A SYSTEMATIC REVIEW OF AUTOMATED PROGRAM REPAIR USING LARGE LANGUAGE MODELS

by

Callum MacNeil

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science with Honours at

Dalhousie University
Halifax, Nova Scotia
December 2023

© Copyright by Callum MacNeil, 2023
This thesis is dedicated to my Uncle Bruce, who encouraged me to do more math problems and switch to a degree in computer science.
# Table of Contents

List of Tables .................................................. vi
List of Figures .................................................. vii
Abstract .......................................................... viii

Chapter 1   Introduction ........................................ 1
  1.1 Motivation and Research Problem .......................... 1
  1.2 Contributions .............................................. 2

Chapter 2   Background ........................................... 3
  2.1 General ...................................................... 3
  2.2 Popular Architectures .................................... 3
  2.3 Input Representation ..................................... 4
  2.4 Modality in LLMs ......................................... 4
  2.5 Loss Functions ............................................ 5
  2.6 Training Paradigms ..................................... 5
  2.7 Pretraining objectives ................................. 5
  2.8 Beam Search ............................................... 6
  2.9 Summary .................................................. 6

Chapter 3   Methodology ........................................ 7
  3.1 Research Questions ..................................... 7
  3.2 Search Strategy ......................................... 7
  3.3 Paper Selection Process ................................. 8
    3.3.1 Selection Analysis .................................. 9
  3.4 Data Extraction from Primary Studies .................. 9
  3.5 Summary .................................................. 11
Chapter 4  RQ1: Which types of pre-training methods, input/prompts, and datasets are used for LLM based APR?  

4.1 Pretraining and Finetuning Methods  
4.1.1 Pretraining and Finetuning  
4.1.2 Finetuning Only  
4.1.3 Neither Pretraining or Finetuning  

4.2 Input Representation and Prompting  
4.2.1 AST  
4.2.2 Abstraction  
4.2.3 Byte-Pair Encoding  

4.3 Dataset Collection and Construction  
4.3.1 Studies building their own Datasets  
4.3.2 Studies Combining Datasets  
4.3.3 Studies using Student Data  
4.3.4 Studies which Scrape Open Source Repositories  

4.4 Summary  

Chapter 5  RQ2: Which evaluation methods are used for LLM based APR?  

5.1 Overview of validation dataset metrics  
5.1.1 APR evaluation assisting studies  
5.1.2 Accuracy and Non-Accuracy Evaluation Metrics  

5.2 Summary  

Chapter 6  RQ3: What are the Strengths, Weaknesses and Future Direction for APR  

6.1 Strengths  
6.1.1 Datasets  
6.1.2 Reinforcement Learning  
6.1.3 Input Representations  

6.2 Weaknesses  
6.2.1 Lack of Extensive Pretraining  
6.2.2 Lack of Certainty  

6.3 Future Direction  
6.3.1 Advanced Reward Functions  
6.3.2 Dataset Curation  
6.3.3 Expanding Input Representation
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4 Summary</td>
<td>36</td>
</tr>
<tr>
<td>Chapter 7 Conclusion</td>
<td>37</td>
</tr>
<tr>
<td>Bibliography</td>
<td>38</td>
</tr>
<tr>
<td>Appendix A Appendix</td>
<td>47</td>
</tr>
<tr>
<td>A.1 Project Repository</td>
<td>47</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Inclusion and exclusion Criteria ........................................... 8
4.1 Pretraining and Finetuning Studies ........................................ 12
4.2 Input Representations ......................................................... 19
4.3 Experimental Dataset (Reused) ............................................ 25
4.4 Experimental Dataset (Constructed) ...................................... 27
5.1 Evaluation Metrics ............................................................. 28
List of Figures

3.1 Years of papers ................................................. 10
3.2 Search and Filtration ........................................... 10
4.1 Input representation example (a) [52] ......................... 18
4.2 Input representation example (b) [52] ........................ 19
4.3 Code Abstraction Example [37] ............................... 21
4.4 Graph explaining the datasets used together across multiple studies ................................................. 26
Abstract

Artificial Intelligence has received significant attention recently, thanks to large language models. An exciting aspect of these language models is their ability to generate code, which has the potential to save development time and effort. However, developers often make mistakes when reusing or writing code, which leads to software bugs. These software bugs take up to 50% of development time. Thus, code generation alone will not help much unless there is also a way to generate fixes to the bugs. Over the years, many approaches have been proposed to synthesize solution code against the buggy or faulty code, which is known as program repair. In this thesis, we perform a systematic review of automated program repair techniques that leverage large language models. We search eight databases and gather a total of 1,276 papers (published between 2017 and 2023) that are related to automated program repair leveraging large language models. We then methodically filter them to obtain our final set of 53 primary studies. Next, we extract important aspects from each paper, e.g., datasets, architecture, performance, answer three research questions, and report three major findings. First, the majority of studies choose to reuse popular datasets and pre-trained models, e.g., Defects4j and CodeT5. Second, rather than addressing the problem as a whole, these studies target specific aspects of program repair, such as validation, input representation, or pre-training. Third, we see that existing approaches are challenged by their choice of input representation, evaluation methods, and learning objectives. We observe that the choice between efficacy or efficiency shapes the overall approach of each study. Finally, we use our developed understanding of the current state of the art in this field and identify future directions and best approaches for research on automated program repair using large language models (LLMs).
Chapter 1

Introduction

1.1 Motivation and Research Problem

Large Language Models (LLMs) have recently enjoyed much attention and development. LLMs are neural networks powered by an attention mechanism, which stems from the transformer architecture introduced in 2017 [74]. They are typically trained on large amounts of data, which equips the model with a general understanding of the underlying topics. These models have shown unprecedented semantic understanding and generation ability. Their rapid development and superior performance in text generation tasks have caused the code generation landscape to change drastically. Code generation tools such as GitHub Copilot and ChatGPT are being widely adopted by the developer communities. Despite this, developers must still be cautious since these LLMs could inject bugs into their designed programs. According to an existing survey, software bugs can consume up to 50% of developers’ time [52]. Code generation alone might not lead to increased developer productivity unless we have effective ways to generate fixes against the buggy code. Over the last 15 years, several traditional techniques, such as genetic algorithms and template-based repair have been applied to automated program repair (APR). Issues including, a low rate of quality of repairs, a lack of ability to produce generic fixes, the unrestricted vocabulary of programming languages, and syntactic differences between languages make a traditional APR tool infeasible. All of these issues prevent a large-scale adoption of traditional APR solutions by industry[25]. Interestingly, since the onset of LLMs in 2017, the generative ability of LLMs has shown promising results in code generation. Therefore, significant work has been done over the last few years that leverages the power of LLMs in automated program repair.

There exists a few systematic reviews of LLMs in the context of software engineering [24, 84, 12, 81, 25]. Some of these reviews briefly mention APR using LLMs [24, 25] and touch on the promising ability of tools such as ChatGPT. However, they
do not cover the fundamental design choices researchers make when leveraging LLMs for APR. To advance the research in this area, we need a comprehensive overview of the current state of work leveraging LLMs for APR. The existing reviews are not adequate for a thorough understanding of APR with LLMs. Therefore, this research performs a systematic literature review of APR using LLMs as an important step for future APR research. Unlike the existing systematic reviews, each of our research questions target a different aspect of LLM based APR, thus increasing the value of this review for the APR research community.

1.2 Contributions

In this thesis, we perform a systematic review of automated program repair techniques that leverage large language models. We first introduce our research questions and describe the process of our systematic search and filtration in Chapter 3. Our search of 8 databases resulted in 1,276 papers being collected. The multiple filtration steps we performed reduced the 1,276 papers to a final total of 53 primary studies. We then carefully extract the relevant data from each study, and organize them into tables. Our methodology, data extraction and overall approach enables us to make the following contributions. We present and summarize the data extracted from each study in Chapter 4 and Chapter 5. In those chapters, we categorize each of the evaluation metrics, datasets, input representations, pretraining and finetuning methods. We determine the categories based on the overall characteristics of the extracted data. From our extracted and organized data, we identify and describe specific subcategories related to each component of our research questions. Using our developed understanding, we identify important strengths and weaknesses in Chapter 6 in order to infer a future direction for research in the APR domain that leverages LLMs.
Chapter 2

Background

In this literature review, we use several technical concepts and jargon related to machine learning. The reader must understand them to get the most out of this thesis. Therefore, we define several helpful terminology related to large language models in this chapter.

2.1 General

Large Language Model is an attention based neural network, it consists of a high number of parameters in the range 1M to 1T. These parameters are obtained by training the network on a large amount of data (e.g. multiple terabytes). The training process usually consumes thousands to millions of GPU hours.

Neural Machine Translation (NMT) is the automatic translation of one language text to another text using neural networks.

2.2 Popular Architectures

Encoder-Decoder, Decoder only and Encoder only: are different classes of transformer based models. The main difference between the three is that encoder-only models cannot be used for generation as the encoder just produces an embedding. Encoder only models such as BERT are effective at classification tasks, when used in combination with a discriminative network. Decoder only models can be used for generation, however, the input to these models must be real numbers, so some kind of embedding must be given to a decoder only model. Encoder-Decoder merge the aforementioned architectures and train both an encoder and decoder in unison.

Long Short Term Memory (LSTM) is a neural network where some outputs are fed back to the model as input, this is called a recurrent neural network (RNN). The LSTM is a recurrent neural network with an attention mechanism, this gives the
model a way to store outputs and use them as additions to inputs.

**CodeT5** is a bimodal, encoder-decoder language model, which was made open source by Salesforce in 2019. Based on T5 CITE, CodeT5 leverages denoising pretraining, and 3 other popular pretraining objectives which are: Masked Span Prediction, Masked Identifier Prediction, and Identifier tagging. [75]

**BERT** stands for bi-directional encoder representations from transformers. BERT is an encoder only model released by researchers at Google in 2018.[13]

**BART** is an encoder-decoder model which leverages denoising pretraining where the encoder randomly adds noise to the input and the decoder tries to recover the original input. [43]

### 2.3 Input Representation

**Embedding:** A vector of numbers that represents some input: (natural language, programming language, pdf image,...)

**Encoding** The process of obtaining an embedding from some input.

**Decoding** The process of obtaining an output from an embedding. This output will typically be an understandable form, such as a picture, or natural/programming language text.

**AST** or abstract syntax tree is an input that is able to represent the syntactic structure of the source code[86].

### 2.4 Modality in LLMs

**Unimodal models** are models that can only understand one input language (EX: a unimodal model built for natural language cannot understand programming languages.)

**Multimodal models** are models that can understand multiple input languages. (EX: CodeT5 is a bimodal model which takes both programming language and natural language as input.)
2.5 Loss Functions

**Cross Entropy Loss**: Cross Entropy is defined as the average number of bits required to represent an event from a source probability distribution, with a different distribution (from a model). Cross entropy loss is used as a loss/error function for classification models: Given a set of labeled classes, the model predicts the probability of an example having a specific labeled class. This gives two probability distributions. Cross entropy is used to quantify and calculate the difference between the ground truth distribution and the model’s learned distribution. [6]

2.6 Training Paradigms

**Supervised Learning**: takes place when researchers have access to data with ground truths. This enables the model to train on predicting/generating these ground truths, from the input data. The ground truths could be class labels, or, for example: correct bug patches. [10]

**Transfer Learning**: is a finetuning method where researchers aim to transfer a pretrained language models knowledge of one domain, to another related domain with less training data. [38]

**Curriculum Learning**: is a training method where a model is trained with training samples ordered by difficulty so that the model can work up to harder examples. [91]

**Syntactic Training**: is training done with cross-entropy loss and/or denoising pretraining. This kind of training enables the model to learn the vocabulary and syntactic rules of the input.

**Semantic Training**: Different from syntactic training, the model will have an understanding of the input syntax before semantic training. Semantic training combines pretraining objectives with additional feedback. For programming languages, an example of additional correctness info is compilation information. [85]

2.7 Pretraining objectives

**Masked Span Prediction**: is when a section or span of the input is hidden from the model, the model is to predict this hidden section. [75]
**Masked Language Modeling**: is when input tokens are randomly masked, different from the section masking in masked span prediction. The model is then trained to predict the masked word based on the surrounding context.[14]

**Masked Identifier Prediction**: A specific case of masked span prediction, the model is trained to predict masked identifiers. This was introduced by CodeT5 and aligns with their other proposed task: identifier tagging.

**Identifier Tagging**: Is a pretraining objective where the model is tasked with classifying tokens 'identifiers' (e.g. variable names) or not.

**Denoising Pretraining**: Also called sequence to sequence (Seq2Seq) pretraining. A source sequence first has noise applied to it. This noise can be applied with an encoder. A decoder then attempts to obtain the original sequence from the noisy sequence. This is done to pretrain decoders and to avoid randomly initializing them. [5]

### 2.8 Beam Search

**Beam Search** is used to obtain multiple outputs from a model, with one given input. Language models generate the next word of a sequence based on their generated probability of the next word. Most use cases call for the beam size to be 1. That is the model outputs the top 1 most probable output. When beam size, k, is greater than 1, the model will output the top k most probable outputs.

### 2.9 Summary

In this section we discussed several key terms which provide some background knowledge on large language models and LLM related tasks.
Chapter 3

Methodology

We follow the guidelines of Kitchenham and Charters [39] for our systematic review. Our objective is to capture all of the state of the art techniques using LLMs for APR and determine the current state of the art research.

3.1 Research Questions

In this chapter, we discuss the methodology of our conducted review. Kitchenham and Charters [39] suggest that research questions are the most important aspect of a systematic literature review. The process of a review is shaped by the research questions. We carefully craft three research questions to facilitate a thorough review that collects relevant information from all primary studies. RQ1 & RQ2 are general questions about the design of LLM based APR tools. On the other hand, RQ3 is a focused question that examines the strengths or weaknesses of the existing work and discusses the future work.

- **RQ1**: Which types of pre-training methods, input/prompts, and datasets are used for LLM based APR?
- **RQ2**: Which evaluation methods are used for LLM based APR?
- **RQ3-a**: What are the strengths and weaknesses of existing approaches?
- **RQ3-b**: What should be the future direction of LLM APR?

3.2 Search Strategy

Fig 3.2 summarizes our study selection and filtration process. Our search keywords are derived from the above research questions. We use the PIO strategy to select search keywords, as suggested by Kitchenham and Charters [39]. We formulate our
keywords according to the template from Teesside University [70]: ”With ‘Population’ what is the effect of ‘Intervention’ on ‘Outcome’”. In other words, population is all aspects related to the scope of this research. Intervention is the specific traits of the population. Outcome refers to what and how this research is relevant to the stakeholders, in our case developers. Thus we choose the following keywords:

\[ \mathbf{P} = \{ \text{Automated Program Repair, APR, Code Language Model, CLM, Large Language Model, LLM, Automated Bug Fixing, Transformer} \} \]

\[ \mathbf{I} = \{ \text{Code Generation, Input, Prompt, Pre-processing, Data Quality, Data Collection, Testing, Evaluation} \} \]

\[ \mathbf{O} = \{ \text{Code Quality, Bug Fixing} \} . \]

We take inspiration from an existing literature review by Rahman et al. [64] and search the following eight databases:


### 3.3 Paper Selection Process

We define our criteria for inclusion and exclusion before conducting the search. (a). *Inclusion Criteria*: All reviewed papers should target automated program repair using LLMs. (b). *Exclusion Criteria*: To avoid irrelevant, or unusable studies we have determined an exclusion criteria in Table 3.1. This criteria guides the research in discarding studies that are unrelated, low quality or unobtainable.

<table>
<thead>
<tr>
<th><strong>Inclusion Criteria</strong></th>
<th><strong>Exclusion Criteria</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Target APR</td>
<td>Non-English Papers</td>
</tr>
<tr>
<td>Primary Study</td>
<td>Written before 2017</td>
</tr>
<tr>
<td>Found in initial search</td>
<td>less than 2 evaluation metrics</td>
</tr>
<tr>
<td>N/A</td>
<td>Unpublished</td>
</tr>
<tr>
<td>N/A</td>
<td>Paid access/inaccessible</td>
</tr>
</tbody>
</table>
After we perform our search, we obtain a total of 1,276 candidate studies. Initial search results are likely to contain tangentially related or even unrelated studies. We thus scan through the titles of the collected papers and found such keywords that indicate the irrelevance of a paper. We identified a total of 11 keywords: medical, genetic, IR-based, gender, school, student, bootcamp, gender, education, junior. All papers containing those keywords in their titles are removed. We also filter all duplicate papers and non-primary studies. Non-primary studies are studies that obtain their data through other studies. An example of a non primary study is a literature review.

The automatic filtering left us with 170 potential papers. Such a number was too large for an in-depth analysis. Therefore we begin manually filtering papers. We first go through the title and abstract of each paper and add them to one of the three categories: irrelevant, possibly relevant, and relevant. From there, we do a full scan on each of the possibly relevant papers, looking for keywords that indicate the study is related. The filtration process continues until the first pass of all relevant papers is completed. In the end we are left with a total of 53 relevant primary studies to review.

3.3.1 Selection Analysis

From Fig 3.1, we see that about 42% of our selected studies were published in 2023 and that 75% of the remaining studies were published in the last two years. This shows an increasing excitement and interest around the applications of large language models.

3.4 Data Extraction from Primary Studies

To extract important data points from each study, we consider our first research question. we make tables for each methodology, Pretraining/Finetuning, Input Representation, Dataset Collection, Dataset size from research question 1. Then, we go through each of the 53 papers and extract the required data defined by these tables.
Then, we attempt to summarize the approaches used by each of the primary studies. We group each category based on their subcategories and attempt to describe the overarching approaches for automated program repair.
3.5 Summary

In this chapter, we introduced our research questions and our review methodology which defines the remaining chapters of our review.
Chapter 4

RQ1: Which types of pre-training methods, input/prompts, and datasets are used for LLM based APR?

In this section, we discuss the pre-training methods, input/prompts and data collection methods used by our primary studies.

4.1 Pretraining and Finetuning Methods

We have 3 overarching categories in terms of pretraining/finetuning. Table 4.1 shows how these categories are distributed across the primary studies. We discuss each of these categories as follows.

4.1.1 Pretraining and Finetuning

In this section, we discuss the studies that pretrain and finetune models. According to our analysis, studies performing both pretraining and finetuning make up around 31% of the collected studies. Their distribution from 2023 to 2019 is: 5, 5, 5, 3, 0, 1. These studies can also be divided into multiple subcategories or reasons for pretraining and finetuning: training multiple models for ensemble learning, modifying the loss function with additional information, pretraining due to an innovative

<table>
<thead>
<tr>
<th>Studies</th>
<th>Pre-training</th>
<th>finetuning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[23, 73], [31, 16, 33], [15, 1, 90, 3], [57, 82, 50], [44, 30, 85, 8], [7, 77, 41] (19 total)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>[34, 53, 61], [68, 54, 29], [83, 37, 38], [36, 20, 91], [45] (13 total)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>[69, 22, 60], [78, 40, 19, 62], [47, 80, 76], [56, 35, 72], [63, 55, 71], [79, 87, 66, 88], [48] (21 total)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
input representation, and pretraining their own architecture consisting of Long-Short-Term-Memory (LSTM) models. A common aspect of these studies is that they have significantly larger datasets than that of studies not performing any pretraining. This is due to the fact that pretraining typically involves training a model and using many gigabyte of data to make it comparable to other relevant pretrained models.

Several primary studies pretrain multiple models. Tufano et al. pretrain multiple encoders and select the best one based on errors in their decoded sequence. A more recent study, Jiang et al. (2023), train multiple models but, instead of taking the best model, they take the top-k best models. These top-k models are used as an ensemble for the bug fix generation task.

Several primary studies trained their models as an ensemble, such as DLFix (Li et al.) [44], CoCoNut (Lutellier et al.) [50], Horváth et al. [23], and CURE (Jiang et al.) [31]. Li et al. trained a model to construct a context vector, which was later used as an extra input to their generative model. Lutellier et al., exploit the randomness in hyper parameter tuning to train multiple convolutional networks specialized at fixing different types of bugs. They also train two context-aware models to encode buggy lines and context separately. Horváth et al. pretrain CodeT5 from scratch and experiment with three different input representations that require two additional encoders. They use RoBERTa for encoding AST and CodeBERT for encoding source code. The ensemble of these 3 models is then used for code generation. Jiang et al. use next word prediction as the pre training objective for an neural machine translation (NMT). During finetuning they combine their trained NMT model with a pretrained LM. Then use this ensemble finetuning to improve their next token prediction.

Several primary studies modify the loss function in their models to include additional relevant information about output correctness. Yang et al. [82] train a adversarial network that receives outputs from the generative model and returns a reward value used to train the generative model with a reinforcement learning policy. Their adversarial network was used to select multiple candidate patches. Yang et al. perform some innovation in their pretraining and finetuning. Using a bug repair dataset, they perform syntactic training based on cross entropy loss. Their main innovation lies in the Semantic training, where a mixed learning objective involving
the cross entropy objective, as well as compilation and execution info are used. In order for their model to be trained on the compilation and execution info, they use a discriminative model that assesses the quality of patches and outputs a reward.

Several primary studies pretrain their models out of necessity. Here, innovation lies in what exactly each study is attempting to teach their model. This involves a complex input representation that requires pretraining. Studies under this category are Namavar et al.[57], Tare (Zhu et al.) [90], Repairnet (Abhinav et al.)[1], and Vulrepair [20].

Namavar et al. perform experiments on code representations by training multiple models and assessing their performance with each representation. Zhu et al. focus on teaching their model a specific aspect of programming languages, type theory. They have the goal of teaching their model to be ‘type aware’. This involves a novel input representation that the model learns during pretraining. We discuss this input representation more in Section 4.3. Abhinav et al. combine the popular method of passing a pair of buggy and repaired code as input with error messages. Their model learns to correlate error messages with specific errors during pretraining.

The following studies avoid re-using existing architectures by pretraining base LSTMs. This differs from other studies which pretrain a previously used architecture (e.g., codeT5) initialized from scratch.

Zirak and Hemati tackle the problem of a huge search space caused by excessive repair templates. They pretrain an LSTM based language model to prioritize patches that appear natural. Ding et al. frame patching as a translation problem, and train an LSTM to predict the next token. Chen at al. focus on the large vocabulary problem and reduce the input vocabulary with a modification of code abstraction. They only abstract the context, leaving the buggy code untouched. Their input representation is then used to train two LSTM models for code generation. One LSTM model is used for encoding and one LSTM model is used for decoding.

Several primary studies perform pretraining followed by task specific finetuning. The workflow of this approach generally involves pretraining on a large dataset to teach the model general information. Then narrowing the general knowledge learned down to a specific downstream task, represented by the finetuning data.
Studies adopting the above methodology are: SeqTrans (Chi et al.) [8], Berabi et al. [3], Drain et al. [16], and Tufano et al. [73]. Chi et al. pretrain their model on raw code bug fix pairs to 'learn fixing experiences'. Berabi et al. pretrain on natural language, before finetuning on a programming language APR dataset with 52 labelled error types. Drain et al. first perform denoising pretraining followed by supervised learning. For pretraining, they treat raw code as text and use span-masking as the pretraining objective. They perform three different experiments: pretraining only, finetuning the pretrained model BART, and pretraining a partially initialized (a warmstart) BART model. Tufano et al. state that pretraining is extremely useful when pretraining dataset is significantly larger than the finetuning one. They pretrain a T5 model using masked language modeling, replaced token detection and sequence to sequence next token prediction as the training objectives.

4.1.2 Finetuning Only

In this section, we discuss the primary studies that rely on finetuning for their code generation. Finetuning makes it possible to adapt a model to a downstream task without retraining it from the scratch, which might require big data or excessive computational resources to perform. Besides this, there are more specific reasons for why studies will perform finetuning and skip pretraining. Here, we attempt categorize the studies into five groups based on those reasons: Dataset Evaluation, Model Evaluation, Transfer Learning, Input Representation, Curriculum Learning.

First, we discuss a study that introduces a new benchmark dataset, FixEval [54]. Once the researchers create their new dataset, they must verify it to determine its reusability. Hence, the authors finetune and test CodeT5 and PLBART models with their newly created dataset.

Second, we have studies that finetune in order to evaluate existing language models on existing benchmark datasets. Studies in this category do not have any innovation in their finetuning methods. They are primarily concerned with applying one or more existing LLMs to automated program repair and evaluating their performance. These studies are: APR GLAD (Kang and Soo) [34], Mashhadi and Hemmati [53], Shi [68],
Jiang et al. [29], and Yang et al. [83]. Within this group, we also find two subcategories: finetune and evaluate models on widely used APR benchmarks, and the models which finetune and evaluate on datasets with a specific error type.

We start with the subcategory of studies which use a dataset consisting of specific error types. Kang and Soo leverage a pretrained gated recurrent unit (GRU) to repair emission faults (bugs where the necessary code is missing). Their GRU is finetuned only on the faulty portions of the code data. Yang et al. attempt to evaluate LLMs at fixing security vulnerabilities. They use the pretrained CodeBERT and graphcodeBERT models with Bug Fix Pairs (BFPs) on a security vulnerability specific dataset.

Next, we have studies which do not use bug specific data for finetuning. Two of them, Jiang et al, and Shi, finetune and evaluate multiple models. Jiang et al. finetune 5 LLMs and Shi finetunes 7, while Mashhadi and Hemmati’s study only finetunes CodeBERT.

Third, several studies studies finetune for transfer learning or to perform curriculum learning. Kim et al. [38] leverage APR and transfer learning for converting the Samsung code base from Java to Kotlin. They do not have sufficient Kotlin data to pretrain a model from scratch. However, there exists several models trained on Java. Because Kotlin is similar to Java, the researchers believe their finetuning will enable the model to transfer its knowledge from Java to Kotlin repair. They perform the transfer learning by finetuning on Samsung own industrial Kotlin projects. Combining transfer learning and Curriculum Learning, Zirak and Hemmatti [91] address domain adaptation problem. That is; they find that existing models do not generalize well. Thus, they use a large amount of bug-fix related PL data to finetune TFix, a model trained only on natural language texts. Their finetuning approach combines full finetuning and curriculum learning. The curriculum orders the training data by difficulty and similarity. The idea behind this curriculum is for the model to work its way up to harder examples. The authors of DEAR [45] also leverage a form of curriculum. They finetune their model with cycle training, where the model is able to see the same bugs being fixed in different ways.

Several primary studies propose novel input representations as part of their finetuning. The authors of MCRRepair (Kim et al.) [36] propose a unique input representation, called a buggy block. They finetune CodeBERT with their constructed ‘buggy
blocks’. Kim et al. [37] finetune two T5 models with the same finetuning approach as TFix. That is, they finetune their models by using labelled error types from the TFix dataset. Paul et al. [61] finetune 4 LLMs with bug-fix pairs (BFPs). The four models are either unimodal or bimodal. For bimodal LLMs (models which can handle code and natural language as input) they provide the BFP and the corresponding code review for finetuning. Lastly, Fu et al., finetune CodeT5 with a novel input representation that addresses the large vocabulary problem. They use two encoders, a word level Clang Tokenizer, and a byte-pair encoder trained on CodeSearchNet[28]. The word level encoding captures semantic information while the other encoder, an absolute positional encoding, captures the positional information of the input tokens. The word level embedding also tackle the large vocabulary problem. Both encodings are then concatenated for finetuning.

4.1.3 Neither Pretraining or Finetuning

Studies which do not pretrain and do not finetune are the most prevalent in our collected studies making up 21 of the 53 primary studies. These studies take advantage of existing, pretrained and/or finetuned LLMs for automated program repair. Despite skipping the pretraining activities, these studies innovate in other areas of APR, such as dataset creation and advanced prompting. Therefore, we do not cover them in detail in this section.

4.2 Input Representation and Prompting

To answer research question 1, we also analyze the collected data relating to input representation and prompting. Users of LLMs must decide how they are going to ask the model a question (prompt), and what input they will provide to the model. In the context of APR, the input will be the code that needs to be fixed, and potentially instructions that provide additional hints. We define input representation as a transformation of the input data that provides the large language models a new perspective of the data. We define prompting as the additional instructions/hints supplied to the model, along with the input to support the generation task.

Input representation techniques are more complex than prompting. They should ensure the input given to the model captures the relevant semantic, and syntactic
information. For natural language, inputs are usually represented as a vector of numbers, where vectors of semantically similar words tend to be close to each other within the vector space. This vector is obtained by an encoding. In the encoder-decoder based models, such vectors are provided by the language models (e.g., BERT). For decoder only models, the input must first be encoded by a separate model. Our selected primary studies use a wide range of input representations. We attempt to provide a general overview of the types of input representations used. We illustrate input representation with an example from Bugsplainer[52]. In Fig 4.1, we see a pair or buggy and bug free version (BFP) of python code. They differ by only a single indent. This subtle difference would be difficult for a model to capture if only raw code is provided as input. Fig 4.2 shows a diffSBT, a type of input representation that was constructed by traversing an abstract syntax tree (AST) of the code. We see that this representation captures the indentation better than the raw code.

The input representation chosen by a study depends on the problem(s) they are trying to solve. Many of the studies can be categorized into three types of input representations (defined later): BPE, AST, Abstraction. These categories are not exclusive; hence, a study which performs abstraction on their input can also use a BPE of an AST. The studies which have these input representations are shown in 4.2. We will discuss each of these categories below.
4.2.1 AST

Language models, intuitively, are extremely good at understanding natural language texts. One downside of natural language is that it is ambiguous: multiple parse trees can be created for the same sentence in a language. Researchers may want to assist a language model by providing a parse tree to the language model as parse trees do an excellent job at representing syntactic structure. Unlike natural languages which are ambiguous, programming languages must be parsed unambiguously, which facilitates automatic parse tree creation. Therefore, the use of parse trees (e.g., AST) has been a popular method for representing source code to the models.

AST or abstract syntax tree is an input that is able to represent the syntactic structure of the source code[86]. As seen in the Fig. 4.2, the diffSBT sequence constructed from the AST is useful for capturing the indentation error in the code. This representation has been exploited in the program repair domain by comparing the AST of buggy and corresponding bug free code. Both DEAR [45], and He et al. [22] leverage representation to derive fine grained AST-based changes between buggy and bug free code. He et al. then uses these differences as feature vectors of input. With
20

this, they give their model a better chance of understanding the difference between buggy and bug free code.

ASTs can be either an intermediate step in obtaining a final input representation, or the input representation themselves. When AST are the input representation being passed to the model, like in Tian et al. [71], they must be encoded before being fed to the model. Besides directly identifying and clearly representing differences between code documents, ASTs are used for obtaining otherwise difficult information to extract from code. These include using AST to: extract buggy lines [66], to extract data-flow dependencies from code [8] or to obtain fix templates from datasets [88].

Zhu et al. [90] provide a prime use case of AST for input representation. They attempt to teach their model about types by incorporating type theory with their input representation. First, by introducing a new context free grammar, they construct an abstract syntax tree. Then, using their new grammar, a 'type graph' is created. The nodes are converted into a pre-order traversal sequence, and the edges are represented as an adjacency matrix. Then, these representations are encoded with an encoder.

4.2.2 Abstraction

Code abstraction is an important method for reducing the vocabulary size in the input. Compared to natural language, the vocabulary of programming languages is unconstrained. This is due to naming conventions like camel case. A large vocabulary is a problem for language models since they will have a higher chance of encountering tokens that they have never seen before. Such a token will either confuse the model or be ignored. When these unseen tokens contribute to variable names and method names that do little to contribute to the functionality of the code, researchers believe they can be abstracted away. Researchers identify abstraction to be a viable input representation for automated program repair with large language models. An example of raw code vs extracted code from Kim et al. [37] can be found in Fig. 4.3.

One study, CoCoNuT, [50] uses abstraction to reduce the vocabulary of their Java dataset from 1,136,767 tokens to just 139,423, which is a 10x reduction.

Abstraction involves different levels of granularity depending on the amount of code provided to the model as input. An extreme example of abstraction is provided by Namavar et al. [57]. They perform code abstraction by adding special tokens.
such as ID and LIT for identifiers and literals respectively. They take the abstraction further by obtaining categories of reusable tokens based on the top-300 most frequent ones. All remaining tokens are then replaced with generics, such as `method` for a method name.

When code abstraction is used, researchers have to decide how much abstraction is necessary to reduce the vocabulary size while retaining enough information from the buggy or bug-free code. Some interesting approaches have been proposed for this specific problem. Kim et al [37] use defect based abstraction, where most keywords are abstracted except for the ones related to the bug in the program. This is intuitive as being able to distinguish between buggy and bug-free variables is important for fixing the buggy code. Other studies, such as SEQUENCER [7] also choose to abstract specific sections of code. They abstract all code in the buggy file except the method containing the bug. This works well for bugs that are isolated to a specific method, but may not work for multi-file or multi-line bugs.

Thus, to summarize, the primary studies use abstraction to create a smaller vocabulary input representation. Their degree of abstraction depends mainly on the
types of bugs being fixed, which define the level of context (code surrounding the bug) required, thus defining the input length.

4.2.3 Byte-Pair Encoding

Other than abstraction, to deal with the large vocabulary problem, several studies use a byte pair encoding (BPE). This is an encoding that splits rare words into smaller sub words before encoding. This is especially useful for dealing with naming conventions in programming languages. For example, the variable pointCounter and numberCounter would be split into point/number+counter. If these variables were encoded before splitting, a model would have a much harder time discovering that these variables are both counter variables.

4.3 Dataset Collection and Construction

In this section, we discuss the datasets used by our primary studies on automated program repair leveraging LLMs. Table 4.3 summarizes the datasets used by each study. In this table, we group the studies by validation dataset used. We observe that most studies rely on the same datasets. They often need to compare performance against other models. If there already exists metrics for other models on certain datasets, the comparison gets easier. To visualize the datasets used together in the same study, we create a graph – Fig. 4.4. Here, the vertices are the datasets, and each edge signifies that datasets are used in the same study. We use a perl script and the python library networkx to design this graph. In this section, we discuss how the primary studies construct and reuse their experimental datasets.

4.3.1 Studies building their own Datasets

Table 4.3 shows primary studies that construct their experimental datasets. Three main approaches were followed to construct a dataset. The most popular approach
is to combine, de-duplicate and occasionally label existing datasets. The second approach is to obtain data from introductory programming courses. The idea behind this approach is similar to the one taken for curriculum learning. By training with basic examples, the researchers attempt to gradually teach the model APR. Introductory programming courses often have these basic examples. Lastly, several studies collect their datasets by scraping open-source repositories. They identify bug fix commits, and extract the BFP corresponding to the code before and after that commit.

4.3.2 Studies Combining Datasets

Several studies use datasets which are combinations of existing datasets. Prenner and Robbes [63] construct an executable dataset by extracting data from CodeNet and the CodeContests dataset [46]. They label each bug into fine grained categories based on difficulty. This dataset would be useful for studies which leverage curriculum learning, e.g., Zirak and Hemmatti [91]. Kechagia et al. [35] extract specific API misuse bugs from existing benchmark datasets: Bears, Bugs.jar and MUBench[2], obtaining a total of 101 diffs and test cases.

4.3.3 Studies using Student Data

This section describes studies that collect data from students, who are learning to program. As described in section 4.3.1, the use of student data enables large language models to gradually work up to harder examples during training. Nakamura et al. [56] collect 21k programs from introductory programming courses. Abhinav et al. [1] collect the code written by students for 93 programming tasks. Yang et al. [82] uses the Prutor dataset[11] for training, which was originally designed as a tutoring system for introductory programming courses.

4.3.4 Studies which Scrape Open Source Repositories

The authors of Tare [90] use an existing dataset of Zhu et al.[89], which was constructed using commits from GitHub. Similar to TARE, Lajkó et al. [41] extract commits and code from GitHub projects. Namavar et al. [57] use code provided by
Raychev et al. [65] to obtain 150,000 JS files for training. Namavar et al. [16] mine 67k Repositories for 1.3B lines of code. Both the authors of GLAD [34] and Kang et al. [33] collect data from 1k OSS Java repos. Meng et al. [55] scrape data from the 2,000 most starred Java GitHub projects. Ding et al. [15] collect data from 10,235 most starred Java repos on GitHub. We observe that there is no magic number of repositories with each study scraping a different amount of them.

A unique study from Kim et al. [38] use their own data. As this study was done with Samsung, they likely have more data than most non-industry researchers. They use their own Kotlin projects, and data scraped from OSS Kotlin projects for fine-tuning. Their final dataset has a total of 20k OSS defects and 60k industrial bugs.

4.4 Summary

In this chapter, we successfully analyzed and grouped the pretraining methods, input representations and datasets used by each of the primary studies. The knowledge gained from this analysis benefits our evaluation of primary studies in Chapter 6.
Table 4.3: Experimental Dataset (Reused)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Study</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ManySSstuBs4j</td>
<td>[53, 48, 66, 71]</td>
<td>Single Statement Java Bugs 73,000 bug-bug free pairs</td>
</tr>
<tr>
<td>QuixBugs</td>
<td>[69, 78, 40, 62]</td>
<td>40 Java problems 40 python problems</td>
</tr>
<tr>
<td>CodeXGLUE[49]</td>
<td>[23, 60, 61] 68, 16, 29, 8]</td>
<td>122k Java Codes (Same as Bugs2Fix and Tufano et al)</td>
</tr>
<tr>
<td>FixJS [9]</td>
<td>[23]</td>
<td>300k Bug-Bug Free Pairs Javascript</td>
</tr>
<tr>
<td>Defects4J[32]</td>
<td>[22, 78, 76], 31, 45, 88, 34], [30, 79, 36], 85, 8, 77]</td>
<td>835 Java Bugs</td>
</tr>
<tr>
<td>Bugs.jar[67]</td>
<td>[22, 72, 44] 71, 85]</td>
<td>1,158 bug-bug free pairs Java</td>
</tr>
<tr>
<td>Bears [51]</td>
<td>[22, 72, 71] 85]</td>
<td>118 Java Bugs</td>
</tr>
<tr>
<td>Reveiw4Repair [26]</td>
<td>[60, 61]</td>
<td>55,060 code review+change pairs</td>
</tr>
<tr>
<td>ManyBugs[42]</td>
<td>[78]</td>
<td>1,183 C defects</td>
</tr>
<tr>
<td>CodeSearchNet[27]</td>
<td>[73, 47, 91]</td>
<td>6M Go, Java, JavaScript, PHP, Python, and Ruby Methods</td>
</tr>
<tr>
<td>Chen et al [7]</td>
<td>[73]</td>
<td>25,578 Diffs</td>
</tr>
<tr>
<td>Repair Them All [17]</td>
<td>[48, 71]</td>
<td>Provides execution framework for Bears, Bugs.jar, Defects4J, QuixBugs and Intro-ClassJava</td>
</tr>
<tr>
<td>BigFix[44]</td>
<td>[45]</td>
<td>26k bugs</td>
</tr>
<tr>
<td>CPatMiner[59]</td>
<td>[45]</td>
<td>17k diffs</td>
</tr>
<tr>
<td>Software Reassurance Dataset[4]</td>
<td>[15]</td>
<td>64,099 bugs</td>
</tr>
<tr>
<td>TFix Dataset[3]</td>
<td>[37]</td>
<td>100k instances of 52 types of errors</td>
</tr>
<tr>
<td>Codeflaws[3]</td>
<td>[50]</td>
<td>3902 bugs</td>
</tr>
<tr>
<td>Bugaid[21]</td>
<td>[50]</td>
<td>219 bug pairs</td>
</tr>
<tr>
<td>Bugs2Fix[21]</td>
<td>[50], [36, 85, 7]</td>
<td>219 bug pairs</td>
</tr>
<tr>
<td>BigVUL[18]</td>
<td>[20]</td>
<td>3754 bugs</td>
</tr>
</tbody>
</table>
Figure 4.4: Graph explaining the datasets used together across multiple studies
Table 4.4: Experimental Dataset (Constructed)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLMDefects [19]</td>
<td>113 problems</td>
<td>LeetCode</td>
</tr>
<tr>
<td>SEQUENCER [7]</td>
<td>25,578 Diffs</td>
<td>Combining existing datasets</td>
</tr>
<tr>
<td>CCTest [47]</td>
<td>23k tests and prompts</td>
<td>extracted from Leetcode and CodeSearchNet</td>
</tr>
<tr>
<td>Lin et al. [48]</td>
<td>50k patches</td>
<td>combines multiple existing datasets, and removes duplicates</td>
</tr>
<tr>
<td>CURE [31]</td>
<td>4M methods</td>
<td>1,700 OSS Java Projects + CoCoNuTs training data</td>
</tr>
<tr>
<td>BigFix (DLFix) [44]</td>
<td>26k bugs</td>
<td>from 8 OSS projects</td>
</tr>
<tr>
<td>FixEval [54]</td>
<td>700 programming</td>
<td>from participants solving programming competitions (Java and Python)</td>
</tr>
<tr>
<td>HumanEvalJava [29]</td>
<td>over 200GB of code</td>
<td>Obtained from existing datasets</td>
</tr>
</tbody>
</table>
Chapter 5

RQ2: Which evaluation methods are used for LLM based APR?

In this chapter, we discuss the use of certain evaluation metrics in our primary studies. We find accuracy to be the most common evaluation metric and discuss why this is the case. We also discuss the alternative evaluation metrics and why researchers may choose to use them.

5.1 Overview of validation dataset metrics

The majority of studies use the % of correct patches generated by a model, on the evaluation dataset. Many others use accuracy based metrics (metrics that use correct bug fixes as the evaluation method) such as accuracy@k. Few studies go beyond accuracy, and use traditional metrics such as: F1-score, recall, precision. Table 5.1 shows a summary of the evaluation metrics used. In this section, we discuss how the primary studies adopt various metrics for their evaluation and validation.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Correct Patches (49 total)</td>
<td>[69, 23] [60, 78, 40][19, 73, 62, 80] [76, 31, 45] [61, 56, 35][68, 54, 66, 88][16, 34, 29, 91][72, 33, 15][83, 1, 90, 55][41, 57, 38] [53, 82, 19, 50] [44, 71, 30] [79, 36, 85] [87, 8, 7, 77] [3, 20]</td>
</tr>
<tr>
<td>NON accuracy</td>
<td>[60, 47, 48] [61, 68, 54] [37, 41, 57]</td>
</tr>
</tbody>
</table>

A few researchers use ‘time to solve’ as an evaluation metric. This is a questionable metric because the performance between graphics cards varies significantly. Studies using this metric are Lutellier et al. [50], and Kechagia et al. [35].
Other metrics used are %compilable patches, plausible patches and plausible fixes. A patch or fix is considered plausible if it passes all test cases, but does not exactly match the ground truth. Studies employing these metrics are Wei et al. [76] and Zhu et al. [90] score a model's ability to almost generate correct patches.

Lastly, studies which primarily use accuracy on benchmark datasets as their evaluation metric may consider accuracy for specific categories of bugs. For example, the authors of DEAR [45], use 5 categories of bugs which are combinations of single hunk or multi hunk bugs and single statement and multi statement bugs. Each of these categories requires a different level of context and knowledge of a buggy program to solve, therefore, the researchers use separate accuracy calculations for each category.

Interestingly, 49 out of 53 studies use accuracy as their evaluation metric. The remaining three using evaluation metrics are Kim et al. [37], Li et al. [47] and Lin et al. [48]. Lin et al. do not use accuracy. Their program repair tool is one that assists other repair tools by testing if solution is correct. This tool is only useful when there is not already test suites. Without using test suites, the model needs to guess if a solution is correct. Using alternative metrics captures how well their tool can guess correct solutions. CCTest uses Levenstien edit distance in the AST to score mutation of input. That is, counting the number of edge and node insert/delete operations it takes to go from one AST to the other. In practice, the tree must first be converted to list representation for this to take place. CCTest employs TP, FN, precision, recall and F1-score to test CCTests ability in identify defects. They also use BLEU score to test CCTests effect on existing APR tools. Similarly, Lin et al. also build a tool for assessing the correctness of generated patches. They use traditional accuracy, F1-score, precision and recall to evaluate the classification ability of their tool.

As with the above two studies, Kim et al. [37] do not use the common % correct patches metric. However, they are evaluating models at a generation task. They are outliers for not using accuracy as an evaluation metric as the 49 other studies focusing on generation do use accuracy. The authors instead use BLEU score. They are interested in not patch correctness, but if the generated output 'contains fix ingredients'. Using accuracy a binary metric such as accuracy to determine this would not be ideal. BLEU score captures similarity, and a code snippet containing the right
fix ingredients should be similar to the ground truth. Thus BLEU score is used, as it can determine if generated outputs contain similar words to expected output. Paul et al. [37] also use BLEU to determine if repair ingredients are in generated code from their defect specific abstraction input technique.

5.1.1 APR evaluation assisting studies

In this section, we discuss the primary studies that use non-accuracy evaluation metrics to determine the correctness of LLMs outputs. Lin et al.[48] uses accuracy, precision, recall and F1-score for their evaluation. They also test its performance of classifying patches generated by multiple APR tools, with f1 score range between 0.7 and 0.9 across 20 models. Using the same metrics, He et al, [22] evaluates an over-fitting patch classifier that uses APR methods.

5.1.2 Accuracy and Non-Accuracy Evaluation Metrics

Many primary studies employ accuracy and other means for evaluating their generative models’ outputs. Paul et al. [61], Shi [68], and Namavar et al.[57] use both BLEU and/or codeBLEU and accuracy. Besides accuracy, the authors of FixEval [54] use exact match, syntax match, dataflow match, codeBLEU and compilation accuracy, as well as correct patches/total patches.

A few studies adopt a unique evaluation metric in their experiments. Lajkó et al. [41] uses ED-k: edit distance within k edits. This metric quantifies how close a model was to the correct output by counting the number of insertion/deletion operations it takes to go from the generated output to the ground truth. Lastly, Yang et al. [82] uses MRR, mean average precision. This is due to their beam search technique where their model will generate multiple possible outputs. The researchers leverage these mean evaluation metrics as their model generated multiple outputs at once using a beam size greater than 1.

5.2 Summary

In this chapter, we determined the most popular evaluation metric used by the primary studies. We find that the majority of studies use accuracy as their evaluation
metric. We also find interesting use cases of non-accuracy evaluation metrics which enable more fine grained feedback.
Chapter 6

RQ3: What are the Strengths, Weaknesses and Future Direction for APR

In this chapter, we discuss the strengths and weaknesses of the studies we reviewed. Then, we attempt to build a road map for future work on automated program repair leveraging LLMs.

6.1 Strengths

The strengths of the primary studies stem from several aspects that are ‘model agnostic’. These aspects are datasets, input representations and reward functions. Model agnosticism is key for the strengths of studies, as a main weakness is the amount of hardware required to run state-of-the-art language models. Devising APR techniques that are ‘language model agnostic’ will benefit this research area, regardless of computing power/parameter increase. Each of the three identified strengths is discussed below:

6.1.1 Datasets

The most obvious model-agnostic feature of these studies is the dataset. The benchmark datasets used by the primary studies are well crafted. When constructing a benchmark dataset, it should be made reusable by other studies to enable result comparison. According to our dataset graph Fig. 4.4, many studies use a combination of multiple datasets. When studies create a new dataset, it is most often based on existing datasets. These new datasets have duplicates removed, and occasionally are combined with extra information that includes more detailed diffs, relevant context, difficulty labels, and test cases. Such a dataset enhancement can greatly benefit the future studies. Furthermore, we find studies are building datasets which include test suites. This benefits with the popular method of evaluating the models accuracy on
fixing the bugs. These kind of datasets enable evaluating the model output with accuracy. In this way, researchers can avoid using BLEU or CodeBLEU, and ensure their models are tested accurately.

### 6.1.2 Reinforcement Learning

Another model-agnostic strength of these studies is the innovation in training/finetuning objectives. Since language models were originally crafted for natural language, the loss function used by most models is not well suited to code. For automated program repair, weight updates should be based on the model producing a correct output. That is, the generated code should be tested against a test suite. Utilizing a reward function based on execution and passing test cases is more intuitive than traditional similarity based loss functions such as BLEU. We see that the use of test suites is somewhat uncommon; most studies do not modify the loss function. The lack of appropriate datasets with corresponding test cases may be limiting the adoption of this approach. However, recent studies are publishing or modifying APR datasets to include test suites, (check table 4.4 for details).

We expect to see this kind of reward function becoming mainstream in future APR approaches. Despite the increase in training time, a reward function based on test cases is much more aligned with for training APR models with LLM.

### 6.1.3 Input Representations

The diversity of input representations can be considered as a strength of the primary studies. First, the chosen input representation can help a language model better learn specific characteristics of the code to better support an APR task. As in Figure 4.2, the choice of input representation can make certain code features stand out to the model. The right choice of input representation can also drastically reduce the vocabulary size, reducing the cost of training time and the size of the model.

Second, input representations offer code generation an advantage over natural language generation. Since natural language text is often ambiguous, it is extremely difficult to obtain a parse tree of natural language. On the other hand, source code is semi-structured, allowing for fair representation with the abstract syntax trees (AST). The AST gives a language model valuable information about the structure of the code.
which in turn helps it produce better outputs.

6.2 Weaknesses

An exact method for leveraging large language models on program repair is still unknown. What we do know, is a model with more parameters will have a larger corpus to extract general knowledge from. Unfortunately, obtaining a pretrained model with a large number of parameters is very expensive. For example, Meta has trained a 70B parameter model with 16 bit precision. The memory required to store 70B parameters is 1.12 terabytes, which is not available in commodity hardware (e.g., personal computers). Furthermore, it took 6000 GPUs and 12 days to obtain the 70B parameter weights. This indicates a significant cost, which leads researchers to only perform finetuning. In this section, we discuss a few limitations of the primary studies.

6.2.1 Lack of Extensive Pretraining

Pretraining a language model provides it with extensive general knowledge which can be honed to a specific task during finetuning. Most existing pretrained language models are trained mostly on natural language. The most common pretraining techniques attempt to predict the next tokens, which is better suited to natural language. Therefore, the majority of the existing models might be more suitable for text generation rather than code generation. To obtain more powerful LLMs specifically for code generation, pre-training on large-scale codebase is warranted. This indicates that, a significant amount of financial resources must be allocated to GPU hours, which could be costly.

6.2.2 Lack of Certainty

While input representation techniques improve model output, it is difficult to know which representation will work better before designing the model. Deep learning models are not inherently explainable. That is, nobody knows exactly how these large language models work in a granular level. Please note that there is a loss function, and weights can be tweaked to minimize the error using gradients. However, it is difficult to explain why tweaking billions of parameters leads to a better understanding of
semantics by the LLMs. The collected studies are largely experimental with little or no grounded theory that ensures a particular approach is more effective for code generation. They often base their design on existing approaches that obtain good results. Such a justification might not be always sufficient. This kind of use also exacerbates the hardware problem.

6.3 Future Direction

Based on the strengths and weaknesses of current approaches, we suggest the following as the future direction for APR research that leverages LLMs.

6.3.1 Advanced Reward Functions

The error function of a machine learning model determines weight updates. If researchers use a loss function that is based on semantic similarity for APR tools, their tool will not be effective. During training, the model could be rewarded for code that does not compile due to a single character miss. This is because semantic similarity metrics do not actually compile or test the code. This code is incorrect but it is only 1 character away from the ground truth. Therefore, using an error function based on a test suite is clearly the best solution. It may make the training more expensive, but the weight might be updated correctly.

In fact, the more fine grained information that gets incorporated into the error function the better a chance the model will have to learn to minimize the error. This is easier said than done. It will be difficult, but necessary for designing an effective loss function targeting code generation. This leads us to the next section of future work.

6.3.2 Dataset Curation

Currently, there exists a number of popular benchmark datasets that are used in APR research.

However, APR datasets do not have important meta information such as test cases.
A few datasets might contain test cases, but they are not sufficient for large-scale finetuning. An area of future work would be curating bigger and better datasets with sufficient meta information (e.g., test cases) and designing advanced reward functions.

6.3.3 Expanding Input Representation

Our primary studies have diversity in their input representations. Future work should continue experimenting with input representations. As language models become more powerful and can handle larger inputs, combining different forms of input representations is likely to benefit the performance of LLM based APR tools.

6.4 Summary

In this chapter, we leveraged our developed understanding APR that leverages LLMs to determine strengths and weaknesses of our primary studies. From there, we were able to determine some directions for future research on APR that relies on LLMs.
Chapter 7

Conclusion

Software bugs take up to 50% of developers’ time. As mentioned, at this point in time, endemic issues of APR make a general APR tool out of reach. Despite the uncontested generational ability of LLMs, end to end solutions for APR are insufficient. We have seen most primary studies improve on existing methods by targeting one aspect of state of the art approaches.

In this systematic review, we examined the datasets, input representations, pretraining methods and evaluation metrics used by 53 state of the art APR studies. We find that research is becoming more reliant on pretrained language models. We see that studies rarely attempt to create an end-to-end LLM based APR tool. There is more focus on specific aspects of program repair such as input representation, dataset curation and PL specific error functions. The focus on these specific aspects is likely due to the state of hardware. We currently do not have models with a large enough context window to combine multiple input representations. Although innovative input representations (abstraction and byte pair encoding) can reduce the vocabulary size of the input language, researchers still need to compromise on the amount of additional context or input representations.

The steady increase in hardware capacity, excitement in academia/industry have led to rapid progress in generative AI and LLMs. Future work should focus on better inputs, error functions and datasets.

They should also focus on perfecting specific aspects of LLM based program repair rather than re-inventing the wheel.
Bibliography


Appendix A

A.1 Project Repository

https://git.cs.dal.ca/callumm/apr-llm-litrev-replication