cup \Lambda_i$
end for
Train model M with Γ and Θ
for Λ_i in Λ **do**
    Calculate results on M and denote the outcome as Φ_i
    $\Phi \leftarrow \Phi \cup \Phi_i$
end for
return Φ

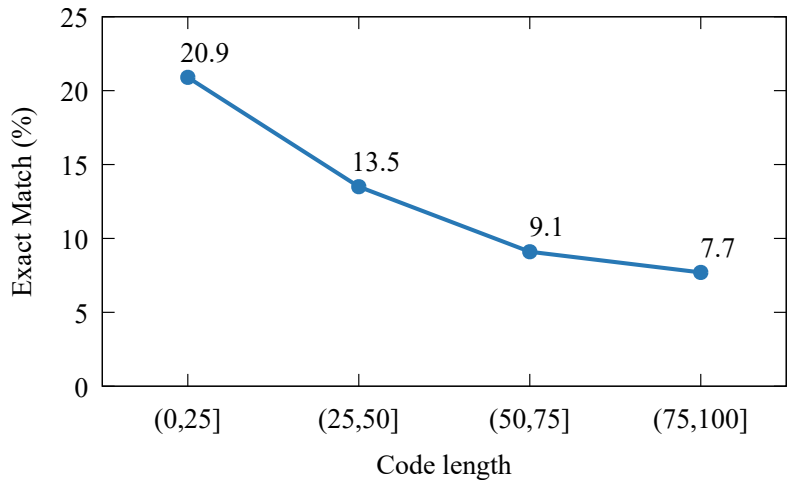

Figure 3. The performance of CodeT5 across different code length groups.

3.4. Dataloader Scheduler

In this part of our discussion, we delve into the mechanics of how the dataloader scheduler orchestrates the organization of training examples into a structured learning curriculum, drawing upon insights from the difficulty measurer detailed in Section 3.3. To craft this curriculum, we employ a scheduling function, denoted as $S(t)$, which dynamically tailors the subset of training data presented at each epoch. This function is conceptualized as follows:

$$S_p(t) = \min\left(1, \left(t\frac{1-c^p}{T} + c^p\right)^{\frac{1}{p}}\right) \quad (1)$$

where t is the current training epoch number, S denotes the fraction of training data in the current epoch, c is the initial proportion of available easiest examples, and T represents the total number of epochs during the training process. The hyperparameter p determines the rate of adding new training samples to the current subset, where $p \geq 1$.

The foundation of our approach is a function inspired by the competence model originally developed for machine translation by Platanios et al. [38]. Leveraging this model, we have crafted a dataloader scheduler that effectively aligns the complexity levels of bug-fixing code samples, as assessed by the difficulty measurer, with the learning needs of PLMCs, as guided by the model training framework. As delineated in Section 3.3, the difficulty measurer categorizes unaltered bug-fix code samples as "easy", while considering those modified by augmentation operators as "hard". In the implementation of our dataloader scheduler, we commence the training with the "easy" samples, utilizing them as the foundational set of training examples. To address the "harder" segments, namely the augmented bug-fix code variants, we further stratify them according to the code length—an indicator of complexity as per the difficulty measurer's assessment. In particular, aligning with the measurer's view that shorter code snippets are less complex, we organize these variants in ascending order of their buggy code length. Subsequently, the scheduler function is employed to progressively integrate these sorted data into the model's training regimen, adjusting the input sequence based on the evolving training dynamics. The scheduling process of the dataloader scheduler is intricately linked to the feedback on training loss from the model trainer. As training advances, the scheduler is designed to introduce data batches corresponding to the next level of difficulty, once the model achieves convergence on the current set, a process visually represented in the right segment of Figure 1.

4. Experimental Setup

4.1. Research Questions

- **RQ1: Effectiveness of our approach.** How effective is our approach in improving the program repair performance of PLMCs?
- **RQ2: Effectiveness of main components.** How effective are the main components of our approach?
- **RQ3: Influence of different dataloader scheduler settings.** How does our approach perform with different dataloader scheduler settings?
- **RQ4: Influence of different code augmentation operator types.** What is the effect of different code augmentation operators?
- **RQ5: Efficiency of our approach.** What is the time cost of applying our approach to PLMCs?
- **RQ6: Generalizability of our approach.** What is the generalizability of our approach in repairing real-world bugs?

4.2. Dataset

We evaluate our approach on three datasets, namely $\text{BFP}_{\text{small}}$ [17], $\text{BFP}_{\text{medium}}$ [17], and Defects4J (v1.2) [18]. The datasets used in our study are all sourced from an open-source GitHub repository, yet they each adhere to distinct methodologies for identifying bugs. $\text{BFP}_{\text{small}}$ and $\text{BFP}_{\text{medium}}$ select commits based on repair-centric messages, while the Defects4J dataset identifies bugs through the execution of test suites. The statistical overview of these datasets is presented in Table 1.

Table 1. Statistics of three automated program repair benchmarks.

Benchmark	Training Set	Validation Set	Test Set
BFP_{small}	46,680	5835	5835
BFP_{medium}	52,364	6545	6545
Defects4J	-	-	388

BFP_{small} and BFP_{medium} mine bug fixes from public GitHub events recorded from March 2011 to October 2017, focusing on commits of Java files flagged by keywords indicative of repair activities. This process involves extracting the source code both before and after each identified bug-fixing intervention, employing sophisticated code abstraction strategies to retain crucial contextual information pertinent to the code's functionality. Within these datasets, each entry is composed of a Java method plagued by a bug alongside its rectified counterpart. Following a rigorous selection process governed by established criteria, code pairs that do not meet the standards are excluded, resulting in the categorization of the remaining data into two distinct subsets: BFP_{small}, encompassing code fragments ranging from 0 to 50 tokens, and BFP_{medium}, including those that extend from 50 to 100 tokens.

Defects4J stands as a comprehensive repository of real-world bugs derived from open-source projects. It includes over 300 meticulously documented and reproducible bugs. Each bug is represented by a pair consisting of a version manifesting the bug and its rectified counterpart, alongside a dedicated test suite engineered to activate the bug, thereby facilitating the verification of potential patches. Furthermore, every bug-fix instance in this dataset is paired with test cases specifically designed to confirm the efficacy of the applied fix.

In the process of fine-tuning PLMCs using our approach, we employ a suite of ten distinct code augmentation operators (elaborated upon in Section 3.2). These operators are utilized to create a variety of code variants from the original training sets. Given that Defects4J provides only the test set, we use the project-specific training data from Ye et al. [32] as the original training set. After transforming the original training sets with code augmentation operators, the generated variants are amalgamated with the original data, resulting in enriched augmented datasets.

4.3. Baselines

In our evaluation, we use both GraphCodeBERT and CodeT5 as foundational models.

GraphCodeBERT is built upon the encoder-only framework reminiscent of BERT, employing a multi-layered bidirectional Transformer as its core structure. It uniquely integrates code data flow into its design and is pre-trained through a triad of tasks: masked language modeling, edge prediction, and node alignment, utilizing the extensive CodeSearchNet dataset, which comprises over 2.3 million functions from six different programming languages, each accompanied by corresponding natural language annotations.

CodeT5 expands the horizon with its 220 million parameter construct, drawing from the encoder–decoder scheme of T5. It contains four pre-training tasks: masked span prediction, identifier tagging, masked identifier prediction, and bimodal dual generation. Masked span prediction requires the model to predict hidden code segments to grasp the code's contextual structure; identifier tagging aims to categorize identifiers in code, like variables and functions, enhancing code comprehension; masked identifier prediction focuses on predicting hidden identifiers in code, deepening the model's understanding of code usage contexts; bimodal dual generation generates two related outputs from a single input, such as code and its documentation, fostering the model's understanding across programming and natural languages. Its training ground is a huge-sized dataset of 8.35 million samples spanning eight programming languages.

We further compare our approach with other different APR methods. For BFP_{small} and BFP_{medium} benchmarks, we set additional baseline methods including Transformer [50],

RoBERTa [51], CodeBERT [3], PLBART [52] and CoTexT [53]. For Defects4J benchmark, the baseline APR methods include SequenceR [29], CoCoNuT [30], DLFix [54], BugLab [55], Recoder [56], CURE [26], RewardRepair [57], DEAR [58], and SelfAPR [32].

4.4. Metrics

For BFP_{small} and BFP_{medium} benchmarks, we use the Exact Match accuracy and BLEU-4 score to evaluate program repair performance following [5]. BLEU-4 offers a more flexible approach to assessing the extent of subword overlap, whereas Exact Match adopts a stringent criterion that demands a perfect match between the predicted and the actual patches in a genuine commit. Across these two evaluation metrics, outcomes are displayed on a scale ranging from 0 to 100 (%), with superior performance denoted by a higher percentile score.

For the Defects4J benchmark, we report the number of bugs that can be accurately fixed by employing both unit testing and manual validation, in alignment with methodologies from previous studies. Initially, we execute test suites to autonomously pinpoint patches that are plausible for each identified bug. This is succeeded by a thorough manual inspection to affirm the accuracy of these patches.

5. Evaluation and Results

5.1. RQ1: Effectiveness of Our Approach

To substantiate the efficacy of our approach, we integrate it into two prominent PLMCs, namely GraphCodeBERT and CodeT5. In the interest of maintaining experimental equity, we adhere to the model parameter configurations as delineated in their respective foundational papers, with the sole modification being an adjustment to the warm-up-steps parameter to accommodate variations in dataset dimensions. For the deployment of our approach, we calibrate the hyperparameter p to a value of 5 and fine-tune the PLMCs for 50 epochs.

The experimental results for both BFP_{small} and BFP_{medium} datasets are shown in Table 2. All baseline outcomes are sourced directly from the respective original publications. Initially, it is noted that the "Naive Copy" strategy yields a significantly high BLEU-4 score, yet it registers a null value in exact match metrics. This observation suggests a substantial overlap between the erroneous code and its correction, underscoring the necessity of prioritizing Exact Match as the foremost evaluation criterion. We find that GraphCodeBERT implemented with our approach, achieves an Exact Match accuracy of 17.82% in BFP_{small} and 10.96% in BFP_{medium}, outperforming the original GraphCodeBERT. Similarly, CodeT5 implemented with our approach achieves an exact match of 22.83% in BFP_{small} and 15.33% in BFP_{medium}, which also substantially outperforms the original CodeT5 model. Moreover, CodeT5 implemented with our approach achieves the best performance among all baseline methods.

It is evident that the application of our approach enhances the success rates of PLMCs, with a notably greater improvement observed in datasets containing longer code samples. This is attributed to the fact that the BFP_{medium} dataset encompasses code samples of longer length, which include a broader array of context variables, identifiers, and literals. The code augmentation operators have the capability to produce a wider variety of code variants for these samples, thereby enriching the PLMCs with a more diverse set of bug-fix data during the fine-tuning phase.

The empirical findings affirm that our approach significantly enhances the efficacy of PLMCs in program repair tasks, without necessitating alterations to their inherent architectural designs.

Table 2. Performance of our approach on the Bugs2Fix benchmark.

Methods	BFP_{small}		BFP_{medium}	
	Exact Match	**BLEU-4**	**Exact Match**	**BLEU-4**
Naive Copy	0.00	78.06	0.00	90.91
LSTM	10.00	76.76	2.50	72.08
Transformer	14.70	77.21	3.70	89.25
RoBERTa (code)	15.90	77.30	4.10	90.07
CodeBERT	16.40	77.42	5.16	91.07
PLBART	19.21	77.02	8.98	88.50
CoTexT	21.58	77.28	13.11	88.40
GraphCodeBERT	17.30	80.02	9.10	91.31
GraphCodeBERT + our approach	17.82	88.43	10.96	88.43
CodeT5	21.61	77.43	13.96	87.64
CodeT5 + our approach	**22.83**	77.77	**15.33**	90.11

5.2. RQ2: Effectiveness of Main Components

To elucidate the impact of core components within our approach, we conduct an ablation analysis. This entailed deploying our approach on the cutting-edge PLMC, CodeT5, utilizing both the BFP_{small} and BFP_{medium} benchmarks, and systematically removing elements such as the curriculum learning (CL) mechanism and code augmentation (CA) operators to gauge their individual contributions. Given that the CL mechanism is contingent upon the enhanced dataset, the exclusion of CA operators inherently nullifies the CL mechanism's functionality. The outcomes of this study are tabulated in Table 3, where the initial entry delineates the comprehensive performance of CodeT5 under the full configuration. The ensuing entry details the effects of omitting the CL mechanism, and the concluding row illustrates the results of excluding both the CA operators and the CL mechanism.

Table 3 demonstrates that the exclusion of any component of our approach diminishes its program repair efficacy, underscoring the integral roles of both the curriculum learning mechanism and code augmentation operators in enhancing bug-fixing capabilities. These results can be attributed to the fact that code variants generated by code augmentation operators enable the model to discern more generalized syntactic and semantic relationships. Concurrently, the curriculum learning mechanism's design facilitates the model's progressive assimilation of increasingly complex bug-fixing code instances, thereby enriching the training process and fostering a more profound comprehension of programming constructs. By synergizing these elements, our approach significantly enhances the PLMC's ability to capture essential syntactic and semantic nuances during the fine-tuning phase.

Table 3. Results of ablation study on Bugs2Fix benchmark.

Methods	Exact Match	
	BFP_{small}	BFP_{medium}
Full Setting	**22.83**	**15.33**
Remove CL	22.23	14.62
Remove CL + CA	22.14	14.19

5.3. RQ3: Influence of Different Dataloader Scheduler Settings

One argument that needs to be predefined in our approach is the dataloader scheduler's parameter p, which determines the rate of adding new training examples to models. With an increment in the value of p, an augmented volume of training data are administered to the model during the initial phases.

To scrutinize the influence of varying p values, we conducted a series of experiments on the BFP_{medium}, as illustrated in Table 4. Utilizing CodeT5 as the foundational model, we implemented our approach with different p. To underscore the scheduler's design

effectiveness, we substituted its function with a geometric progression (GP) function, referenced from [39], while maintaining consistency in all other experimental parameters. Contrary to our approach, the GP function adds harder examples at a faster rate and provides easier examples with more training time. Notably, all dataloader scheduler strategies eventually incorporate the full training dataset during the terminal training phase.

Observations from Table 4 reveal that, across all configurations, our methodology consistently surpasses the baseline. Even when compared with the GP scheduler function, our approach remains superior under all p parameter settings. This superior performance is attributed to our method's provision of ample training duration for the model to internalize complex examples, a boon for the curriculum learning mechanism. The empirical evidence suggests that if we increase the value of p to larger values, such as $p = 20$, the results would be relatively lower. In fact, setting p to infinity renders the scheduler function (1) value to 1, which means the curriculum learning mechanism no longer works. In general, while distinct p values might influence model performance to some degree, our method broadly enhances the model's program repair capabilities.

Table 4. Success Rate of different dataloader scheduler settings on $\text{BFP}_{\text{medium}}$ benchmark.

Methods	Exact Match
Baseline	14.19
GP [39]	14.74
$p = 1$	15.29
$p = 2$	15.14
$p = 5$	15.33
$p = 10$	15.07
$p = 20$	15.00

5.4. RQ4: Influence of Different Code Augmentation Operator Types

In this section, we focus on exploring the impact of different types of code augmentation operations on the performance of automated program repair. The reason for choosing this evaluation approach is that code augmentation techniques are one of the key factors in enhancing the performance of PLMCs in automated program repair tasks. By testing these techniques at various levels of granularity, we can gain a deeper understanding of the specific effects of each augmentation operation on model learning and performance. Similarly to RQ3, we conduct experiments on $\text{BFP}_{\text{medium}}$. We construct augmented datasets of the same size employing code augmentation operators of different types (defined in Section 3.2) and independently train CodeT5 on these datasets. This experimental design allows us to compare the effects of different augmentation operation types under the same experimental conditions, thereby accurately quantifying their impact on model performance. The findings are presented in Table 5. The initial row displays outcomes utilizing all code augmentation operators, whereas the subsequent rows, from the second to the fourth, detail the results when omitting the augmentation operator types specific to declaration, API, or control structure, respectively.

Table 5. Success rate of different code augmentation operator types on $\text{BFP}_{\text{medium}}$ benchmark.

Methods	Exact Match
All Types	15.33
Remove Declaration	14.65
Remove API	14.79
Remove Control	14.94

The results indicate that the omission of different types of code augmentation operators results in diminished outcomes. We can also find that the contribution of control structure transformation is the least compared with other types. We speculate that alterations to

the control structure more significantly affects token sequencing and context compared to augmentation techniques at other granularity levels. Given that the PLMC we use is based on masked language modeling and exhibits sensitivity to context, these factors make it more challenging for the models to assimilate knowledge and features introduced through modifications to the control structure. This insight highlights the need for PLMCs to delve deeper into the structural complexities of source code, thereby improving their understanding of program semantics.

5.5. RQ5: Efficiency of Our Approach

In addressing this RQ, we assess the efficiency of our approach by measuring the time it takes to apply it to a PLMC. CodeT5 serves as our baseline model, and we evaluate the temporal expenditure on BFP_{small} and BFP_{medium}. To maintain equitable conditions, all experiments are conducted over 50 epochs. The findings are depicted in Table 6.

Table 6. The relative time required for applying our method. Baseline refers to the time cost of the model fine-tuned without our approach.

Methods	Time Cost	
	$\mathbf{BFP_{small}}$	$\mathbf{BFP_{medium}}$
Baseline	1	1
Our Method	1.08	1.28

From the results, we can find that after applying our approach, the training time on the BFP_{small} dataset is 1.08 times longer than without it, and it is 1.28 times longer on the BFP_{medium} dataset. The reason for the relatively longer training time on the BFP_{medium} dataset compared to the BFP_{small} dataset is that the augmentation operators in our approach can generate more code variants for the former. Given the expansion in data size and the enhancement in model performance, this represents an acceptable time cost.

In practical applications, software developers utilize PLMCs that have been fine-tuned with our approach. Consequently, the efficiency of generating patches in real-world scenarios aligns with the performance of the original PLMCs.

5.6. RQ6: Generalizability of Our Approach in Repairing Real-World Bugs

In this RQ, we evaluate the generalizability of our approach in repairing real-world bugs on the Defect4J benchmark. We use CodeT5 as a baseline model and apply our approach to it. We also compare it with other APR tools, as shown in Table 7.

Table 7. Performance of our approach on Defects4J benchmark.

Methods	# Correct
SequenceR	14
BugLab	17
DLFix	40
CoCoNuT	43
RewardRepair	44
DEAR	53
CURE	55
Recoder	64
SelfAPR	65
CodeT5	58
CodeT5 + our approach	**67**

As shown in Table 7, after applying our method to CodeT5, the number of correctly generated patches increased by 9 compared to the original model, surpassing other APR

tools. This corroborates the results of previous experiments. This finding further demonstrates our method's effectiveness in fixing real-world bugs, paving the way for its wider implementation in automated program repair.

6. Conclusions

In this study, we proposed an innovative fine-tuning framework engineered to boost the success rate of PLMCs in the realm of APR. Our approach employs a curriculum learning mechanism to strategically sequence the training of bug-fixing code pairs, which are derived using a series of code augmentation operators. We deploy our approach on representative PLMCs to assess their adaptability and effectiveness. Our thorough experimental analysis reveals that our approach substantially improves the performance of PLMCs in program repair.

Despite the achievements of our approach, we recognize that it has certain limitations. Firstly, because the total number of training epochs in the dataloader scheduler is fixed, the model cannot dynamically adjust the number of iterations based on the current training outcomes. Secondly, our experimental analysis was primarily conducted on Java datasets. For datasets involving other programming languages or multi-languages, the code augmentation operators we used might need to be replaced or redesigned.

Future research will be dedicated to expanding our framework to adaptively adjust according to the training state of the model at different times and to further optimize the quality and diversity of code augmentation operations, thus addressing a broader range of programming errors. Moreover, we plan to explore the possibility of integrating our method with other large language models to further increase the success rate of repairs. We believe that with continuous effort and innovation, the application of PLMCs in software engineering will become more widespread and efficient.

Author Contributions: Conceptualization, S.H.; Methodology, S.H.; Validation, X.S.; Formal analysis, X.S.; Investigation, H.L.; Writing—original draft, S.H.; Writing—review & editing, H.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported by the National Key Research and Development Program of China (No. 2022YFC3301800).

Data Availability Statement: The contributions presented in the study are included in the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Monperrus, M. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* **2018**, *51*, 1–24 . [CrossRef]
2. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. Competition-level code generation with AlphaCode. *Science* **2022**, *378*, 1092–1097. [CrossRef] [PubMed]
3. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020, Online, 16–20 November 2020; pp. 1536–1547. [CrossRef]
4. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proceedings of the International Conference on Learning Representations, Vienna, Austria, 4 May 2021.
5. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Virtual, 7–11 November 2021; pp. 8696–8708. [CrossRef]
6. Bubeck, S.; Chandrasekaran, V.; Eldan, R.; Gehrke, J.; Horvitz, E.; Kamar, E.; Lee, P.; Lee, Y.T.; Li, Y.; Lundberg, S.; et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv* **2023**, arXiv:2303.12712.
7. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA, 2–7 June 2019; pp. 4171–4186. [CrossRef]
8. Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; Liu, P.J. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* **2020**, *21*, 1–67.

9. Xia, C.S.; Wei, Y.; Zhang, L. Automated Program Repair in the Era of Large Pre-trained Language Models. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)—ICSE '23, Melbourne, Australia, 14–20 May 2023; pp. 1482–1494. [CrossRef]
10. Niu, C.; Li, C.; Luo, B.; Ng, V. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. In Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, Vienna, Austria, 23–29 July 2022; Volume 6, pp. 5546–5555. [CrossRef]
11. Panthaplackel, S.; Allamanis, M.; Brockschmidt, M. Copy that! editing sequences by copying spans. In Proceedings of the AAAI Conference on Artificial Intelligence, Virtual, 2–9 February 2021; Volume 35, pp. 13622–13630.
12. Tay, Y.; Dehghani, M.; Rao, J.; Fedus, W.; Abnar, S.; Chung, H.W.; Narang, S.; Yogatama, D.; Vaswani, A.; Metzler, D. Scale Efficiently: Insights from Pretraining and Finetuning Transformers. In Proceedings of the International Conference on Learning Representations, Virtual, 25–29 April 2022.
13. Jiang, N.; Liu, K.; Lutellier, T.; Tan, L. Impact of Code Language Models on Automated Program Repair. *arXiv* **2023**, arXiv:2302.05020.
14. Fan, Z.; Gao, X.; Mirchev, M.; Roychoudhury, A.; Tan, S.H. Automated Repair of Programs from Large Language Models. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)—ICSE '23, Melbourne, Australia, 14–20 May 2023; pp. 1469–1481. [CrossRef]
15. Xia, C.S.; Zhang, L. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, 14–18 November 2022. [CrossRef]
16. Bengio, Y.; Louradour, J.; Collobert, R.; Weston, J. Curriculum learning. In Proceedings of the 26th Annual International Conference on Machine Learning, Montreal, QC, Canada, 14–18 June 2009; pp. 41–48. [CrossRef]
17. Tufano, M.; Watson, C.; Bavota, G.; Penta, M.D.; White, M.; Poshyvanyk, D. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* **2019**, *28*, 1–29. [CrossRef]
18. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis—ISSTA 2014, San Jose, CA, USA, 23–25 July 2014; pp. 437–440. [CrossRef]
19. Hao, S.; Shi, X.; Liu, H.; Shu, Y. Enhancing Code Language Models for Program Repair by Curricular Fine-tuning Framework. In Proceedings of the 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bogotá, Colombia, 1–6 October 2023; pp. 136–146. [CrossRef]
20. Radford, A.; Narasimhan, K.; Salimans, T.; Sutskever, I. Improving Language Understanding by Generative Pre-Training. 2018. Available online: https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf (accessed on 18 March 2024).
21. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.d.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating large language models trained on code. *arXiv* **2021**, arXiv:2107.03374.
22. Niu, C.; Li, C.; Ng, V.; Ge, J.; Huang, L.; Luo, B. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 1–13. [CrossRef]
23. Zhang, J.; Panthaplackel, S.; Nie, P.; Li, J.J.; Gligoric, M. CoditT5: Pretraining for Source Code and Natural Language Editing. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering—ASE '22, Rochester, MI, USA, 10–14 October 2022; pp. 1–12. [CrossRef]
24. Mastropaolo, A.; Cooper, N.; Palacio, D.N.; Scalabrino, S.; Poshyvanyk, D.; Oliveto, R.; Bavota, G. Using Transfer Learning for Code-Related Tasks. *IEEE Trans. Softw. Eng.* **2022**, *49*, 1580–1598. [CrossRef]
25. Yuan, W.; Zhang, Q.; He, T.; Fang, C.; Hung, N.Q.V.; Hao, X.; Yin, H. CIRCLE: Continual repair across programming languages. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, Virtual, 18–22 July 2022; pp. 678–690. [CrossRef]
26. Jiang, N.; Lutellier, T.; Tan, L. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain, 25–28 May 2021; pp. 1161–1173. [CrossRef]
27. Berabi, B.; He, J.; Raychev, V.; Vechev, M. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In Proceedings of the 38th International Conference on Machine Learning, Online, 18–24 July 2021; pp. 780–791.
28. Chi, J.; Qu, Y.; Liu, T.; Zheng, Q.; Yin, H. SeqTrans: Automatic Vulnerability Fix Via Sequence to Sequence Learning. *IEEE Trans. Softw. Eng.* **2023**, *49*, 564–585. [CrossRef]
29. Chen, Z.; Kommrusch, S.J.; Tufano, M.; Pouchet, L.N.; Poshyvanyk, D.; Monperrus, M. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Softw. Eng.* **2021**, *47*, 1943–1959. [CrossRef]
30. Lutellier, T.; Pham, H.V.; Pang, L.; Li, Y.; Wei, M.; Tan, L. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Virtual, 18–22 July 2020; pp. 101–114. [CrossRef]
31. Chakraborty, S.; Ray, B. On Multi-Modal Learning of Editing Source Code. In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering—ASE '21, Melbourne, Australia, 15–19 November 2022; pp. 443–455. [CrossRef]

32. Ye, H.; Martinez, M.; Luo, X.; Zhang, T.; Monperrus, M. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering—ASE '22, Rochester, MI, USA, 10–14 October 2022; pp. 1–13. [CrossRef]
33. Elman, J.L. Learning and development in neural networks: The importance of starting small. *Cognition* **1993**, *48*, 71–99. [CrossRef] [PubMed]
34. Krueger, K.A.; Dayan, P. Flexible shaping: How learning in small steps helps. *Cognition* **2009**, *110*, 380–394. [CrossRef] [PubMed]
35. Huang, Y.; Wang, Y.; Tai, Y.; Liu, X.; Shen, P.; Li, S.; Li, J.; Huang, F. CurricularFace: Adaptive Curriculum Learning Loss for Deep Face Recognition. In Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 13–19 June 2020; pp. 5900–5909. [CrossRef]
36. Xu, B.; Zhang, L.; Mao, Z.; Wang, Q.; Xie, H.; Zhang, Y. Curriculum Learning for Natural Language Understanding. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online, 5–10 July 2020; pp. 6095–6104. [CrossRef]
37. Hacohen, G.; Weinshall, D. On The Power of Curriculum Learning in Training Deep Networks. In Proceedings of the 36th International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; pp. 2535–2544.
38. Platanios, E.A.; Stretcu, O.; Neubig, G.; Poczos, B.; Mitchell, T.M. Competence-based Curriculum Learning for Neural Machine Translation. In Proceedings of the NAACL-HLT, Minneapolis, MN, USA, 2–7 June 2019; pp. 1162–1172.
39. Penha, G.; Hauff, C. Curriculum Learning Strategies for IR: An Empirical Study on Conversation Response Ranking. *Proc. Adv. Inf. Retr.* **2020**, *12035*, 699–713.
40. Wang, D.; Jia, Z.; Li, S.; Yu, Y.; Xiong, Y.; Dong, W.; Liao, X. Bridging pre-trained models and downstream tasks for source code understanding. In Proceedings of the 44th International Conference on Software Engineering—ICSE '22, Pittsburgh, PA, USA, 8–20 May 2022; pp. 287–298. [CrossRef]
41. Zhong, W.; Ge, H.; Ai, H.; Li, C.; Liu, K.; Ge, J.; Luo, B. StandUp4NPR: Standardizing SetUp for Empirically Comparing Neural Program Repair Systems. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering—ASE '22, Rochester, MI, USA, 10–14 October 2022; pp. 1–13. [CrossRef]
42. Yu, S.; Wang, T.; Wang, J. Data Augmentation by Program Transformation. *J. Syst. Softw.* **2022**, *190*, 111304. [CrossRef]
43. Chakraborty, S.; Ahmed, T.; Ding, Y.; Devanbu, P.T.; Ray, B. NatGen: Generative Pre-Training by "Naturalizing" Source Code. In Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, 14–18 November 2022; pp. 18–30. [CrossRef]
44. Yang, Z.; Shi, J.; He, J.; Lo, D. Natural Attack for Pre-Trained Models of Code. In Proceedings of the 44th International Conference on Software Engineering—ICSE '22, Pittsburgh, PA, USA, 8–20 May 2022; pp. 1482–1493. [CrossRef]
45. Jain, P.; Jain, A.; Zhang, T.; Abbeel, P.; Gonzalez, J.; Stoica, I. Contrastive Code Representation Learning. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Online, 7–11 November 2021; pp. 5954–5971. [CrossRef]
46. Yefet, N.; Alon, U.; Yahav, E. Adversarial Examples for Models of Code. *Proc. ACM Program. Lang.* **2020**, *4*, 1–30. [CrossRef]
47. Zhou, Y.; Zhang, X.; Shen, J.; Han, T.; Chen, T.; Gall, H. Adversarial Robustness of Deep Code Comment Generation. *ACM Trans. Softw. Eng. Methodol.* **2022**, *31*, 1–30. [CrossRef]
48. Zeng, Z.; Tan, H.; Zhang, H.; Li, J.; Zhang, Y.; Zhang, L. An extensive study on pre-trained models for program understanding and generation. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022, Virtual, 18–22 July 2022; pp. 39–51. [CrossRef]
49. Liu, S.; Wu, B.; Xie, X.; Meng, G.; Liu, Y. ContraBERT: Enhancing Code Pre-trained Models via Contrastive Learning. *arXiv* **2023**, arXiv:2301.09072.
50. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In Proceedings of the Advances in Neural Information Processing Systems 30 (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017.
51. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv* **2019**, arXiv:1907.11692. https://doi.org/10.48550/arXiv.1907.11692.
52. Ahmad, W.; Chakraborty, S.; Ray, B.; Chang, K.W. Unified Pre-training for Program Understanding and Generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Online, 6–11 June 2021; pp. 2655–2668. [CrossRef]
53. Phan, L.; Tran, H.; Le, D.; Nguyen, H.; Annibal, J.; Peltekian, A.; Ye, Y. CoTexT: Multi-task Learning with Code-Text Transformer. In Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), Online, 6 August 2021; pp. 40–47. [CrossRef]
54. Li, Y.; Wang, S.; Nguyen, T.N. DLFix: Context-based Code Transformation Learning for Automated Program Repair. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering—ICSE '20, Seoul, Republic of Seoul, 27 June–19 July 2020; pp. 602–614. [CrossRef]
55. Allamanis, M.; Jackson-Flux, H.R.; Brockschmidt, M. Self-Supervised Bug Detection and Repair. *arXiv* **2021**, arXiv:2105.12787.
56. Zhu, Q.; Sun, Z.; Xiao, Y.a.; Zhang, W.; Yuan, K.; Xiong, Y.; Zhang, L. 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, ESEC/FSE 2021, Athens, Greece, 23–28 August 2021; pp. 341–353. [CrossRef]

57. Ye, H.; Martinez, M.; Monperrus, M. Neural program repair with execution-based backpropagation. In Proceedings of the 44th International Conference on Software Engineering—ICSE '22, Pittsburgh, PA, USA, 8–20 May 2022; pp. 1506–1518. [CrossRef]
58. Li, Y.; Wang, S.; Nguyen, T.N. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In Proceedings of the 44th International Conference on Software Engineering—ICSE '22, Pittsburgh, PA, USA, 8–20 May 2022; pp. 511–523. [CrossRef]