UNIVERSITY OF CALIFORNIA SAN DIEGO

Neurosymbolic Tools for Effective Coding and Debugging

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Georgios Sakkas

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Loris D'Antoni
Professor Philip Guo
Professor Sorin Lerner

2024

The dissertation of Georgios Sakkas is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

DEDICATION

To my parents, Maria and Panagiotis, and my brother, Vasilis, your unwavering love and support have been my foundation throughout this long and challenging journey. From the very beginning, you instilled in me the values of hard work, perseverance, and the belief that I could achieve whatever I set my mind to. You've always been there through every success and setback, offering guidance and encouragement. Words cannot fully express how grateful I am for everything you've done to help me get to this point. This accomplishment is as much yours as it is mine.

To my friends from Greece (Vasilis, Thodoris, Giannis, Giannis, Giorgos, Thymios, Anastasis, Vasilis, Rafail, Thomas, Kostas, Stavros, Nikos, Giannis, Dimitris, Tasos), you have been a source of endless joy and comfort, no matter the distance. Even though we've been apart, the good times we've shared and the support you've given me from afar have meant the world. To my friends in San Diego and across the U.S.A (Dimitris, Eleni, Thodoris, Lydia, Giannis, Maria, Petros, Marialena, Nikos, Nana-Ama, Giannis, Giorgos, Chris, Jacob, Gen, Isaak, Fritz, Panagiota), you've made this chapter of my life so much richer. Whether through study sessions, conversations, or moments of laughter, you've been there to help me keep going and to remind me to enjoy the process along the way.

To my wonderful partner, Eleni, for the fun, adventure, and happiness you've brought into my life. From our time together here in San Diego to our travels and explorations, you've shown me a world full of new experiences. You've helped me grow as a person and made me a better version of myself, and for that, I'm endlessly thankful.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Ranjit Jhala. Your guidance, wisdom, and mentorship have shaped my journey as a researcher and as a person. From the very beginning, you provided me with opportunities that allowed me to grow and pursue my passions, not just within this dissertation but also in my career. I am incredibly grateful for the trust and confidence you placed in me, as well as for the countless insightful discussions that have helped me navigate both research challenges and professional opportunities.

I would also like to extend my sincere thanks to Prof. Westley Weimer. Your unwavering support throughout my PhD has been invaluable. From helping to rewrite papers and refine ideas to preparing for presentations, you have always offered the kind of feedback that sharpened my work and made it better. I truly appreciate your time, effort, and commitment to helping me become a better researcher and communicator.

A special thank you goes to my co-authors and collaborators, Madeline Edres and Benjamin Cosman. Our time working together during your own PhD journeys (now both Professors), has been one of the most rewarding parts of my experience. The support, knowledge, and camaraderie we shared during long hours of research and writing papers has meant a great deal to me. I'm incredibly proud of what we accomplished together.

I also want to extend my appreciation to the rest of my PhD committee: Prof. Loris D'Antoni, Prof. Sorin Lerner, and Prof. Phillip Guo and ex-committee member Prof. Nadia Polikarpova. Your feedback, insightful comments, and fresh perspectives have been invaluable in shaping this dissertation. Each of you brought unique expertise and ideas that pushed me to think more deeply and critically about my work. I am grateful for your time, support, and commitment to my success throughout this process.

Finally, I would like to thank everyone else who has contributed to this journey in one way or another. This dissertation would not have been possible without the support, collaboration,

and guidance of so many people along the way.

Chapter 2, in part, is a reprint of the material as it appears in the Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, Ranjit Jhala, PLDI 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in the Proceedings of the ACM on Programming Languages, Volume 6 (OOPSLA2). Georgios Sakkas, Madeline Endres, Philip J Guo, Westley Weimer, Ranjit Jhala, SPLASH 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it will appear in the Proceedings of the 2025 IEEE/ACM 46th International Conference on Software Engineering. Georgios Sakkas, Pratyush Sahu, Kyeling Ong, Ranjit Jhala, ICSE 2025. The dissertation author was the primary investigator and author of this paper.

VITA

2018	B. S. in Electrical and Computer Engineering, National Technical University of Athens, Greece
2022	M. S. in Computer Science, University of California San Diego, U.S.A.
2024	Ph. D. in Computer Science, University of California San Diego, U.S.A.

PUBLICATIONS

Georgios Sakkas, Pratyush Sahu, Kyeling Ong, Ranjit Jhala, "Neurosymbolic Modular Refinement Type Inference", *Accepted at ICSE*, 2025.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madan Musuvathi, Shuvendu Lahiri, "Exploring the Effectiveness of LLM based Test-driven Interactive Code Generation: User Study and Empirical Evaluation", *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 390-391, 2024.

Georgios Sakkas, Madeline Endres, Philip J Guo, Westley Weimer, Ranjit Jhala, "Seq2Parse: Neurosymbolic Parse Error Repair", *Proceedings of the ACM on Programming Languages, Volume 6 (OOPSLA2)*, 1180-1206, 2022.

Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, Ranjit Jhala, "Type Error Feedback via Analytic Program Repair", *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 16-30, 2020.

ABSTRACT OF THE DISSERTATION

Neurosymbolic Tools for Effective Coding and Debugging

by

Georgios Sakkas

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Ranjit Jhala, Chair

This dissertation presents neurosymbolic approaches for developing tools that enhance programming and debugging by combining symbolic reasoning with machine learning techniques. As modern programming languages grow more complex, the need for automated tools that can efficiently identify and fix errors becomes more critical. By integrating traditional program analysis methods with the predictive power of machine learning models, neurosymbolic approaches offer a robust solution for automated program repair and synthesis. This work focuses on creating tools that target common errors in OCaml and Python, and aiming to reduce manual intervention in Haskell program verification, while improving the accuracy and efficiency of error detection and correction.

The first contribution of this research is RITE, a tool that provides type error feedback in OCaml programs through a data-driven approach to program repair. RITE uses a training dataset of ill-typed programs and their fixes to predict and generate repairs for new errors. The second contribution is SEQ2PARSE, a neurosymbolic tool that addresses syntax errors in Python by combining neural sequence models with symbolic error-correcting parsers. This hybrid method can efficiently pinpoint relevant corrections and generate accurate fixes. Lastly, this dissertation introduces LHC, a tool that uses large language models (LLMs) to automatically generate refinement type annotations in Haskell programs. LHC drastically reduces the time and expertise needed to perform formal verification by leveraging LLMs and symbolic refinement type checking.

Each tool demonstrates the effectiveness of neurosymbolic approaches in simplifying the programming and debugging process. Evaluations show that these methods not only improve the accuracy of repairs but also provide users with clear and useful feedback. This work concludes with an exploration of future directions for neurosymbolic tools, particularly their potential to scale automated program repair techniques across different programming languages and development environments. These findings highlight the promise of neurosymbolic methods in optimizing software development and improving the overall efficiency of programming tasks.

Chapter 1

Introduction

In today's fast-paced software industry, the demand for reliable, efficient, and scalable software is at an all-time high [1, 14, 49, 63, 71, 129]. However, as software systems grow in complexity, so does the difficulty of ensuring their correctness [1, 18, 22, 42, 63, 94, 112, 124]. Large-scale projects often involve multiple developers working on millions of lines of code, increasing the likelihood of errors that are time-consuming to detect and fix. For example, a type error in a statically typed language like OCaml might ripple through a codebase, leading to cascading failures if not promptly addressed. Similarly, a single syntax error in a dynamically typed language like Python can prevent an entire application from running. Traditional symbolic debugging tools, though powerful, struggle to keep pace with the scale and complexity of modern software development, making the development of ***modern automated programming and debugging tools*** essential.

Formal verification offer a rigorous approach to ensuring program correctness, addressing many limitations of manual testing and traditional debugging methods. Program verification tools, for example, enable the detection of a plethora of bugs at compile-time, significantly reducing the reliance on runtime or testing-phase error discovery. However, their complexity often makes them inaccessible to non-experts, limiting their widespread adoption in the software industry. For

instance, utilizing a refinement type system in Haskell [126, 135] in order to prove correctness properties of a codebase's functions, requires deep expertise and significant manual effort, limiting their adoption in everyday development. Automated tools [1, 18, 22, 42, 63, 94, 112, 124] that assist in error detection, correction, and formal verification can bridge this gap, making high-assurance software development more accessible.

A promising direction to overcome these limitations is the integration of *machine learning models* into traditional symbolic methods. Machine learning models trained on large code datasets, often augmented by natural language data, can predict error locations and suggest appropriate fixes through supervised learning techniques. Recent advances in combining machine learning with symbolic reasoning have resulted in tools that not only reduce the manual effort associated with programming and debugging but also minimize the risk of human error, ultimately improving the software development lifecycle [24, 29, 39, 41, 51, 77, 82, 99, 130]. hese hybrid approaches leverage the predictive power of machine learning with the rigorous, logical precision of symbolic reasoning to create robust error detection and correction systems.

This dissertation seeks to address the challenges posed by manual debugging and formal verification by developing ***automated neurosymbolic tools*** that enhance various aspects of programming, from type error feedback to syntax error correction and formal verification. Specifically, this dissertation investigates methods for addressing errors in programs written in languages such as OCAML and Python, and augmenting program verification in Haskell. By integrating machine learning models with symbolic reasoning, we aim to streamline error detection, provide clearer feedback, and automate much of the error-correction process, significantly reducing the manual intervention typically required.

One example of this work is the development of **RITE**, a tool for providing type error feedback via analytic program repair. RITE is based on the idea that similar errors often require similar repairs. Using a dataset of ill-typed programs and their corrected versions, RITE learns from past errors to generate candidate repair templates, predict the most suitable template for a

new error, and synthesize a concrete repair.

Another significant contribution is **SEQ2PARSE**, a neurosymbolic approach to fixing syntax errors in Python programs. Traditional symbolic parsers can generate repairs, but they often struggle with the overwhelming number of irrelevant error-correction rules. Neural approaches, while capable of identifying relevant patterns, are prone to generating inaccurate or overly broad fixes. SEQ2PARSE combines the best of both worlds by using neural models to pinpoint relevant error-correction rules, which are then applied by a symbolic parser.

Finally, this dissertation introduces **LHC**, a neurosymbolic tool for automating refinement type inference in Haskell. Refinement types are a powerful method for verifying correctness properties in programs, but they require developers to write detailed annotations for each function in their code, which can be a highly time-consuming task. LHC uses large language models (LLMs) and heuristic-based search algorithm to automatically generate these annotations, drastically reducing the time and expertise needed to perform formal verification.

Throughout this dissertation, we explore how neurosymbolic techniques can be applied to various aspects of program repair and debugging. By integrating machine learning with traditional programming language methods, we aim to create tools that are not only effective at fixing errors but also provide developers with insightful and actionable feedback. These contributions represent significant steps toward automating many of the tedious and error-prone aspects of software development, helping to optimize the programming and debugging process for developers working in diverse environments.

The rest of this dissertation is structured as presented in the following Overview.

Overview

Chapter 2: Type Error Feedback via Analytic Program Repair

This chapter presents *Analytic Program Repair*, a data-driven strategy for providing feedback for type-errors via repairs for the erroneous program. Our strategy is based on insight that *similar errors have similar repairs*. Thus, we show how to use a training dataset of pairs of ill-typed programs and their fixed versions to: (1) *learn* a collection of candidate repair templates by abstracting and partitioning the edits made in the training set into a representative set of templates; (2) *predict* the appropriate template from a given error, by training multi-class classifiers on the repair templates used in the training set; (3) *synthesize* a concrete repair from the template by enumerating and ranking correct (*e.g.* well-typed) terms matching the predicted template. We have implemented our approach in RITE: a type error reporting tool for OCAML programs. This chapter also presents an evaluation of the *accuracy* and *efficiency* of RITE on a corpus of 4,500 ill-typed OCAML programs drawn from two instances of an introductory programming course, and a user-study of the *quality* of the generated error messages that shows the locations and final repair quality to be better than the state-of-the-art tool in a statistically-significant manner.

Chapter 3: SEQ2PARSE: Neurosymbolic Parse Error Repair

This chapter presents SEQ2PARSE, a language-agnostic neurosymbolic approach to automatically repairing parse errors. SEQ2PARSE is based on the insight that *Symbolic* Error Correcting (EC) Parsers can, in principle, synthesize repairs, but, in practice, are overwhelmed by the many error-correction rules that are not *relevant* to the particular program that requires repair. In contrast, *Neural* approaches are fooled by the large space of possible sequence level edits, but can precisely pinpoint the set of EC-rules that *are* relevant to a particular program. We show how to combine their complementary strengths by using neural methods to train a sequence classifier that predicts the small set of relevant EC-rules for an ill-parsed program, after which, the

symbolic EC-parsing algorithm can make short work of generating useful repairs. Additionally, in this chapter, we train and evaluate our method on a dataset of 1,100,000 Python programs, and show that SEQ2PARSE is *accurate* and *efficient*: it can parse 94% of our tests within 2.1 seconds, while generating the exact user fix in 1 out 3 of the cases; and *useful*: humans perceive both SEQ2PARSE-generated error locations and repairs to be almost as good as human-generated ones in a statistically-significant manner.

Chapter 4: Neurosymbolic Modular Refinement Type Inference

This chapter presents LHC, a neurosymbolic agent that uses LLMs to automatically generate refinement type annotations for all the functions in an entire package or module, using the refinement type checker LIQUIDHASKELL as an oracle to verify the correctness of the generated specifications. Refinement types, a type-based generalization of Floyd-Hoare logics, are an expressive and modular means of statically ensuring a wide variety of correctness, safety, and security properties of software. However, their expressiveness and modularity means that to use them, a developer must laboriously *annotate* all the functions in their code with potentially complex type specifications that specify the contract for that function. This chapter showcases a dataset of three Haskell packages where refinement types are used to enforce a variety of correctness properties from data structure invariants to low-level memory safety and use this dataset to evaluate LHC. Previously these packages required expert users several days to weeks to annotate with refinement types. Our evaluation shows that even when using relatively smaller models like the 3 billion parameter StarCoder LLM, by fine-tuning it and carefully chosen contexts, our neurosymbolic agent generates refinement types for up to 94% of the functions across entire libraries automatically in just a few hours, thereby showing that LLMs can drastically shrink the human effort needed to use formal verification.

Chapter 5: Conclusion and Future Work

The conclusion summarizes this dissertation that demonstrated the power of neurosymbolic approaches in enhancing programming and debugging tools through the development of RITE, SEQ2PARSE, and LHC. Each of these tools combines machine learning and symbolic reasoning to automate program repair and verification across OCaml, Python, and Haskell, achieving high accuracy and efficiency while offering valuable insights to developers. Future work includes broadening these methods to support additional programming languages, more complex error types, and adaptive feedback for novice programmers. Additionally, there is potential to enhance neurosymbolic techniques through semi-supervised learning or reinforcement learning, aiming to further reduce the need for extensive labeled data and expand the capabilities of automated program assistance.

Chapter 2

Type Error Feedback via Analytic Program Repair

2.1 Introduction

Languages with Hindley-Milner style, unification-based inference offer the benefits of static typing with minimal annotation overhead. The catch, however, is that programmers must first ascend the steep learning curve associated with understanding the *error messages* produced by the compiler. While *experts* can, usually, readily decipher the errors, and view them as invaluable aids to program development and refactoring, *novices* are typically left quite befuddled and frustrated, without a clear idea of *what* the problem is [129]. Owing to the importance of the problem, several authors have proposed methods to help debug type errors, typically, by *slicing* down the program to the problematic locations [42, 95], by *enumerating* possible causes [18, 69], or by *ranking* the possible locations using MAX-SAT [89], Bayesian [137] or statistical analysis [112]. While valuable, these approaches at best help localize the problem but students are still left in the dark about how to *fix* their code.

Repairs as Feedback. Several recent papers have proposed an inspiring new line of attack on the feedback problem: using techniques from synthesis to provide feedback in the form of *repairs* that students can apply to improve their code. These repairs can be found by symbolically searching a space of candidate programs circumscribed by an expert-defined repair model [46, 114]. However, for type errors, the space of candidate repairs is massive. It is quite unclear whether a small set of repair models *exists* or even if it does, what it *looks like*. More importantly, to scale, it is essential that we remove the requirement that an expert carefully curate some set of candidate repairs.

Alternately, we can generate repairs via the observation that *similar programs* have similar repairs, *i.e.* by calculating "diffs" from the student's solution to the "closest" *correct* program [39, 130]. However, this approach requires a corpus of similar programs, whose syntax trees or execution traces can be used to match each incorrect program with a "correct" version that is used to provide feedback. Programs with static type errors have no execution traces. More importantly, we desire a means to generate feedback for *new* programs that novices write, and

hence cannot rely on matching against some (existing) correct program.

Analytic Program Repair. In this work, we present a novel error repair strategy called *Analytic Program Repair* that uses supervised learning instead of manually crafted repair models or matching against a corpus of correct code. Our strategy is based on the key insight that *similar errors* have similar repairs and realizes this insight by using a training dataset of pairs of ill-typed programs and their fixed versions to: (1) *learn* a collection of candidate repair templates by abstracting and partitioning the edits made in the training set into a representative set of templates; (2) *predict* the appropriate template from a given error, by training multi-class classifiers on the repair templates used in the training set; (3) *synthesize* a concrete repair from the template by enumerating and ranking correct (*e.g.* well-typed) terms matching the predicted template, thereby, generating a fix for a candidate program. Critically, we show how to perform the crucial abstraction from a particular *program* to an abstract *error* by representing programs via *bag-of-abstracted-terms* (BOAT) *i.e.* as numeric vectors of syntactic and semantic features [110]. This abstraction lets us train predictors over high-level code features, *i.e.* to learn correlations between features that cause errors and their corresponding repairs, allowing the analytic approach to generalize beyond matching against existing programs.

RITE. We have implemented our approach in RITE: a type error reporting tool for OCAML programs. We train (and evaluate) RITE on a set of over 4,500 ill-typed OCAML programs drawn from two years of an introductory programming course. Given a new ill-typed program, RITE generates a list of potential solutions ranked by likelihood and an *edit-distance* metric. We evaluate RITE in several ways. First, we measure its *accuracy*: we show that RITE correctly predicts the right repair template 69% of the time when considering the top three templates and surpasses 80% when we consider the top six. Second, we measure its *efficiency*: we show that RITE is able to synthesize a concrete repair within 20 seconds 70% of the time. Finally, we measure the *quality* of the generated messages via a user study with 29 participants and show

that humans perceive both RITE's edit locations and final repair quality to be better than those produced by SEMINAL, a state-of-the-art OCaml repair tool [69] in a statistically-significant manner.

2.2 Overview

We begin with an overview of our approach to suggesting fixes for faulty programs by learning from the processes novice programmers follow to fix errors in their programs.

```
1 let rec mulByDigit i l =
2   match l with
3   | []     -> []
4   | hd::tl -> (hd * i) @ mulByDigit i tl
```

```
1 let rec mulByDigit i l =
2   match l with
3   | []     -> []
4   | hd::tl -> [hd * i] @ mulByDigit i tl
```

Figure 2.1: (top) An ill-typed OCAML program that should multiply each element of a list by an integer. (bottom) The fixed version by the student.

The Problem. Consider the program `mulByDigit` shown at the top of Figure 2.1, written by a student in an undergraduate Programming course. The program is meant to multiply all the numbers in a list with an integer digit. The student accidentally misuses the list append operator (`@`), applying it to a number and a list rather than two lists. Novice students who are still building a mental model of how the type checker works are often perplexed by the compiler's error message [83]. Hence a novice will often take a long time to arrive at a suitable fix, such as the one shown at the bottom of Figure 2.1, where `@` is used with a singleton list containing the multiplication of the head `hd` and `i`. Our goal is to use historical data of how programmers have fixed similar errors in their programs to automatically and rapidly guide novices to come up with candidate solutions like the one above.

Solution: Analytic Program Repair. One approach is to view the search for candidate repairs as a synthesis problem: synthesize a (small) set of edits to the program that yields a good (*e.g.* type-correct) one. The key challenge is to ensure that synthesis is *tractable* by restricting the repairs to an efficiently searchable space, and yet *precise* so the search does not miss the right

fixes for an erroneous program. In this work, we present a novel strategy called *Analytic Program Repair* which enables tractable and precise search by decomposing the problem into three steps: First, *learn* a set of widely used *fix templates*. Second, *predict*, for each erroneous program, the correct fix template to apply. Third, *synthesize* candidate repairs from the predicted template. In the remainder of this section, we give a high-level overview of our approach by describing how to:

1. Represent fixes abstractly via *fix templates* (§ 2.2.1),
2. Acquire a *training set* of labeled ill-typed programs and fixes (§ 2.2.2),
3. Learn a small set of candidate fix templates by *partitioning* the training set (§ 2.2.3),
4. Predict the appropriate template to apply by training a *multi-class classifier* from the training set (§ 2.2.4), and
5. Synthesize fixes by enumerating and checking terms from the predicted templates to give the programmer localized feedback (§ 2.2.5).

2.2.1 Representing Fixes

Our notion of a fix is defined as a *replacement* of an existing expression with a new *candidate* expression at a specific program location. For example, the `mulByDigit` program is fixed by replacing (hd $*$ i) with the expression [hd $*$ i] on line 4. We focus on AST-level replacements as they are compact yet expressive enough to represent fixes.

Generic Abstract Syntax Trees. We represent the different possible candidate expressions via abstract fix templates called *Generic Abstract Syntax Trees* (GAST) which each correspond to many possible expressions. GASTs are obtained from concrete ASTs in two steps. First, we abstract concrete variable, function, and operator names. Next, we prune GASTs at a certain

depth d to keep only the top-level changes of the fix. Pruned sub-trees are replaced with *holes*, which can represent *any* possible expression in our language.

Together, these steps ensure that GASTs only contain information about a fix's *structure* rather than the specific changes in variables and functions. For example, the fix [hd $*$ i] in the `mulByDigit` example is represented by the GAST of the expression [_ $\oplus$ _], where variables hd and i are abstracted into holes (*e.g.* by pruning the GAST at a depth $d = 2$) and $*$ is represented by an abstract binary operator $\oplus$. Our approach is similar to that of Lerner *et al.* [69], where AST-level modifications are used, however, our proposed GASTs represent more abstract fix schemas.

2.2.2 Acquiring a Fix-Labeled Training Set

Previous work has used experts to create a set of ill-typed programs and their fixed versions [69, 73], or to manually create *fix templates* [56] that can yield *repair patches* [78, 79]. These approaches are hard to scale up to yield datasets suitable for machine learning. Also, they do not discover the *frequency* in practice of particular classes of novice mistakes and their fixes. In contrast, we show that such fix templates can be *learned* from a large, automatically constructed training set of ill-typed programs labeled with their repairs. Fixes in our dataset are represented as the ASTs of the expressions that students changed in the ill-typed program to transform it into the correct solution.

Interaction Traces. Following [112], we extract a labeled dataset of erroneous programs and their fixed versions from *interaction traces*. Usually students write several versions of their programs until they reach the correct solution for a programming assignment. An instrumented compiler is used to capture such sequences (or *traces*) of student programs. The first type-correct solution in this sequence of attempts is considered to be the fixed version of all the previous ones and thus a pair for each of them is added to the dataset. For each program pair, we then produce

a *diff* of their abstract syntax trees (ASTs), and assign as the dataset's fix labels the *smallest* sub-tree that changed between the correct and ill-typed attempt of the program.

2.2.3 Learning Candidate Fix Templates

Each labeled program in our dataset contains a fix, which we abstract to a fix template. For example, for the `mulByDigit` program in Figure 2.1 we get the candidate fix [hd $*$ i] and hence the fix template [_ $\oplus$ _]. However, a large dataset of fix-labeled programs, which may include many diverse solutions, can introduce a huge set of fix templates, which can be inappropriate for predicting the correct one to be used for the final program repair.

Therefore, the next step in our approach is to learn a set of fix templates that is *small enough* to automatically predict which template to apply to a given erroneous program, but nevertheless *covers* most of the fixes that arise in practice.

Partitioning the Fixes. We learn a suitable small set of fix templates by *partitioning* all the templates obtained from our dataset, and then selecting a single GAST to represent the fix templates from each fix template set. The partitioning serves two purposes. First, it identifies a small set of the most common fix templates which then enables the use of discrete classification algorithms to predict which template to apply to a new program. Second, it allows for the principled removal of outliers that arise because student submissions often contain non-standard or idiosyncratic solutions that we do not wish to use for suggesting fixes.

Unlike previous repair approaches that have used clustering to group together similar programs (e.g., [39, 130]), we partition our set of fix templates into their *equivalence classes* based on a fix similarity relation.

2.2.4 Predicting Templates via Multi-classification

Next, we train models that can correctly predict error locations and fix templates for a given ill-typed program. We use these models to generate candidate expressions as possible program fixes. To reduce the complexity of predicting the correct fix templates and error locations, we separate these problems and encode them into two distinct *supervised classification* problems.

Supervised Multi-Class Classification. We propose using a *supervised multi-class classification* problem for predicting fix templates. A *supervised* learning problem is one where, given a labeled training set, the task is to learn a function that accurately maps the inputs to output labels and generalizes to future inputs. In a *classification* problem, the function we are trying to learn maps inputs to a discrete set of two or more output labels, called *classes*. Therefore, we encode the task of learning a function that will map subexpressions of ill-typed programs to a small set of candidate fix templates as a *multi-class* classification (MCC) problem.

Feature Extraction. The machine learning models that we will train to solve our MCC problem expect datasets of labeled *fixed-length vectors* as inputs. Therefore, we define a transformation of fix-labeled programs to fixed-length vectors. Similarly to Seidel *et al.* [112], we define a set of feature extraction functions $f_1, \ldots, f_n$, that map program subexpressions to a numeric value (or just $\{0, 1\}$ to encode a boolean property). Given a set of feature extraction functions, we can represent a single program's AST as a set of fixed-length vectors by decomposing the AST e into a set of its constituent subexpressions $\{e_1, \ldots, e_m\}$ and then representing each e_i with the n-dimensional vector $[f_1(e_i), \ldots, f_n(e_i)]$. This method is known as a *bag-of-abstracted-terms* (BOAT) representation in previous work [112].

Predicting Templates via MCC. Our fix-labeled dataset can be updated so the labels represent the corresponding template that fixes each location, drawn from the minimal set of fix templates

that were acquired through partitioning. We then train a *Deep Neural Network (DNN)* classifier on the updated template-labeled data set.

Neural networks have the advantage of associating each class with a *confidence score* that can be interpreted as the model's probability of each class being correct for a given input according to the model's estimated distribution. Therefore, confidence scores can be used to rank fix template predictions for new programs and use them in descending order when synthesizing repairs. Exploiting recent advances in machine learning, we use deep and dense architectures [109] for more accurate fix template predictions.

Error Localization. We view the problem of finding error locations in a new program as a *binary* classification problem. In contrast with the template prediction problem, we want to learn a function that maps a program's subexpressions to a binary output representing the presence of an error or not. Therefore, this problem is equivalent to MCC with only two classes and thus, we use similar deep architectures of neural networks. For each expression in a given program, the learned model outputs a confidence score representing how likely it is an error location that needs to be fixed. We exploit those scores to synthesize candidate expressions for each location in descending order of confidence.

2.2.5 Synthesizing Feedback from Templates

Next, we use classic program *synthesis* techniques to synthesize candidate expressions that will be used to provide feedback to users. Additionally, synthesis is guided by predicted fix templates and a set of possible error locations, and returns a ranked list of *minimal* repairs to users as feedback.

Program Synthesis. Given a set of locations and candidate templates for those locations, we are trying to solve a problem of *program synthesis*. For each program location, we search over

all possible expressions in the language's grammar for a small set of candidate expressions that match the fix template and make the program type-check. Expressions from the ill-typed program are also used during synthesis to prune the search space of candidate expressions.

Synthesis for Multiple Locations. It is often the case that more than one location needs to be fixed. Therefore, we do not only consider the ordered set of single error locations for synthesis, but rather its power set. For simplicity, we consider fixing different program locations as independent; the probability we assign that a set of locations needs to be fixed is thus the product of their individual confidence scores. This is unlike recent approaches to multi-hunk program repair [104] where modifications depend on each other.

Ranking Fixes. Finally, we rank each solution by two metrics, the *tree-edit distance* and the *string-edit* distance. Previous work [39, 69, 130] has used such metrics to consider minimal changes, *i.e.* changes that are as close as possible to the original programs, so novice programmers are presented with more coherent feedback.

```
1    let rec mulByDigit i l =
2      match l with
3      | []      -> []
4      | hd::tl -> [v1 * v2] @ mulByDigit i tl
```

Figure 2.2: A candidate repair for the `mulByDigit` program.

Example. We see in Figure 2.2 a minimal repair that our method could return ($[v_1 * v_2]$ in line 4) using the template discussed in § 2.2.3 to synthesize it. While this solution is not the highest-ranked that our implementation returns (which would be identical to the human solution), it demonstrates relevant aspects of the synthesizer. In particular, this solution has some abstracted variables, v_1 and v_2. Our algorithm suggests to the user that they can replace the two variables with two distinct variables and insert the whole expression into a list, in order to obtain the correct

program. We hypothesize that such solutions produced by our algorithm can provide valuable feedback to novices, and we investigate that claim empirically in § 2.6.3.

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.e \mid e\ \bar{e} \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& \mid & n \mid b \mid e + e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \langle e, e \rangle \mid \texttt{match}\ e\ \texttt{with}\ \langle x, x \rangle \to e \\
& \mid & [] \mid e :: e \mid \texttt{match}\ e\ \texttt{with} \begin{cases} [] \to e \\ x :: x \to e \end{cases} \\
n & ::= & 0, 1, -1, \ldots \\
b & ::= & \texttt{true} \mid \texttt{false} \\
t & ::= & \alpha \mid \texttt{bool} \mid \texttt{int} \mid t \to t \mid t \times t \mid [t]
\end{array}
$$

Figure 2.3: Syntax of λ^{ML}

$$
\begin{array}{rcl}
e & ::= & \_ \mid \hat{x} \mid \lambda \hat{x}.e \mid \hat{x}\ \bar{e} \mid \texttt{let}\ \hat{x} = e\ \texttt{in}\ e \\
& \mid & \hat{n} \mid e \oplus e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & \langle e, e \rangle \mid \texttt{match}\ e\ \texttt{with}\ \langle \hat{x}, \hat{x} \rangle \to e \\
& \mid & [] \mid e :: e \mid \texttt{match}\ e\ \texttt{with} \begin{cases} [] \to e \\ \hat{x} :: \hat{x} \to e \end{cases}
\end{array}
$$

Figure 2.4: Syntax of λ^{RTL}

2.3 Learning Fix Templates

We start by introducing our approach for extracting useful *fix templates* from a training dataset comprised of paired erroneous and fixed programs. We express those templates in terms of a language that allows us to succinctly represent fixes in a way that captures the essential structure of various fix patterns that novices use in practice. However, extracting a single fix template for *each* fix in the program pair dataset yields too many templates to perform accurate predictions. Hence, we define a *similarity* relation between templates, which we use to *partition* the extracted templates into a small but representative set, that will make it easier to train precise models to predict fixes.

2.3.1 Representing User Fixes

Repair Template Language. Figure 2.4 describes our Repair Template Language, λ^{RTL}, which is a lambda calculus with integers, booleans, pairs, and lists, that extends our core ML language

λ^{ML} (Figure 2.3) with syntactic abstraction forms:

1. *Abstract variable* names $\hat{x}$ are used to denote variable occurrences for functions, variables and binders, *i.e.* $\hat{x}$ denotes an unknown variable name in λ^{RTL};

2. *Abstract literal* values $\hat{n}$ can represent *any* integer, float, boolean, character, or string;

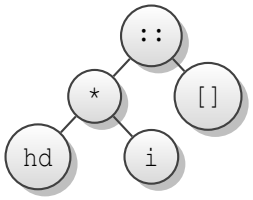
3. *Abstract operators* $\oplus$ similarly denote unknown unary or binary operators;

4. *Wildcard* expressions _ are used to represent *any* expression in λ^{RTL}, *i.e.* a program *hole*.

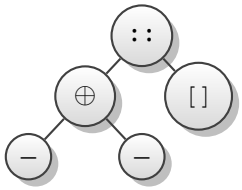
Recall from § 2.2.1 that we define fixes as replacements of expressions with new candidate expressions at specific program locations. Therefore, we use candidate expressions over λ^{RTL} to represent fix templates.

Generalizing ASTs. A *Generic Abstract Syntax Tree* (GAST) is a term from λ^{RTL} that represents many possible expressions from λ^{ML}. GASTs are abstracted from standard ASTs over the core language λ^{ML} using the abstract function that takes as input an expression e^{ML} over λ^{ML} and a depth d and returns an expression e^{RTL} over λ^{RTL}, *i.e.* a GAST with all variables, literals and operators of e^{ML} abstracted and all subexpressions starting at depth greater than d pruned and replaced with holes _.

Example. Recall our example program `mulByDigit` in Figure 2.1. The expression [hd $*$ i] replaces (hd $*$ i) in line 4, and hence, is the user's *fix*, whose AST is given in Figure 2.5a. The output of abstract, given this AST and a depth $d = 2$ as input, would be the GAST in Figure 2.5b, where the operator $*$ has been replaced with an abstract operator $\oplus$, and the sub-terms hd and i at depth 2 have been abstracted to wildcard expressions _. Hence, the λ^{RTL} term [_ $\oplus$ _] represents a potential fix template for `mulByDigit`.



(a) Fix AST



(b) Template GAST

Figure 2.5: (left) The fix from example Figure 2.1 and (right) a possible template for that fix.

2.3.2 Extracting Fix Templates from a Dataset

Our approach fully automates the extraction of fixes by harvesting a set of fix templates from a training set of program pairs. Given a program pair (p_{err}, p_{fix}) from the dataset, we extract a unique fix for each location in p_{err} that changed in p_{fix}. We do so with an expression-level diffRules [68] function. Recall that our fixes are replacements of expressions, so we abstract these extracted changes as our fix templates.

Contextual Repairs. Following Felleisen *et al.* [28], let $\mathbf{C}$ be the *context* in which an expression e appears in a program p, *i.e.* the program p with e replaced by a hole _. We write that $p = \mathbf{C}[e]$, meaning that if we fill the hole with the original expression e we obtain the original program p. In this fashion, diffRules finds a *minimal* (in number of nodes) expression replacement e_{fix} for an expression e_{err} in p_{err}, such that $p_{err} = \mathbf{C}_{p_{err}}[e_{err}]$ and $\mathbf{C}_{p_{err}}[e_{fix}] = p_{fix}$. There may be several such expressions, and diffRules returns all such changes.

Examples. If $f\ x$ is rewritten to $g\ x$, the context is $\mathbf{C} = \_\ x$ and the fix is g, since $\mathbf{C}[g] = g\ x$. If $f\ x$ is rewritten to $(f\ x) + 1$, the context is $\mathbf{C} = \_$, and the fix is the whole expression $(f\ x) + 1$, thus $\mathbf{C}[(f\ x) + 1] = (f\ x) + 1$. (Even though $f\ x$ appears in both the original and fixed programs, we consider the application expression $f\ x$ — but not f or x — to be replaced with the $+$ operator.)

2.3.3 Partitioning the Templates

Programs over λ^{ML} force similar fixes, such as changes to variable names, to have identical GASTs. Our next step is to define a notion of program fix *similarity*. Our definition supports the formation of a small but widely-applicable set of fix templates. This small set is used to train a repair predictor.

GAST Similarity. Two GASTs are *similar* when the root nodes are the same and their child subtrees (if any) can be ordered such that they are pairwise similar. For example, $x+3$ and $7-y$ yield the *similar* GASTs $\hat{x} \oplus \hat{n}$ and $\hat{n} \oplus \hat{x}$, where the root nodes are both abstract binary operators, one child is an abstract literal, and one child is an abstract variable.

Partitioning. GAST similarity defines a relation which is reflexive, symmetric, and transitive and thus an *equivalence* relation. We can now define *partitioning* as the computation of all possible *equivalence classes* of our extracted fix templates *w.r.t.* GAST similarity. Each class can consist of several member-expressions and any one of them can be viewed as the class *representative*. Each representative can then be used as a fix template to produce repairs for ill-typed programs.

For example, $\hat{x} \oplus \hat{n}$ and $\hat{n} \oplus \hat{x}$ are in the same class and either one can be used as the representative. The repair algorithm in section 2.5 will essentially consider both when fixing an erroneous program with this template.

Finally, our partitioning algorithm returns the top N equivalence classes based on their member-expressions frequency in the dataset. N is a parameter of the algorithm and is chosen to be as small as possible while the top N classes represent a large enough portion of the dataset.

2.4 Predicting Fix Templates

Given a candidate set of templates, our next task is to *train* a model that, when given an (erroneous) program, can predict which template to use for each location in that program. We do so by defining a function predict which takes as input (1) a feature extraction function Features, (2) a dataset DataSet of program pairs $(p_{err},\ p_{fix})$, and (3) a list of fix templates T. It returns as output a *fix-template-predictor* which, given an erroneous program, returns the locations of likely fixes, and the templates to be applied at those locations.

We build predict using three helper functions that carry out each of the high-level steps. First, the extract function extracts *features* and *labels* from the program pair dataset. Next, these feature vectors are grouped and fed into train which produces two models, LModel and TModel, that are respectively used for error localization and predicting fix templates. Finally, rank takes the features for a new (erroneous) program and queries the trained models to return the likely fix locations and corresponding fix templates.

Next, we describe the key data-types in Figure 2.6, our implementations of the three key steps, and how they are combined to yield the predict algorithm.

Confidences, Data and Labels. As shown in Figure 2.6, we define EMap a as a mapping from expressions e to values of type a, and TMap a as a mapping from templates T to such values. For example, TMap $\mathcal{C}$ is a mapping from templates T to their confidence scores $\mathcal{C}$. Data represents feature vectors used to train our predictive models, while Label $\mathcal{B}$ are the dataset labels for training and Label $\mathcal{C}$ are the output confidence scores. Finally, Pair is a program pair $(p_{err},\ p_{fix})$.

Features and Predictors. We define Features as a function that generates the feature vectors Data for each subexpression of an input program e. Those feature vectors are given in the form of a map EMap Data, which maps all subexpressions of the input program e to its feature vector Data.

Predictors are learned fix-template-predictors returned from our algorithm that are used to generate confidence score mappings for input programs e. Specifically, they return a map EMap (Label $\mathcal{C}$) that associates each subexpression of the input program e with a confidence score Label $\mathcal{C}$.

Architecture. First, the extract function takes as input the feature extraction functions Features, a list of templates [T] and a single program pair Pair and generates a map EMap (Data $\times$ Label $\mathcal{B}$) of feature vectors and boolean labels for all subexpressions of the erroneous input program from Pair. All feature vectors Data and labels Label $\mathcal{B}$ are then accumulated into one list, which is given as input to train and are used for training the two models LModel and TModel that are respectively used for predicting error locations and fix templates. Next, the two trained models LModel and TModel, along with Data from a new and previously unseen program, can be fed into rank. This produces a Predictor, which can be used to map subexpressions of the new program to possible error locations and fix templates.

2.4.1 Feature and Label Extraction

The machine learning algorithms that we use for predicting fix templates and error locations expect fixed-length *feature vectors* Data as their input. However, we want to repair variable-sized programs over λ^{ML}. We thus use the extract function to convert programs to feature vectors.

Following Seidel *et al.* [112], we choose to model a program as a *set* of feature vectors, where each element corresponds to a subexpression in the program. Thus, given an erroneous program p_{err} we first split it into its constituent subexpressions and then transform each subexpression into a single feature vector, *i.e.* Features p_{err} :: EMap Data. We only consider expressions inside a minimal type-error *slice*. We show here the five major feature categories used.

$$
\begin{array}{lcl}
\mathcal{C} & \doteq & \{r \in \mathbb{R} \mid 0 \leq r \leq 1\} \\
\mathcal{B} & \doteq & \{b \in \mathbb{R} \mid b = 0 \vee b = 1\} \\
\mathsf{T} & \doteq & e^{RTL} \\
\mathsf{EMap}\ a & \doteq & e \to a \\
\mathsf{TMap}\ a & \doteq & \mathsf{T} \to a \\
\mathsf{Data} & \doteq & [\mathcal{C}] \\
\mathsf{Label}\ a & \doteq & a \times \mathsf{TMap}\ a \\
\mathsf{Pair} & \doteq & e \times e \\
\mathsf{DataSet} & \doteq & [\mathsf{Pair}] \\
\hline
\mathsf{Features} & \doteq & e \to \mathsf{EMap\ Data} \\
\mathsf{Predictor} & \doteq & e \to \mathsf{EMap}\ (\mathsf{Label}\ \mathcal{C}) \\
\hline
\mathsf{abstract} & : & e \to \mathsf{T} \\
\mathsf{diffRules} & : & \mathsf{Pair} \to [e] \\
\hline
\mathsf{extract} & : & \mathsf{Features} \to [\mathsf{T}] \to \mathsf{Pair} \\
 & & \quad \to \mathsf{EMap}\ (\mathsf{Data} \times \mathsf{Label}\ \mathcal{B}) \\
\mathsf{train} & : & [\mathsf{Data} \times \mathsf{Label}\ \mathcal{B}] \to \mathsf{LModel} \times \mathsf{TModel} \\
\mathsf{rank} & : & \mathsf{LModel} \to \mathsf{TModel} \to \mathsf{Data} \to \mathsf{Label}\ \mathcal{C} \\
\hline
\mathsf{predict} & : & \mathsf{Features} \to [\mathsf{T}] \to \mathsf{DataSet} \to \mathsf{Predictor}
\end{array}
$$

Figure 2.6: A high-level API for converting program pairs to feature vectors and template labels.

Local syntactic features. These features describe the syntactic category of each expression e. In other words, for each production rule of e in Figure 2.3 we introduce a feature that is enabled (set to 1) if the expression was built with that production, and disabled (set to 0) otherwise.

Contextual syntactic features. The *context* in which an expression occurs can be critical for correctly predicting error sources and fix templates. Therefore, we include contextual features, which are similar to the local syntactic features but describe the parent and children of an expression. For example, the Is-[]-C1 feature would describe whether an expression's *first child* is []. This is similar to the *n-grams* used in linguistic models [33, 47].

Expression size. We also include a feature representing the *size* of each expression, *i.e.* how many subexpressions does it contain? This allows the model to learn that, *e.g.*, expressions closer to the leaves are more likely to be fixed than expressions closer to the root.

Typing features. The programs we are trying to repair are *untypeable*, but a *partial typing* derivation from the type checker could still provide useful information to the model. Therefore, we include *typing* features in our representation. Due to the parametric type constructors $\cdot \to \cdot$, $\cdot \times \cdot$, and $[\cdot]$, there is an *infinite* set of possible types — but we must have a *finite* set of features. We add features for each abstract type constructor that describes whether a given type uses that constructor. For example, the type $\texttt{int} \to \texttt{int} \to \texttt{bool}$ would enable the $\cdot \to \cdot$, `int`, and `bool` features.

We add these features for parent and child expressions to summarize the context, but also for the current expression, as the type of an expression is not always clear syntactically.

Type error slice. We wish to distinguish changes that could fix the error from changes that *cannot possibly* fix the error. Thus, we compute a minimal type-error *slice* (*e.g.* [42, 123]) for the program (*i.e.* the set of expressions that contribute to the error) and if the program contains

multiple type-errors, we compute a minimal slice for each error. We then have a post-processing step that discards all expressions that are not included in those slices.

Labels. Recall that we use two predictive models, LModel for error localization and TModel for predicting fix templates. We thus require two sets of *labels* associated with each feature vector, given by Label $\mathcal{B}$. LModel is trained using the set [Data $\times$ $\mathcal{B}$], while TModel using the set [Data $\times$ TMap $\mathcal{B}$].

LModel's labels of type $\mathcal{B}$ are set to "true" for each subexpression of a program p_{err} that changed in p_{fix}. A label TMap $\mathcal{B}$, for a subexpression of p_{err}, maps to the repair template T that was used to fix it. TMap $\mathcal{B}$ associates all subexpressions with a fixed number of templates [T] given as input to extract. Therefore, for the purpose of template prediction, TMap $\mathcal{B}$ can be viewed as a fixed-length boolean vector that represents the fix templates used to repair each subexpression. This vector has at most one slot set to "true", representing the template used to fix p_{err}. These labels are extracted using diffRules and abstract, similarly to the way that templates were extracted in § 2.3.2.

2.4.2 Training Predictive Models

Our goal with the train function is to train two separate *classifiers* given a training set [Data $\times$ Label $\mathcal{B}$] of labeled examples. LModel predicts error locations and TModel predicts fix templates for a new input program p_{err}. Critically, we require that the error localization classifier output a *confidence score* $\mathcal{C}$ that represents the probability that a subexpression is the error that needs to be fixed. We also require that the fix template classifier output a confidence score $\mathcal{C}$ for each fix template that measures how sure the classifier is that the template can be used to repair the associated location of the input program p_{err}.

We consider a standard learning algorithm to generate our models: *neural networks*. A thorough introduction to neural networks is beyond the scope of this work [45, 87].

Neural Networks. The model that we use is a type of neural network called a *multi-layer perceptron*. A multi-layer perceptron can be represented as a directed acyclic graph whose nodes are arranged in layers that are fully connected by weighted edges. The first layer corresponds to the input features, and the final to the output. The output of an internal node is the sum of the weighted outputs of the previous layer passed to a non-linear function, called the activation function. The number of layers, the number of nodes per layer, and the connections between layers constitute the *architecture* of a neural network. In this work, we use relatively *deep neural networks* (DNN). We can train a DNN LModel as a binary classifier, which will predict whether a location in a program p_{err} has to be fixed or not.

Multi-class DNNs. While the above model is enough for error localization, in the case of template prediction we have to select from more than two *classes*. We again use a DNN for our template prediction TModel, but we adjust the output layer to have N nodes for the N chosen template-classes. For multi-class classification problems solved with neural networks, usually a *softmax* function is used at output layer [11, 37]. Softmax assigns probabilities to each class that must add up to 1. This additional constraint speeds up training.

2.4.3 Predicting Fix Templates

Our ultimate goal is to be able to pinpoint what parts of an erroneous program should be repaired and what fix templates should be used for that purpose. Therefore, the predict function uses rank to predict all subexpressions' confidence scores $\mathcal{C}$ to be an error location and confidence scores TMap $\mathcal{C}$ for each fix template. We show here how all the functions in our high-level API in Figure 2.6 are combined to produce a final list of confidence scores for a new program p. Algorithm 1 presents our high-level predict algorithm.

Algorithm 1 Predicting Templates Algorithm

Input: Feature Extraction Functions F, Fix Templates Ts, Program Pair Dataset D
Output: Predictor Pr

procedure PREDICT(F, Ts, D)
  $D_{ML} \leftarrow \emptyset$
  for all $p_{err} \times p_{fix} \in D$ **do**
    $d \leftarrow$ EXTRACT(F, Ts, $p_{err} \times p_{fix}$)
    $D_{ML} \leftarrow D_{ML} \cup$ INSLICE(p_{err}, d)
  $Models \leftarrow$ TRAIN(D_{ML})
  $Data \leftarrow \lambda p.$ INSLICE(p, EXTRACT(F, Ts, $p \times p$))
  $Pr \leftarrow \lambda p.$ MAP($\lambda \tilde{p}.$ RANK($Models$, $\tilde{p}[0]$), $Data(p)$)
  return Pr

The Prediction Algorithm. Our algorithm first extracts the machine-learning-amenable dataset D_{ML} from the program pairs dataset D. For each program pair in D, EXTRACT returns a mapping from the erroneous program's subexpressions to features and labels. Then, INSLICE keeps only the expressions in the the type-error slice and evaluates to a list of the respective feature and label vectors, which is added to the D_{ML} dataset. This dataset is used by the TRAIN function to generate our predictive $Models$, *i.e.* LModel and TModel.

At this point we want to generate a Predictor for a new unknown program p. We perform feature extraction for p with EXTRACT, and use INSLICE to restrict to expressions in p's type-error slice. The result is given by $Data(p)$.

RANK is then applied to all subexpressions produced by $Data(p)$ with MAP, which will create a mapping of the type EMap (Label $\mathcal{C}$) associating expressions with confidence scores. We apply RANK to each feature vector that corresponds to an expression in the type-error slice of p. These vectors are the first elements of $\tilde{p} \in Data(p)$, which are of type Data $\times$ Label $\mathcal{B}$. Finally, Predictor Pr is returned, which is used by our synthesis algorithm in section 2.5 to correlate subexpressions in p with their confidence scores.

2.4.4 Discussion

An alternative to the two separate predictive models, LModel and TModel, would be to have one *joint* model to predict both error locations and fix templates. One could simply add an "empty" fix template to the set of the N extracted templates. Then, a multi-class DNN could be trained on the dataset, using $N+1$ classes instead. When the "empty" fix template is predicted, it denotes no error at that location, while the rest of the classes denote an error along with the fix template to be used. While the approach of one joint model is quite intuitive, we found in our early experiments that it does not produce as accurate predictions as the two separate models.

Learning representations is a remarkable strength of DNNs, so manually extracting features is usually discouraged. Recently, there has been some work in learning program representations for use in predictive models [6, 10]. However, we found that the BOAT features are essential for high accuracy (see subsection 2.6.1) given the relatively small size of our dataset, similarly to previous work [112]. In future work, however, it would be interesting to learn features automatically and avoid the step of manually extracting them.

2.5 Template-Guided Repair Synthesis

We use program synthesis to fully repair a program using predicted fix templates and locations from our machine learning models. We present in § 2.5.1 a synthesis algorithm for producing *local repairs* for a given program location. In § 2.5.2, we show how we use local repairs to repair programs that may have *multiple* error locations.

2.5.1 Local Synthesis from Templates

Enumerative Program Synthesis. We utilize classic *enumerative* program synthesis that is guided by a fix template. Enumerative synthesis searches all possible expressions over a language until a high-level specification is reached. In our case, we initially synthesize independent *local repairs* for a program that already captures the user's intent. Therefore, the required specification is that the repaired program is type-safe. However, if the users provide type signatures for their programs, they can be used as a stricter specification.

Given a location l, a template t and a maximum depth d, Algorithm 2 searches over all possible expressions over λ^{ML} that will satisfy those goals by generating a local repair that fills t's GAST with concrete variables, literals, functions *etc.* Our technique can also reuse subexpressions e at location l for t's concretization to further optimize the search.

Template-Guided Local Repair. Using the REPAIR method (Algorithm 2), we produce local repairs R for a given location L of an erroneous program P. REPAIR fills in a template T based on the context-free grammar λ^{ML}. It traverses the GAST of template T from root node downward, producing candidate local repairs of maximum depth D.

When a hole $\alpha \in T$ is found, the algorithm expands T's GAST one more level using λ^{ML}'s production rules Q. The production rules are considered in a ranked order based on the subexpressions that already appear in the rest of the template T and program location L. Each rule is then applied to template T, returning an *instantiated* template $\hat{T}$, which is inserted into the

Algorithm 2 Local Repair Algorithm

Input: Language Grammar λ^{ML}, Program P, Template T, Repair Location L, Max Repair Depth D
Output: Local Repairs R

```
1:  procedure REPAIR(λ^ML, P, T, L, D)
2:      R ← ∅
3:      for all d ∈ [1...D] do
4:          α̃ ← NONTERMINALSAT(T, d)
5:          for all α ∈ RANKNONTERMINALS(α̃, P, L) do
6:              if ISHOLE(α) then
7:                  Q ← GRAMMARRULES(λ^ML)
8:                  β̃ ← {β | (α, β) ∈ Q}
9:                  for all β ∈ RANKRULES(β̃, T) do
10:                     T̂ ← APPLYRULE(T, (α, β))
11:                     R ← R ∪ {T̂}
12:             else
13:                 for all σ ∈ GETTERMINALS(P, L, λ^ML) do
14:                     T̂ ← REPLACENODE(T, α, σ)
15:                     R ← R ∪ {T̂}
16:     return R
```

list of candidate local repairs R.

If node α is not a hole, terminals from the subexpressions at location L, the program P in general and the grammar λ^{ML} are used to concretize that node, depending on the λ^{RTL} terminal node α. For each of these template T modifications, we insert an instantiated template $\hat{T}$ into R.

2.5.2 Ranking Error Locations

Error Location Confidence. Recall from section 2.4 that for each subexpression in a program's type-error slice, LModel generates a confidence score C for it being the error location, and TModel generates scores for the fix templates.

Our synthesis algorithm ranks all program locations based on their confidence scores C. For all locations in descending confidence score order, a fix template is used to produce a local repair using Algorithm 2. Fix templates are considered in descending order of confidence. Then expressions from the returned list of local repairs R replace the expression at the given program location. The procedure tries the remaining repairs, templates, and locations until a type-correct

program is found.

Following [69], we allow our final local repairs to have program holes _ or abstracted variable $\hat{x}$ in them. However, Algorithm 2 will prioritize the synthesis of complete solutions. Abstract λ^{RTL} terms can have any type when type-checking concrete solutions, similarly to OCaml's raise Exn.

Multiple Error Locations. In practice, frequently more than one program location needs to be repaired. We thus extend the above approach to fix programs with multiple errors. Let the confidence scores $\mathcal{C}$ for all locations L in the type error slice from our error localization model LModel be $(l_1, c_1), \ldots, (l_k, c_k)$, where l_i is a program location and c_i its error confidence score. We assume for simplicity that the probabilities c_i are independent. Thus the probability that *all* the locations $\{l_i \ldots l_j\}$ need to be fixed is the product $c_i \cdots c_j$. Therefore, instead of ranking and trying to find fixes for single locations l, we use *sets* of locations ($\{l_i\}, \{l_i, l_j\}, \{l_i, l_j, l_k\}$, *etc.*), ranked by the products of their confidence scores. For a given set, we use Algorithm 2 independently for each location in the set and apply all possible combinations of local repairs, looking again for a type-correct solution.

2.6 Evaluation

We have implemented analytic program repair in RITE: a system for repairing type errors for a purely functional subset of OCAML. Next, we describe our implementation and an evaluation that addresses three questions:

- **RQ1**: How *accurate* are RITE's predicted repairs? (§ 2.6.1)
- **RQ2**: How *efficiently* can RITE synthesize fixes? (§ 2.6.2)
- **RQ3**: How *useful* are RITE's error messages? (§ 2.6.3)
- **RQ4**: How *precise* are RITE's template fixes? (§ 2.6.4)

Training Dataset. For our evaluation, we use an OCAML dataset gathered from an undergraduate Programming Languages university course, previously used in related work [110, 112]. It consists of erroneous programs and their subsequent fixes and is divided in two parts; the Spring 2014 class (SP14) and the Fall 2015 class (FA15). The homework required students to write 23 distinct programs that demonstrate a range of functional programming idioms, *e.g.* higher-order functions and (polymorphic) algebraic data types.

Feature Extraction. RITE represents programs with BOAT vectors of 449 features from each expression in a program: 45 local syntactic, 315 contextual, 88 typing features, and 1 expression size feature. For contextual features, for each expression we extract the local syntactic features of its first 4 (left-to-right) children. In addition, we extract those features for its ancestors, starting from its parent and going up to two more parent nodes. For typing features, we support int s, float s, chars, string s, and the user-defined expr. These features are extracted for each expression and its context.

Dataset Cleaning. We extract fixes as expressions replacements over a program pair using diffRules. A disadvantage of using diffRules s with this dataset is that some students may have made many, potentially unrelated, changes between compilations; at some point the "fix" becomes a "rewrite". These rewrites can lead to meaningless fix templates and error locations. We discard such outliers when the fraction of subexpressions that have changed in a program is more than one standard deviation above the mean, establishing a diffRules threshold of 40%. We also discard programs that have changes in 5 or more locations, noting that even state-of-the-art multi-location repair techniques cannot reproduce such "fixes" [104]. The discarded changes account for roughly 32% of each dataset, leaving 2,475 program pairs for SP14 and 2,177 pairs for FA15. Throughout, we use SP14 as a training set and FA15 as a test set.

DNN based Classifier. RITE's template prediction uses a multi-layer neural network DNN based classifier with three fully-connected hidden layers of 512 neurons. The neurons use rectified linear units (ReLU) as their activation function [85]. The DNN was trained using *early stopping* [45]: training is stopped when the accuracy on a distinct small part of the training set is not improved after a certain amount of epochs (5 epochs, in our implementation). We set the maximum number of epochs to 200. We used the ADAM optimizer [58], a variant of stochastic gradient descent that converges faster.

2.6.1 RQ1: Accuracy

Most developers will consider around five or six suggestions before falling back to manual debugging [61, 88]. Therefore, we consider RITE's accuracy up to the *top six* fix template predictions, *i.e.* we check if any of the top-N predicted templates actually correspond to the users's edit. These predicted templates are not shown to the user; they are only used to guide the synthesis of concrete repairs which are then presented to the user.

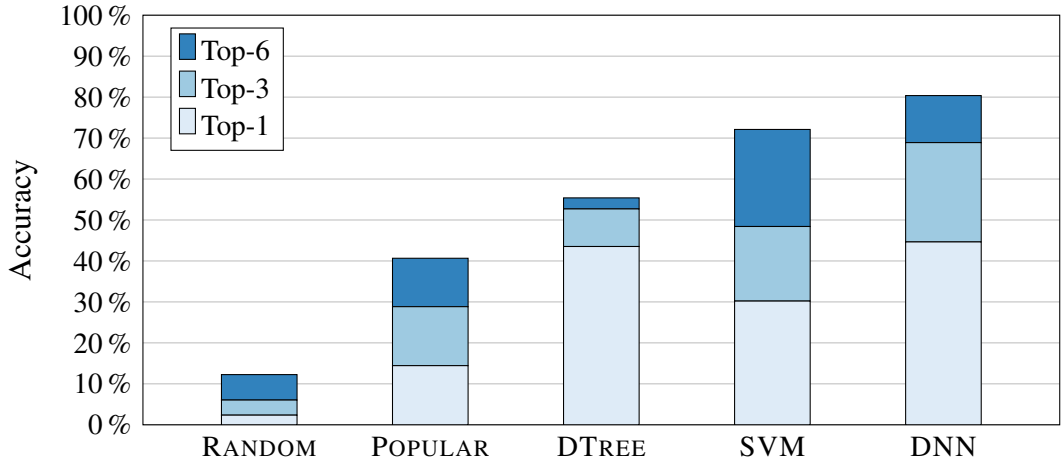


Figure 2.7: Results of our template prediction classifiers using the *50 most popular* templates. We present the results up to the top 6 predictions, since our synthesis algorithm considers that many templates before falling to a different location.

Baselines. We compare RITE's DNN-based predictor against two baseline classifiers: a RANDOM classifier that returns templates chosen uniformly at random from the 50 templates learned from the SP14 training dataset, and a POPULAR classifier that returns the most popular templates in the training set in decreasing order. We also compare to a *decision tree* (DTREE) and an SVM classifier trained on the SP14 data, since these are two of the most common learning algorithms [45].

Results: Accuracy of Prediction. Figure 2.7 shows the accuracy results of our template prediction experiments. The y-axis describes the fraction of *erroneous* sub-terms (locations) for which the actual repair was one of the top-K predicted repairs. The naive baseline of selecting templates at random achieves 2% Top-1 accuracy (12% Top-6), while the POPULAR classifier achieves a Top-1 accuracy of 14% (41% Top-6). Our DNN classifier significantly outperforms these naive classifiers, ranging from 45% Top-1 accuracy to 80% Top-6 accuracy. In fact, even with only DNN's first prediction one outperforms top 6 predictions of both RANDOM and POPULAR. The RANDOM classifier's low performance is as expected. The POPULAR classifier performs better: some homework assignments were shared between SP14 and FA15 quarters and,

while different groups of students solved these problems for each quarter, the novice mistakes that they made seem to have a pattern. Thus, the most *popular "fixes"* (and therefore the relevant templates) from SP14 were also popular in FA15.

We also observe that DTREE achieves a Top-1 accuracy close to that of DNN's (*i.e.* 44% vs. 45%) but fails to improve with more predictions (*i.e.* with Top-6, 55% vs. 80%). On the other hand, the SVM does poorly on the Top-1 accuracy (*i.e.* 30% vs. 45%) but does significantly better with more predictions (*i.e.* with Top-6, 72% vs. 80%). Therefore, we observe that more sophisticated learning algorithms can actually learn patterns from a corpus of fixed programs, with DNN classifiers achieving the best performance in each category.

Results: Template "Confusion". The *confusion matrix* of the each location's top prediction shows which templates our models mix up. Figure 2.8 shows this matrix for the top 30 templates acquired from the SP14 training set and were tested on the FA15 dataset. Note that most templates are predicted correctly and only a few of them are often mis-predicted for another template. For example, we see that programs that require template 20 ($\texttt{let}\ \hat{z} = \texttt{match}\ \hat{t}\ \texttt{with}\ (\hat{x},\ \hat{y}) \rightarrow \hat{a}\ \texttt{in}\ \_$) to be fixed, almost always are mis-predicted with template 11 ($\texttt{let}\ (\hat{x},\ \hat{y}) = \hat{t}\ \texttt{in}\ (\_,\ \_)$). We observe that these templates are still very similar, with both of them having a top-level let that manipulates tuples $\hat{t}$.

RITE learns correlations between program features and repair templates, yielding almost *2x higher* accuracy than the naive baselines and 8% more than the other sophisticated learning algorithms. By abstracting programs into features, RITE is able to *generalize* across years and different kinds of programs.

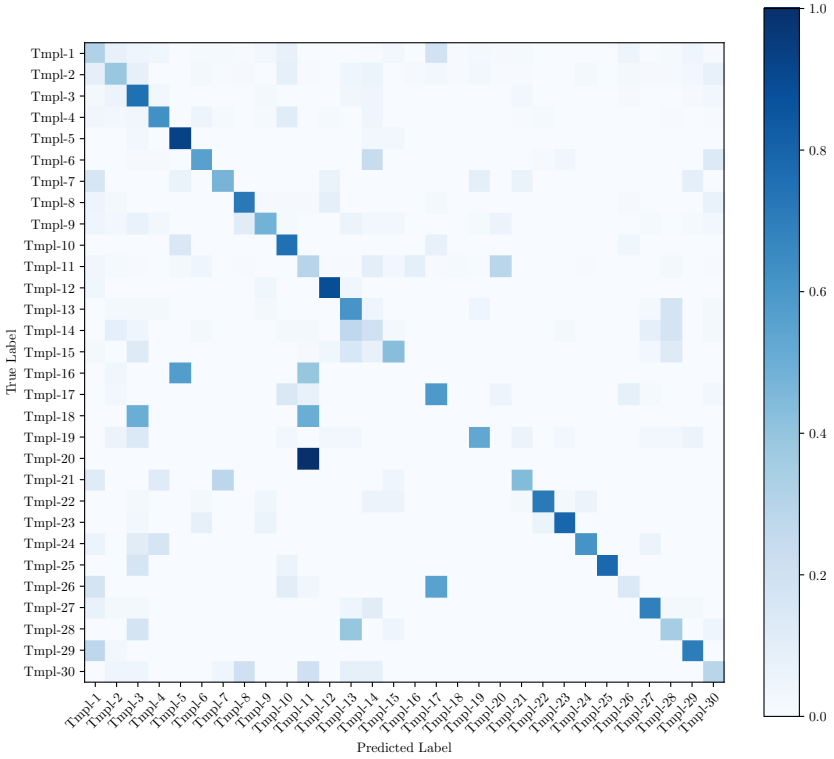


Figure 2.8: The confusion matrix of the *top 30* templates. Bolder parts of the heatmap show templates that are often mis-predicted with another template. The bolder the diagonal is, the more accurate predictions we make.

2.6.2 RQ2: Efficiency

Next we evaluate RITE's efficiency by measuring how many programs it is able to generate a (well-typed) repair for. We limit the synthesizer to 90 seconds. (In general the procedure is undecidable, and we conjecture that a longer timeout will diminish the practical usability for novices.) Recall that the repair synthesis algorithm is guided by the repair template predictions. We evaluate the efficiency of RITE by comparing it against a baseline NAIVE implementation that, given the predicted fix location, attempts to synthesize a repair from the trivial "hole" template.

Figure 2.9 shows the cumulative distribution function of RITE's and NAIVE's repair rates over their synthesis time. We observe that using the predicted templates for synthesis allows RITE to generate type-correct repairs for almost 70% of the programs in under 20 seconds, which is nearly 12 points higher than the NAIVE baseline. We also observe that RITE successfully repairs

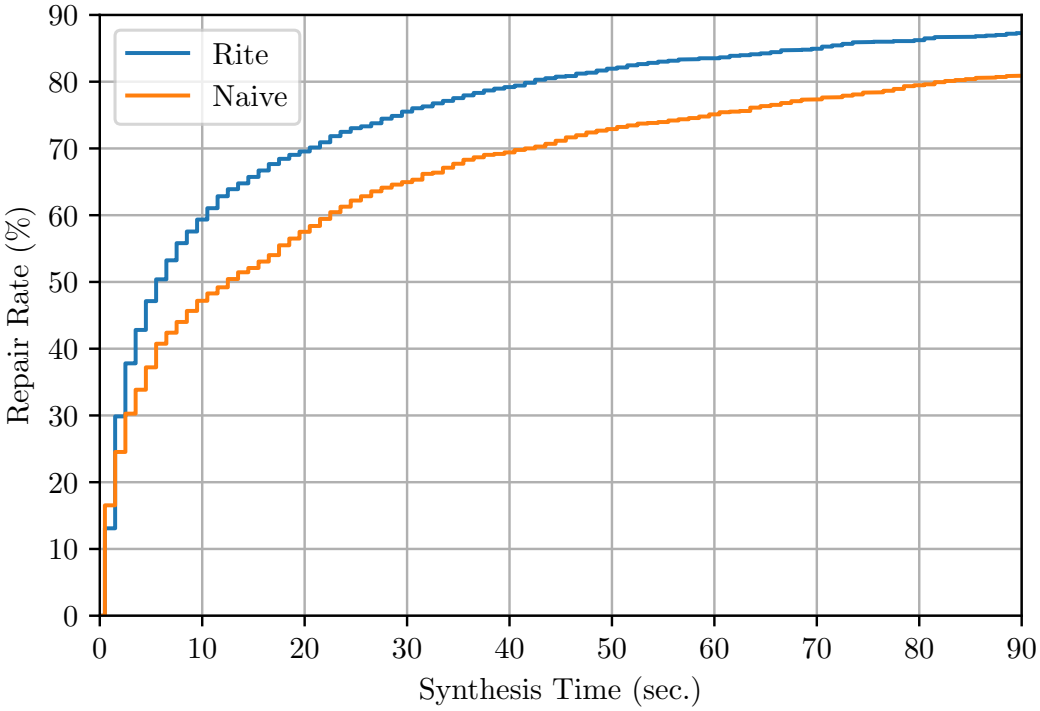


Figure 2.9: The proportion of the test set that can be repaired within a given time.

around 10% more programs than NAIVE for times greater than 20 seconds. While the NAIVE approach is still able to synthesize well-typed repairs relatively quickly, we will see that these repairs are of much lower quality than those generated from the predicted templates (§ 2.6.4).

RITE can generate type-correct repairs for the vast majority of ill-typed programs in under 20 seconds.

2.6.3 RQ3: Usefulness

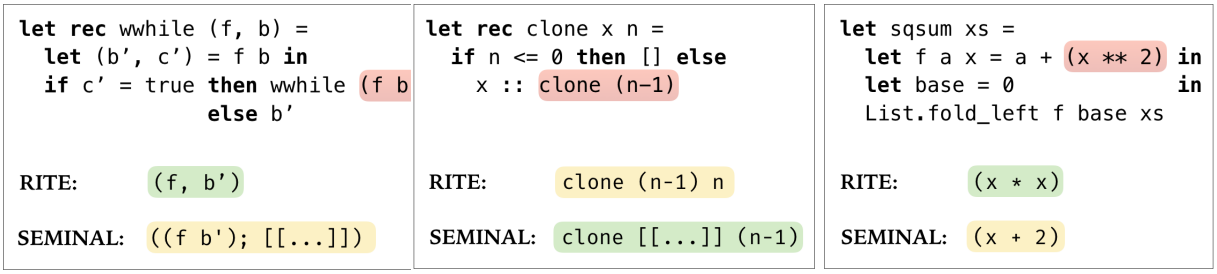
The primary outcome is whether the repair-based error messages generated by RITE were actually useful to novices. To assess the quality of RITE's repairs, we conducted an online human study with 29 participants. Each participant was asked to evaluate the quality of the program fixes and their locations against a state-of-the-art baseline (SEMINAL [69]). For each program, beyond

the two repairs, participants were presented with the original ill-typed program, along with the standard OCAML compiler's error message and a short description of what the original author of the program intended it to do. From this study, we found that both the edit locations and final repairs produced by RITE were better than SEMINAL's in a statistically significant manner.

User Study Setup. Study participants were recruited from two public research institutes (University of California, San Diego and University of Michigan), and from advertisement on Twitter. Participants had to assess the quality of, and give comprehensible bug descriptions for, at least 5 / 10 stimuli. The study took around 25 minutes to complete. Participants were compensated by entering a drawing for an Amazon Echo voice assistant. There were 29 valid participants. We created the stimuli by randomly selecting a corpus of 21 buggy programs from the 1834 programs in our dataset where repairs were synthesized. From this corpus, each participant was shown 10 randomly-selected buggy programs, and two candidate repairs: one generated by RITE and one by SEMINAL. For both algorithms, we used the highest-ranked solution returned. Participant were always unaware which tool generated which candidate patch. Participants were then asked to assess the quality of each candidate repair on a Likert scale of 1 to 5 and were asked for a binary assessment of the quality of each repair's edit location. We also collected self-reported estimates of both programming and OCAML-specific experience as well as qualitative data assessing factors influencing each participant's subjective judgment of repair quality. From the 29 participants, we collected 554 patch quality assessments, 277 each for RITE and SEMINAL generated repairs.

Results. In a statistically-significant manner, humans perceive that RITE's fault localization and final repairs are both of higher quality than those produced by SEMINAL ($p = 0.030$ and $p = 0.024$ respectively).[1] Regarding fault localization, we find that humans agreed with RITE-identified edit locations 81.6% of the time but only agreed with those of SEMINAL 74.0% of the time. As for the final repair, humans also preferred RITE's patches to those produced by SEMINAL. Specifically,

[1] All tests for statistical significance used the Wilcoxon signed-rank test.



(a) RITE (4.5/5) better than SEMINAL (1.1/5) with 12 responses $p = 0.002$.

(b) RITE (1.5/5) worse than SEMINAL (4.1/5) with 18 responses $p = 0.0002$.

(c) RITE (4.8/5) better than SEMINAL (1.2/5) with 17 responses $p = 0.0003$.

Figure 2.10: Three erroneous programs with the repairs that RITE and SEMINAL generated for the *red* error locations.

RITE's repairs achieved an average quality rating of 2.41/5 while SEMINAL's repairs had an average rating of only 2.11/5, a 14% increase ($p = 0.030$), showing a statistically-significant improvement over SEMINAL.

Qualitative Comparison. We consider several case studies where there were statistically-significant differences between the human ratings for RITE's and SEMINAL's repairs. The task in Figure 2.10a is that `wwhile(f, b)` should return x where there exist values $v_0, ..., v_n$ such that: $b = v_0$, $x = v_n$, and for each i between 0 and $n-2$, we have $fv_i = (v_i + 1, true)$ and $fv_{n-1} = (v_n, false)$. The task in Figure 2.10b is to return a list of `n` copies of `x`. The task in Figure 2.10c is to return the sum of the squares of the numbers in the list `xs`. Humans rated RITE's repairs better for the programs in Fig 2.10a and 2.10c. In both cases, RITE's found a solution which type-checks and conforms to the problem's semantic specification. SEMINAL, however, found a repair that was either incomplete (2.10a) or semantically incorrect (2.10c). On the other hand, in 2.10b, RITE does worse as the *second* parameter should be `n-1`. In fact, RITE's second ranked repair is the correct one, but it is equal to the first in terms of edit distance.

Humans perceive both RITE's edit locations and final repair quality to be better than those produced by SEMINAL, a state-of-the-art OCAML repair tool, in a statistically-significant

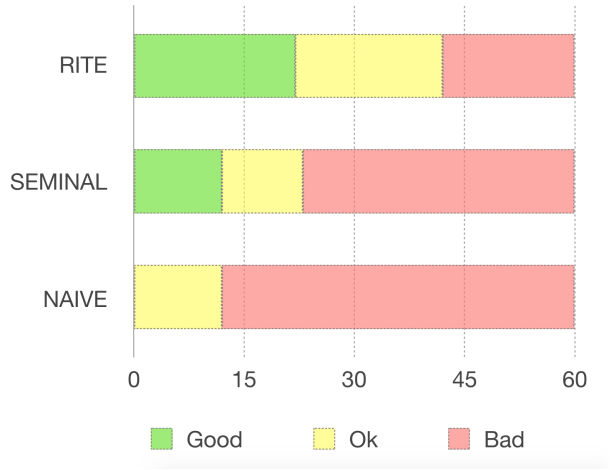


Figure 2.11: Rating the errors generated by RITE, SEMINAL and NAIVE enumeration.

manner.

2.6.4 RQ4: Impact of Templates on Quality

Finally, we seek to evaluate whether RITE's template-guided approach is really at the heart of its effectiveness. To do so, as in § 2.6.2, we compared the results of using RITE's error messages synthesized from predicted templates to those generated by a NAIVE synthesizer that returns the first well-typed term (*i.e.* synthesized from the trivial "hole" template).

User Study Setup. For this user study, we used a corpus of 20 buggy programs randomly chosen in § 2.6.3. For each of the programs we generated three messages: using RITE, using SEMINAL, and using the NAIVE approach but at the *same location* predicted by RITE. We then randomized and masked the order in which the tools' messages were reported, and asked three experts (authors of the original paper who had not seen the output of any tool for any of those instances) to rate the messages as one of "Good", "Ok" or "Bad".

Results. Figure 2.11 summarizes the results of the rating. Since each of 20 programs received 3 ratings, there are a total of 60 ratings per tool. RITE dominates with 22 Good, 20 Ok and 18

Bad ratings; SEMINAL follows with only 12 Good, 11 Ok and 37 Bad; while NAIVE received no Good scores, 12 Ok scores and a dismal 48 Bad scores. On average (with Bad = 0, Ok = 0.5, Good = 1), RITE scored 0.53, SEMINAL 0.30, and NAIVE just 0.1. Our rating agreement kappa is 0.54, which is considered "moderate agreement".

Repairs generated from predicted templates were of significantly higher quality than those from expert-biased enumeration (SEMINAL) or NAIVE enumeration.

2.7 Related Work

There is a vast literature on automatically repairing or patching programs: we focus on the most closely related work on providing feedback for novice errors.

Example-Based Feedback. Recent work uses *counterexamples* that show how a program went wrong, for type errors [111] or for general correctness properties where the generated inputs show divergence from a reference implementation or other correctness oracle [115]. In contrast, we provide feedback on how to fix the error.

Fault Localization. Several authors have studied the problem of *fault localization*, *i.e.* winnowing down the set of locations that are relevant for the error, often using slicing [42, 95, 123, 129], counterfactual typing [18] or bayesian methods [137]. NATE [112] introduced the BOAT representation, and showed it could be used for accurate localization. We aim to go beyond localization, into suggesting concrete *changes* that novices can make to understand and fix the problem.

Repair-model based feedback. SEMINAL [69] *enumerates* minimal fixes using an expert-guided heuristic search. The above approach is generalized to general correctness properties by [114] which additionally performs a *symbolic* search using a set of expert provided *sketches* that represent possible repairs. In contrast, RITE learns a template of repairs from a corpus yielding higher quality feedback (§ 2.6).

Corpus-based feedback. CLARA [39] uses code and execution traces to match a given incorrect program with a "nearby" correct solution obtained by clustering all the correct answers for a particular task. The matched representative is used to extract repair expressions. Similarly, SARFGEN [130] focuses on structural and control-flow similarity of programs to produce repairs, by using AST vector embeddings to calculate distance metrics (to "nearby" correct programs) more robustly. CLARA and SARFGEN are data-driven, but both assume there is a "close" correct

sample in the corpus. In contrast, RITE has a more general philosophy that *similar errors have similar repairs*: we extract generic fix templates that can be applied to arbitrary programs whose errors (BOAT vectors) are similar. The TRACER system [3] is closest in philosophy to ours, except that it focuses on single-line compilation errors for C programs, where it shows that NLP-based methods like sequence-to-sequence predicting DNNs can effectively suggest repairs, but this does not scale up to fixing general type errors. We have found that OCAML's relatively simple *syntactic* structure but rich *type* structure make token-level seq-to-seq methods quite imprecise (*e.g. deleting* offending statements suffices to "repair" C but yields ill-typed OCAML) necessitating RITE's higher-level semantic features and (learned) repair templates.

HOPPITY [24] is a DNN-based approach for fixing buggy JavaScript programs. HOPPITY treats programs as graphs that are fed to a *Graph Neural Network* to produce fixed-length embeddings, which are then used in an LSTM model that generates a sequence of primitive edits of the program graph. HOPPITY is one of the few tools that can repair errors spanning multiple locations. However, it relies solely on the learned models to generate a sequence of edits, so it doesn't guarantee returning valid JavaScript programs. In contrast, RITE, uses the learned models to get appropriate error locations and fix templates, but then uses a synthesis procedure to always generate type-correct programs.

GETAFIX [7] and REVISAR [99] are two more systems that learn fix patterns using AST-level differencing on a corpus of past bug fixes. They both use *anti-unification* [64] for generalizing expressions and, thus, grouping together fix patterns. They cluster together bug fixes in order to reduce the search space of candidate patches. While REVISAR [99] ends up with one fix pattern per bug category using anti-unification, GETAFIX [7] builds a hierarchy of patterns that also include the context of the edit to be made. They both keep before and after expression pairs as their fix patterns, and they use the before expression as a means to match an expression in a new buggy program and replace it with the after expression. While these methods are quite effective, they are only applicable in recurring bug categories *e.g.* how to deal with a null pointer

exception. RITE on the other hand, attempts to generalize fix patterns even more by using the GAST abstractions, and predicts proper error locations and fix patterns with a learned model from the corpus of bug fixes, and so so can be applied to a diverse variety of errors.

PROPHET [74] is another technique that uses a corpus of fixed buggy programs to learn a probabilistic model that will rank candidate patches. Patches are generated using a set of predefined transformation schemas and condition synthesis. PROPHET uses logistic regression to learn the parameters of this model and uses over 3500 extracted program features to do so. It also uses an instrumented recompile of a faulty program together with some failing input test cases to identify what program locations are of interest. While this method can be highly accurate for error localization, their experimental results show that it can take up to 2 hours to produce a valid candidate fix. In contrast, RITE's pretrained models make finding proper error locations and possible fix templates more robust.

2.8 Conclusion

We have presented analytic program repair, a new data-driven approach to provide repairs as feedback for type errors. Our approach is to use a dataset of ill-typed programs and their fixed versions to learn a representative set of fix templates, which, via multi-class classification allows us to accurately predict fix templates for new ill-typed programs. These templates guide the synthesis of program repairs in a tractable and precise manner.

We have implemented our approach in RITE, and demonstrate, using a corpus of 4,500 ill-typed OCAML programs drawn from two instances of an introductory programming course, that RITE makes accurate fix predictions 69% of the time when considering the top three templates and surpass 80% when we consider the top six, and that the predicted templates let us synthesize repairs for over 70% of the test set in under 20 sec. Finally, we conducted a user study with 29 participants which showed that RITE's repairs are of higher quality than those from the state-of-the-art SEMINAL tool which incorporates several expert-guided heuristics for improving the quality of repairs and error messages. Thus, our results demonstrate the unreasonable effectiveness of data for generating better error messages.

2.9 Acknowledgements

Chapter 2, in part, is a reprint of the material as it appears in the Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, Ranjit Jhala, PLDI 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 3

SEQ2PARSE: Neurosymbolic Parse Error Repair

3.1 Introduction

Parse errors can vex novices who are learning to program. Traditional error messages only indicate the first error or produce messages that are either incomprehensibly verbose or not descriptive enough to help swiftly remedy the error [94, 124]. When they occur in larger code bases, parse errors may even trouble experts, and can require a great deal of effort to fix [1, 22, 63].

Owing to their ubiquity and importance, there are *two* established lines of work on automatically suggesting *repairs* for parse errors. In the first line, Programming Languages researchers have investigated *symbolic* approaches starting with classical parsing algorithms, *e.g.*, LR [4] or Earley [25]. These algorithms can accurately locate syntax errors, but do not provide *actionable* feedback on how to fix the error. [5] extends these ideas to implement *error-correcting parsers* (EC-Parsers) that use special error production rules to handle programs with syntax errors and synthesize minimal-edit parse error repairs. Sadly, EC-parsers have remained mostly of theoretical interest, as their running time is cubic in the program size, and quadratic in the size of the language's grammar, which has rendered them impractical for real-world languages [80, 97].

In the second line, Machine Learning and NLP researchers have pursued *Deep Neural Network* (DNN) approaches using advanced sequence-to-sequence models [44, 119] that use a large corpus of code to predict the next token (*e.g.*, at a parse error location). Unfortunately, these methods ignore the high-level structure of the language (or must learn it from vast amounts of data) and hence, lack accuracy in real-world contexts. For example, state-of-the-art methods such as [2] parse and repair only *32%* of real student code with up to 3 syntax errors while [133] repair only *58%* of syntax errors in a real-world dataset.

In this chapter, we present SEQ2PARSE, a new language-agnostic approach to automatically repairing parse errors based on the following key insight. Symbolic EC-Parsers [5] can, in principle, synthesize repairs, but, in practice, are overwhelmed by the many error-correction rules that are not *relevant* to the particular program that requires repair. In contrast, Neural approaches

are fooled by the large space of possible sequence level edits, but can precisely pinpoint the set of EC-rules that are *relevant* to a particular program. Thus, SEQ2PARSE addresses the problem of parse error repair by a neurosymbolic approach that combines the complementary strengths of the two lines of work via the following concrete contributions.

1. Motivation. Our first contribution is an empirical analysis of a real-world dataset of more than a million novice Python programs that shows that parse errors constitute the majority of novice errors, take a long time to fix, and that the fixes themselves can often require multiple edits. This analysis clearly demonstrates that an automated tool that suggests parse error repairs in a few seconds could greatly benefit many novices (§ 3.2).

2. Implementation. Our second contribution is the design and implementation of SEQ2PARSE, which exploits the insight above to efficiently and accurately suggest repairs in a neurosymbolic fashion: (1) train sequence classifiers to predict the *relevant* EC-rules for a given program (§ 3.5), and then (2) use the predicted rules to synthesize repairs via EC-Parsing (§ 3.6).

3. Abstraction. Predicting the rules is challenging. Standard NLP token-sequence based methods are confused by long trailing contexts that are independent of the parse error. This confusion yields to inaccurate classifiers that predict irrelevant rules yielding woefully low repair rates. Our second key insight eliminates neural confusion via a symbolic intervention: we show how to use Probabilistic Context Free Grammars (PCFGs) to *abstract* long low-level token sequences so that the irrelevant trailing context is compressed into single non-terminals, yielding compressed abstract token sequences that can be accurately understood by DNNs (§ 3.4).

4. Evaluation. Our final contribution is an evaluation of SEQ2PARSE using a dataset of more than 1,100,000 Python programs that demonstrates its benefits in three ways. First, we show its *accuracy*: SEQ2PARSE correctly predicts the right set of error rules 81% of the time when

considering the top 20 rules and can parse 94% of our tests within 2.1 seconds with these predictions, a significant improvement over prior methods which were stuck below a 60% repair rate. Second, we demonstrate its *efficiency*: SEQ2PARSE parses and repairs erroneous programs within 20 seconds 83% of the time, while also generating *the user fix in almost 1 out 3 of the cases*. Finally, we measure the *quality* of the generated repairs via a human study with 39 participants and show that humans perceive both SEQ2PARSE's edit locations and final repair quality to be useful and helpful, in a statistically-significant manner, even when not equivalent to the user's fix (§ 3.7).

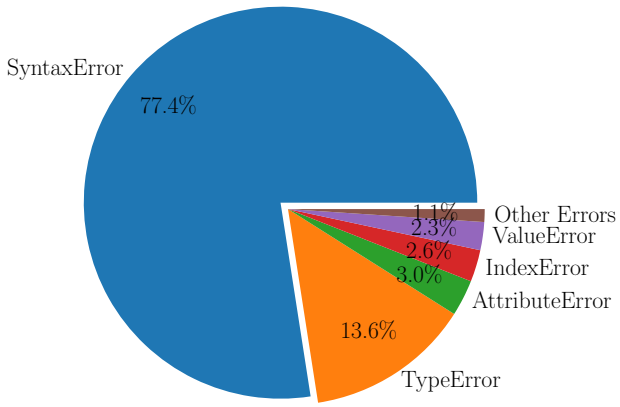


Figure 3.1: The Python error type distribution.

3.2 A Case for Parse Error Repair

We motivate SEQ2PARSE by analyzing a dataset comprising *1,100,000 erroneous Python programs* and their respective fixes. This dataset was gathered from PythonTutor.com [40] between the years 2017 and 2018, previously used in related work [21, 27]. Each program which throws an uncaught PYTHON exception is paired with the next program by the same user that does not crash, under the assumption that the latter is the fixed version of the former. We discard pairs that are too different between buggy and fixed versions, since these are usually unrelated submissions or complete refactorings. We also discard submissions that violate PythonTutor's policies (*e.g.*, those using forbidden libraries). The resulting dataset contains usable program pairs, representing students from dozens of universities (PythonTutor has been used in many introductory courses [40]) as well as non-traditional novices.

One might imagine that parse (or *syntax*) errors are usually easier to locate and repair than other algorithmic or runtime errors [22]. For example, the Python parser will immediately inform the programmer about missing parentheses in function argument lists or improper indentation. However, as has also been shown in previous work [1, 63], our data confirm that programmers (especially novices) deal with these kinds of errors regularly and spend a considerable amount of

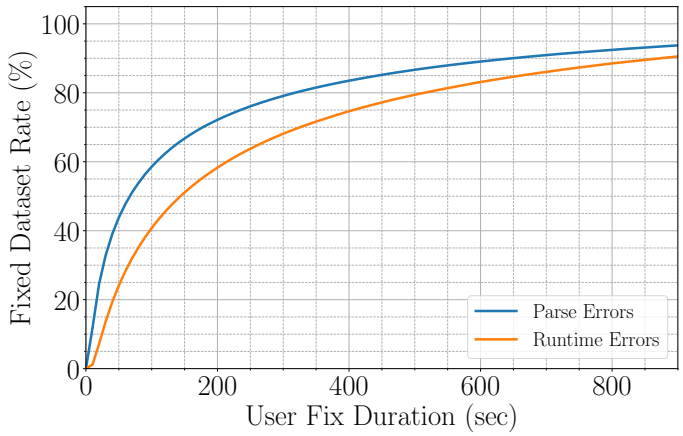


Figure 3.2: The repair rates of the Python dataset.

time fixing them.

Observation 1: Parse errors are very common. Figure 3.1 presents the statistics of the different types of errors that users encountered in this dataset. We observe that 77.4% of all faulty programs failed with a syntax error, accounting for the vast majority of the errors that (novice) programmers face with their programs. The second category is merely 13.6% of the dataset and represents Python type errors. This is a strong indication that parse errors are a very common category of error.

Observation 2: Parse errors take time to fix. The web-based compiler used to obtain this dataset provides *server timestamps*. The timestamp is associated with each program attempt submission, erroneous or not. The *repair time* is the difference between the erroneous and fixed program timestamps. This timing can be imprecise, as there are various reasons these timings may be exaggerated, (*e.g.*, users stepping away from the computer, internet lag *etc.*). However, in aggregate, due to the large dataset size, these timings can still be viewed as an approximate metric of the time it took novice programmers to repair their errors.

Figure 3.2 shows the *programmer repair rate*, *i.e.* the dataset percentage that is repaired under a given amount of time. It presents the repair rate for parse errors and the rest of the error

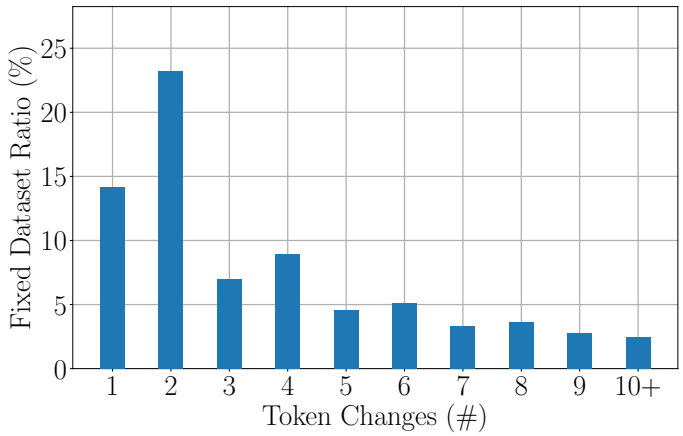


Figure 3.3: The Python dataset ratio that is fixed under the given number of token changes.

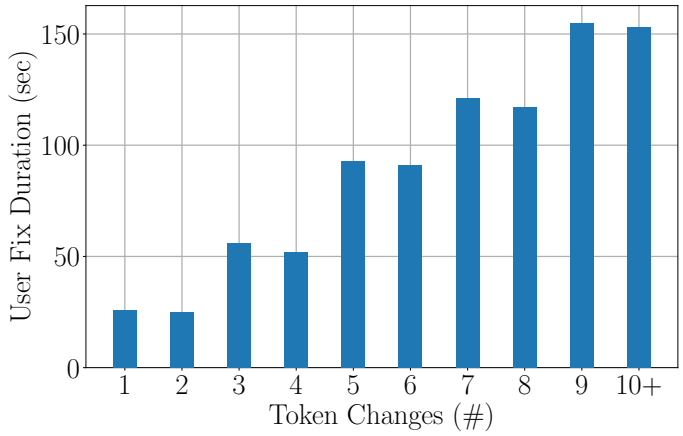


Figure 3.4: The average time the user needed to fix the erroneous program for the needed token changes.

types, grouped together here as *runtime* errors. As expected, parse errors are fixed faster than the rest, but *not by a large difference*. For example, we observe that within 2 minutes, 46% of the runtime errors are repaired, while 63% of the syntax errors are. Although this is a non-trivial difference, we observe that there are still many "simpler" parse errors that required more than 2 minutes to fix.

Observation 3: Parse errors may need multiple edits to fix. The average *token-level changes* needed to fix a program with syntax errors, *i.e.* the number of changes in the lexed program token

sequence, is *10.7 token changes*, while the *median is 4*. (This does not count lexeme content changes, such as variable renamings, and thus underapproximates the work required.) As shown in Figure 3.3, 14.2% of errors need only 1 token change, 23.2% need 2 token changes, 7.0% need 3 and 9.0% need 4. Ultimately, 46.6% of these errors require more than 4 token changes.

Observation 4: Parse errors with more edits take longer to fix. Figure 3.4 shows the average time for users to fix syntax errors as a function of the number of token changes needed. As expected, with an increasing number of token changes needed, programmers need more time to implement those changes. Most importantly, even for 1 or 2 token changes the average user spends *25 sec*, which is still a considerable amount of time for such simple and short fixes. The repair time jumps to *56 sec* for three token changes.

These four observations indicate that, while some errors can be easily and quickly fixed by programmers using existing error messages, there are many cases where novices struggle with fixing syntax errors. Therefore, we can conclude that an automated tool that parses and repairs such programs in only a few seconds could benefit many novices.

```
1  def foo(a):
2    return a + 42
3
4  def bar(a):
5    b = foo(a) + 17
6    return b +
```

(a) A Python program with two functions that manipulate an integer. The second one has a parse error.

```
1  def foo(a):
2    return a + 42
3
4  def bar(a):
5    b = foo(a) + 17
6    return b
```

(b) A fixed version for the previous example that has no parse errors.

Figure 3.5: A Python program example with syntax errors (left) and its fix (right).

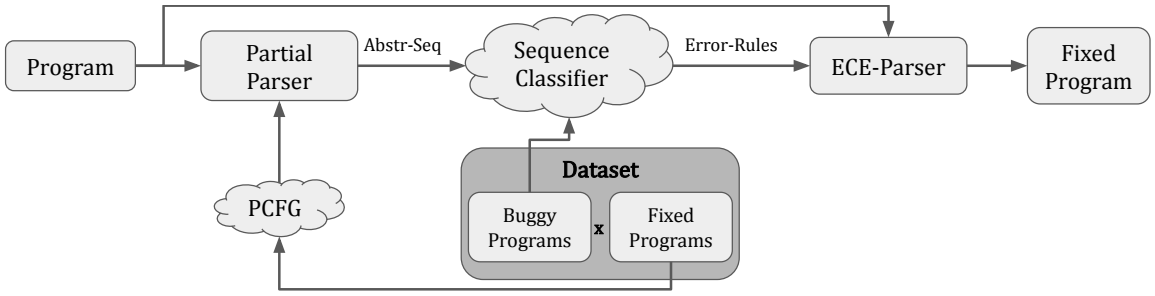


Figure 3.6: SEQ2PARSE's overall approach.

3.3 Overview

We begin with an overview of SEQ2PARSE's neurosymbolic approach to repairing parse errors, that uses two components. *(Neural)* Given a dataset of ill-parsed programs and their fixes, we partially parse the programs into *abstract sequences* of tokens (§ 3.3.2), that can be used to train *sequence classifiers* (§ 3.3.3), that predict program-relevant error rules for new erroneous programs (§ 3.3.4). *(Symbolic)* Next, given an erroneous program, and a (small) set of predicted *program relevant* error rules, the ECE-parser can exploit the high-level grammatical structure of the language to make short work of synthesizing the best repair (§ 3.3.1). We now give an overview of SEQ2PARSE, describing these elements in turn, using as a running example, the program in Figure 3.5a where the programmer has introduced an extra `+` operator after the **`return`** `b` on line 6. This extra `+` should be deleted, as shown in the developer-fixed program in Figure 3.5b.

```
S         → Stmts end_marker
Stmts     → Stmt \n | Stmt \n Stmts
Stmt      → FuncDef | ExprStmt
           | RetStmt | PassStmt | ...
FuncDef   → def name Params : Block
Block     → \n indent Stmts dedent
RetStmt   → return | return Args
Args      → ExprStmt | ExprStmt , Args
ExprStmt  → ArExpr | ...
ArExpr    → Literal
           | ArExpr BinOp Literal
Literal   → name | number | ...
```

Figure 3.7: Simplified Python production rules.

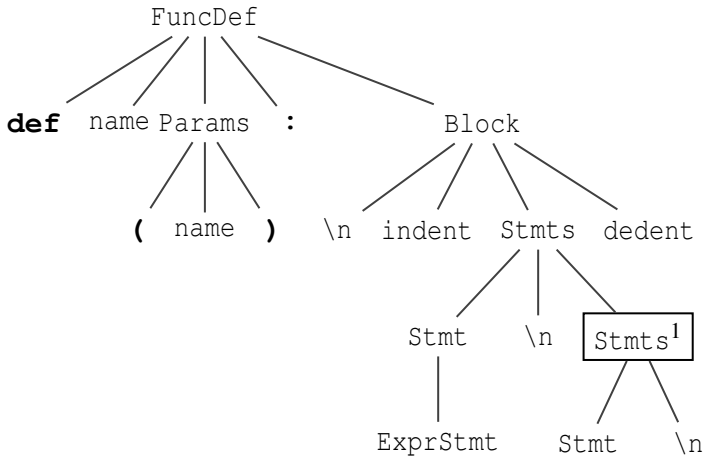


Figure 3.8: The partial parse tree generated for `bar` in the example at Figure 3.5a

3.3.1 Error-Correcting Parsing

Earley Parsers for Python. An Earley parser accepts programs that belong to a language that is defined by a given *grammar G* by using dynamic programming, to store top-down partial parses in a data structure called a *chart* [25]. The grammar *G* has a starting symbol `S` and a set of *production rules*. Figure 3.7 presents some simplified production rules for the Python programming language that will help parse the program in Figure 3.5a. *Terminal* symbols (or *tokens*) are syntactical symbols and are here presented in lowercase letters. Uppercase letters denote *non-terminal* symbols, which are rewritten using production rules during a parse. For example, the non-terminal `Stmt` defines all possible Python statements, including expressions

(`ExprStmt`), return statements (`RetStmt`), *etc.* Figure 3.8 shows the top levels of the parse tree for the `bar` function in Figure 3.5a using these productions rules.

Error-Correcting Parsers for Python. An *Error-Correcting Earley* (ECE) Parser extends the original algorithm's operations, to find a *minimum-edit* parse for a program with parse errors [5]. An ECE-Parser extends the original grammar G with a set of *error production rules* to create a new *error grammar* G' which has rules to handle *insertion*, *deletion*, and *replacement* errors. Let's see how to adapt Python's production rules for an ECE-Parser. First, the ECE-Parser adds to G' a new start symbol `New_S`, the helper symbol `Replace` that is used for replacement errors and the symbols `Insert` and `Token` that introduce insertion errors. Additionally, for each terminal `t` in G it adds the new non-terminal `E_t` that introduces errors relevant to the `t` symbol.

Next, in addition to the existing production rules, the error grammar G' has the following error rules. The new start symbol uses the old one with the option of an insertion error at the end:

- `New_S` → `S | S Insert`

Additionally, for each production rule of a non-terminal `T` in G, another non-terminal error rule is added that introduces the terminal symbols `E_t`, for each original terminal `t` it has. For example, the `Stmts`, `Block` and `RetStmt` rules are updated as:

- `Stmts` → `... | Stmt E_\n | Stmt E_\n Stmts`
- `Block` → `... | E_\n E_indent Stmts E_dedent`
- `RetStmt` → `... | E_return | E_return Args`

Next, for each terminal `t` in G, we add four error rules of the type:

- `E_t` → `t | ε | Replace | Insert t`

These four new error rules have the following usage for each terminal `t`:

1. The `E_t` $\rightarrow$ `t` rule will match the original terminal `t` without any errors. This error rule is used in cases that the *non-error* version of the rule is needed. For example, in `Block` $\rightarrow$ `E_\n E_indent Stmts E_dedent` it can be the case that only `E_dedent` is needed to match the error and `E_\n` and `E_indent` can match their respective symbols.

2. Using `E_t` $\rightarrow$ ε a *deletion* error is considered. The error rule will match *nothing*, or the *empty token* ε, in the program, meaning the terminal is missing.

3. Using `E_t` $\rightarrow$ `Replace` a *replacement* error is considered. `Replace` will match any terminal token that is *different* than `t`, making a replacement possible.

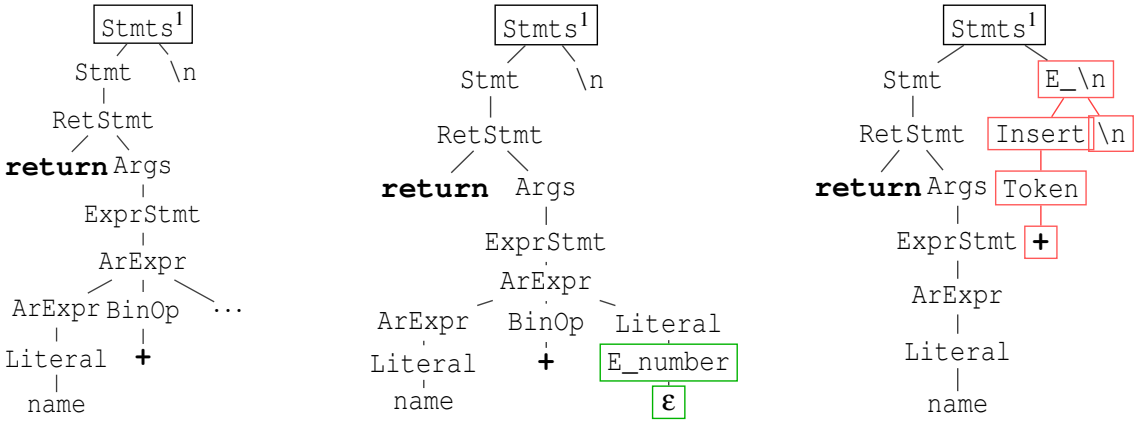
4. The rules `E_t` $\rightarrow$ `Insert t` will introduce an *insertion* error, *i.e.* `Insert` will match any *sequence* of `Token`s that are not supposed to precede `t` in order to make the program parse.

For example, for the terminal tokens **`return`**, `number` and `\n` (a new line) the relevant error production rules are:

- `E_return` $\rightarrow$ **`return`** | ε | `Replace` | `Insert` **`return`**
- `E_number` $\rightarrow$ `number` | ε | `Replace` | `Insert number`
- `E_\n` $\rightarrow$ `\n` | ε | `Replace` | `Insert \n`

Finally, the `Replace` non-terminal can match any possible terminal in G to introduce replacement errors, the `Insert` non-terminal will introduce a sequence of insertion errors by using `Token` which also matches every terminal and we just differentiate the name in order to be able to distinguish the different types of errors.

- `Replace` $\rightarrow$ **`return`** | **`pass`** | `\n` | `+` | `...` [**`all`** `terminals`]
- `Insert` $\rightarrow$ `Token` | `Insert Token`
- `Token` $\rightarrow$ **`return`** | **`pass`** | `\n` | `+` | `...` [**`all`** `terminals`]



(a) The partial parse tree for the example at Figure 3.5a.

(b) Adding a number with the green `E_number` error rule.

(c) Deleting the + with red `E_\n` error rule.

Figure 3.9: The rest of the problematic function in Figure 3.8 and two possible error-correcting parses

ECE Parsing Considerations for Python. Unfortunately, we run into various problems if we try to directly use an ECE-Parser for large, real-world languages like Python. Figure 3.9a presents a *partial parse* of the problematic statements `Stmts`[1] of Figure 3.8. Considering a deletion error (Figure 3.9b), the `E_number` $\rightarrow$ ε error rule is used to match the empty symbol and generate a parse that suggests that a *number* is missing after the + operator. On the other hand, the `E_\n` $\rightarrow$ `Insert \n` error rule can be used to consider an insertion error (Figure 3.9c) before the new line character, basically *deleting* the + operator. In this case, `ArExpr` $\rightarrow$ `Literal` is used to parse the program instead of `ArExpr` $\rightarrow$ `ArExpr BinOp Literal`.

The ECE-Parser is an effective approach on finding minimum distance parses for programs than do not belong in the given programming language, *i.e.* have parse errors. However, this parsing algorithm has limited use in large real-world programming languages, *e.g.* Python or Java, and more time- and memory-efficient parsing algorithms are often used, *e.g.* LR parsing *etc.* [16, 59]. For example, Python has 91 *terminal symbols* (including the program's `end_marker`) which means that for all the cases of the error rules `E_t` (excluding the non-error base case `E_t` $\rightarrow$ `t`), `Replace` and `Token`, 455 *terminal error rules* have to be added to the grammar G'. The Python grammar that we used has also 283 *production rules*, from which 182 *rules* have terminals

in them, meaning another 182 *error rules* need to be added. Including the four helper production rules, *e.g.* for the new start symbol, the new grammar G' has 641 *new error production rules*. This large amount of error rules renders the ECE-Parser not scalable for large programs or programs with a lot of parse errors when using real-world programming languages.

One of our insights, as seen in our running example in Figure 3.9, is that only a handful of error rules are relevant to each parse error. Therefore, we propose to improve ECE-Parsing's scalability by only adding *a small set* of error production rules, *i.e.* keeping the size of G' limited and only slightly larger than the original grammar G. We propose to do so by *training classifiers* to select a small set of error rules only relevant to the parse error. However, the program token sequences that we can use may have irrelevant information, *e.g.* the `foo` function in our example in Figure 3.5 that does not contribute to the parse error. To address this problem, we propose to further *abstract* our program token sequences.

3.3.2 Abstracting Program Token Sequences

As shown in Figure 3.6, our neural component has the task of training a classifier to predict the relevant error rules for a given ill-parsed program.

Problem: Representing Ill-parsed Programs. As the inputs are ill-parsed, the training and classification cannot use any form of analysis that requires a syntax tree as input [39, 78, 106, 130]. One option is to view the ill-parsed program as a plain *sequence of tokens* eliding variable names and such, as shown in Figure 3.10a. Unfortunately, we found such token sequences yielded inaccurate classifiers that were confused by irrelevant trailing context and predicted rules that were not relevant to repair the error at hand.

Solution: Abstract with Partial Parses. SEQ2PARSE solves the problem of irrelevant context by *abstracting* the token sequences using *partial parses* to abstract away the irrelevant context.

```
1 def name(name): \n
2 indent return name + number \n
3 dedent \n
4
5 def name(name): \n
6 indent name = name(name) + number \n
7 return name + \n
8 dedent end_marker
```

(a) The program token sequence generated by the lexer.

```
1 Stmt \n
2
3 def name Params: \n
4 indent Stmt \n
5 return Expr BinOp \n
6 dedent end_marker
```

(b) The abstracted token sequence for the same program. Parts of the program that can't be abstracted (*e.g.* **def** name) remain the same.

Figure 3.10: The token sequences for the Python program example in Figure 3.5.

That is, we can use partial parse trees to represent ill-parsed programs as an *abstracted token sequence* shown in Figure 3.10b, where any *completed* production rules can be used to abstract the relevant *token sub-sequences* with the high-level *non-terminal*.

Figure 3.8 shows how partial parses can be used to abstract long low-level sequences of tokens into short sequences of non-terminals. (1) The function `foo` is completely parsed, since it had no parse errors and the highest level rule that can be used to abstract it is `Stmt` $\rightarrow$ `FuncDef`. (2) Similarly, note that `Params` $\rightarrow$ `( name )` is another completed production rule, therefore the low-level sequence of parameter tokens in the `bar` function can be abstracted to just the non-terminal `Params`. (3) However, the production rule for `FuncDef` is *incomplete* since the last statement `Stmt` (under `Stmts`[1]) has a parse error as shown in Figure 3.9a.

Problem: Ambiguity. The generation of this abstraction, however, poses another difficulty. Earley parsing collects a large amount of partial parses (via dynamic programming) until the program is fully parsed. That means at each program location, multiple partial parses can be chosen to abstract our programs. This *ambiguity* can be seen even in the two suggested repairs in Figure 3.9: if we delete the colored nodes in Figure 3.9b and Figure 3.9c we obtain two possible partial parses for our program, the first one matching Figure 3.9a and the second one not shown here.

```
S         → Stmts end_marker (p = 100.0%)
Stmts     → Stmt \n (p = 38.77%) | Stmt \n Stmts (p = 61.23%)
Stmt      → ExprStmt (p = 62.64%) | RetStmt (p = 7.59%) | ...
RetStmt   → return (p = 1.61%) | return Args (p = 98.39%)
Args      → ExprStmt (p = 99.20%) | ...
ExprStmt  → ArExpr (p = 29.40%) | ...
ArExpr    → Literal (p = 86.89%) | ArExpr BinOp Literal (p = 13.11%)
Literal   → name (p = 64.89%) | number (p = 20.17%) | ...
```

Figure 3.11: The production rules shown in Figure 3.7 with their learned probabilities.

Solution: Probabilistic Context-Free Grammars. SEQ2PARSE solves the ambiguity problem of choosing between multiple possible partial parses via a data-driven approach based on *Probabilistic Context-Free Grammars* which have been used in previous work to select *complete* parses for ambiguous grammars [19, 50]. A PCFG associates each of its production rules with a *weight* or *probability*. These weights can be learned [19] by using the data set to count the production rules used to parse a number of programs belonging to that language. SEQ2PARSE uses PCFGs to resolve the ambiguity of partial parses by associating each partial tree (in the Earley table) with a probability which is the *product* of the used rules' probabilities. The tree with the highest probability is selected as a final parse tree which can then be used to generate an abstracted token sequence, as described above.

Figure 3.11 shows the learned probabilities for the example Python grammar. We observe, for example, that `ReturnStmt` has two possible production rules and almost 98.4% of the times a **`return`** is followed by an argument list. Additionally, 62.6% of the times a `Stmt` is an `ExprStmt` and only 7.6% of the times it is a `RetStmt`. Thus, in our example, the probability that would be assigned to the partial parse for $\texttt{Stmts}^1$ in Figure 3.9b (only the sub-tree without the colored error nodes) is the product of the probabilities of the production rules `Stmts` → `Stmt \n`, `Stmt` → `RetStmt`, `RetStmt` → **`return`** `Args`, `Args` → `ExprStmt` *etc.* which is $38.77\% \cdot 7.59\% \cdot 98.39\% \cdot 99.20\% \cdot \cdots = 4.57‰$, while the partial parse for $\texttt{Stmts}^1$ in Figure 3.9c would similarly be calculated as $47.61‰$, making it the proper choice for the abstraction of the program.

3.3.3 Training Sequence Classifiers

The abstracted token sequences we extracted from the partial parses present us with short abstracted sequences that abstract irrelevant details of the context into non-terminals. Next, SEQ2PARSE uses the NLP approach of *sequence models* [44, 119] to (use the abstract token sequences) to train a classifier that can predict the relevant error rules.

Seq2Seq Architectures. Sequence-to-sequence (seq2seq) architectures transform an input sequence of tokens into a new sequence [119] and consist of an *encoder* and a *decoder*. The encoder transforms the input sequence into an *abstract vector* that captures all the essence and context of the input. This vector does not necessarily have a physical meaning and is instead an internal representation of the input sequence into a higher dimensional space. The abstract vector is given as an input to the decoder, which in turn transforms it into an output sequence.

SEQ2PARSE uses a *sequence classifier* that can correctly predict a small set of relevant error production rules for a given abstracted token sequence. We use a *transformer encoder* [125] to encode the input sequences into abstract vectors that we then feed into a DNN classifier to train and make accurate predictions [109].

Training From a Dataset. Given a dataset of fixed parse errors, such as Figure 3.5, we extract the small set of relevant error rules needed for each program to make it parse with an ECE-Parser. Running the ECE-Parser on every program in the dataset with the *full set* of error production rules is prohibitively slow. Therefore, we extract the erroneous and fixed program token-level differences or *token diffs* and map them to *terminal error production rules*. The non-terminal error rules can be inferred using the grammar and the terminal rules. Next, we run the ECE-Parser with the extracted error rules to confirm which ones would make the program parse and assign them as *labels*.

For example, the diff for the program pair in Figure 3.5 would show the deleted + operator,

thus extracting the error rules `Token` → `+` and `E_\n` → `Insert \n`, since the extra `+` precedes a newline character `\n`. Similarly, if a token `t` is added in the fixed program, the error rule `E_t` → ε is added and if a token `t` replaces a token `a`, the error rules `E_t` → `Replace` and `Replace` → `a` are added.

3.3.4 Predicting Error Rules with Sequence Classifiers

The learned sequence classifier model, which has been trained on the updated error-rule-labeled data set can now be used to predict the relevant rules for new erroneous programs. Additionally, neural networks have the advantage of associating each class with a *confidence score* that can be used to rank error rule predictions for new programs, letting us select the top-*N* ones that will yield accurate repairs when used with the ECE-Parser.

For our running example, in Figure 3.5a, we abstract the program as shown in Figure 3.10b and then we predict the error production rules for it with the trained sequence classifier. We rank the full set of *terminal* error rules based on their predicted confidence score from the classifier and return the *top 10* predictions for our example. Therefore, the predicted set of error rules is the following: `E_number` → ε, `E_number` → `Insert number`, `E_\n` → `Insert \n`, `E_(` → ε, `E_return` → ε, `Token` → `)`, `Token` → `+`, `Token` → `:`, `Token` → `name`, `Token` → `number`.

The classifier predicts mostly relevant error rules such as the ones that use `E_number`, `E_\n` and `E_return` for example, as we showed previously. There are also rules that are not very relevant to this parse error but the classifier predicts probably due to them being common parse errors, *e.g.* `Token` → `)`, `Token` → `:`. Finally, we added the *non-terminal* error rules needed to introduce these errors, which can be inferred by them. For example, we can infer `Stmts` → `Stmt E_\n`, `Stmts` → `Stmt E_\n Stmts` and `Block` → `E_\n indent Stmts dedent` from `E_\n` (we don't need `E_indent` or `E_dedent` here since no such terminal error rules were predicted).

We then parse the program in Figure 3.5a with the ECE-Parser and these specific error rules to generate a valid parse. We observe that it takes our implementation (as we show later

in depth) less than *2 seconds* to generate a valid parse, which is also the one that leads to the user repair in Figure 3.5b! On the other hand, when we use a baseline ECE-Parser with the full set of error rules it takes *2 minutes and 55 seconds* to generate a valid parse, which is, however, not the expected user parse but the one shown in Figure 3.9b. These examples demonstrate the effectiveness of accurately *predicting error rules using sequence classifiers, which are trained on abstracted token sequences*.

In the next three sections, we describe in depth the specifics of our approach by defining all the methods in Figure 3.12. We start by presenting the program abstraction (section 3.4) using partial parses and a learnt PCFG, we then explain how we train sequence models for making error rule predictions (section 3.5) and, finally, we demonstrate our algorithms for building SEQ2PARSE (section 3.6), an approach for efficiently parsing erroneous programs.

G	$\doteq$	(N, Σ, P, S)
PCFG	$\doteq$	(N, Σ, P, S, W)
G'	$\doteq$	(N', Σ, P', S')
P'	$\doteq$	$P \cup$ ErrorRules
N'	$\doteq$	$N \cup \{S', H, I\} \cup \{E_a \mid a \in \Sigma\}$
e	$\in$	$L(G)$
$e_\perp$	$\notin$	$L(G)$
Pair	$\doteq$	$e_\perp \times e$
DataSet	$\doteq$	[Pair]

§ 3.4	learnPCFG	:	$G \to [e] \to$ PCFG
	partialParse	:	PCFG $\to e_\perp \to t^a$
§ 3.5	trainDL	:	$[t^a \times$ ErrorRules$] \to$ DLModel
	predictDL	:	DLModel $\to t^a \to$ ErrorRules
§ 3.6	diffRules	:	Pair $\to$ ErrorRules
	ECEParse	:	ErrorRules $\to e_\perp \to e$
	train	:	DataSet $\to$ DLModel
	predict	:	DLModel $\to G \to e_\perp \to$ ErrorRules
	Seq2Parse	:	$G \to$ DataSet $\to (e_\perp \to e)$

Figure 3.12: A high-level API of the SEQ2PARSE system that learns to repair syntax errors.

3.4 Abstracting Programs with Parse Errors

SEQ2PARSE abstracts programs (with parse errors) into sequences of *abstract* tokens that are used to train sequence classifiers. Next, we explain how a traditional Earley parser can be used to extract *partial* parses using a Probabilistic Context-Free Grammar (PCFG), to get a higher level of abstraction that preserves more contextual information than the low-level sequence output by the lexer.

Lexical Analysis. *Lexical analysis* (or lexing or tokenization) converts a sequence of characters into a sequence of tokens comprising strings with an assigned and thus identified meaning (*e.g.* numbers, identifiers *etc.*). Lexing is usually combined with a parser, which together analyze the syntax of a programming language $L(G)$, defined by the grammar G. When a program has a syntax error, the output token sequence of the lexer is the highest available level of abstraction as, since the parser fails without producing a parse tree.

Token Sequences. Our goal is to parse a *program token sequence* t^i, which is a lexed program *with* parse errors (*i.e.* $t^i \notin L(G)$), and repair it into a *fixed token sequence* $t^o \in L(G)$ that can be used to return a repaired program without syntax errors. Let t^i be a sequence $t^i_1, t^i_2, \ldots, t^i_n$ and t^o be the updated sequence $t^o_1, t^o_2, \ldots, t^o_i, \ldots, t^o_j, \ldots, t^o_k$. The subsequence $t^o_i, \ldots, t^o_j$ can either

replace a subsequence in t^i, it can *be inserted* in t^i or can be the empty subsequence ε and *delete* a subsequence in t^i to generate the t^o. It can be the whole program, part of it or multiple parts of it. t^o will finally be a token sequence that can be parsed by the original language's $L(G)$ parser.

However, programs and hence, n can be large which makes these token sequences unsuitable for training effectively sequence models. Therefore, our goal is to first generate an *abstracted token sequence* t^a that removes all irrelevant information from t^i and gives hints for the parse error fix by using the internal states of an *Earley* parser.

3.4.1 Earley Partial Parses

SEQ2PARSE uses an *Earley parser* [25] to generate the abstracted token sequence t^a for an input program sequence t^i. An Earley parser holds internally a *chart* data structure, *i.e. a list of partial parses*. Given a production rule $X \to \alpha\beta$, the notation $X \to \alpha \cdot \beta$ represents a condition in which α has already been parsed and β is expected and both are sequences of terminal and non-terminal symbols (tokens).

Each state is a tuple $(X \to \alpha \cdot \beta,\ j)$, consisting of

- the production rule currently being matched $(X \to \alpha\beta)$
- the current position in that production (represented by the dot $\cdot$)
- the origin position j in the input at which the matching of this production began

We denote the state set at an input position k as $S(k)$. The parser is seeded with $S(0)$ consisting of only the top-level rule $S \to \gamma$. It then repeatedly executes three operations: *prediction, scanning,* and *completion*. There exists a *complete parse* if the complete top-level rule $(S \to \gamma\cdot, 0)$ is found in $S(n)$, where n the input length. We define a *partial parse* to be any partially completed rules, *i.e.* if there is $(X \to \alpha \cdot \beta, i)$ in some state $S(k)$, where $i < k \leq n$.

Let, $t^i_1, t^i_2, \ldots, t^i_j, \ldots, t^i_k, \ldots, t^i_n$ be the input token sequence t^i, where there is a parse error at location k and the parser has exhausted all possibilities and can not add any more rules in state

$S(k+1)$, *i.e.* $S(k+1) = \emptyset$. We want to abstract program subsequences $t^i_j, \ldots, t^i_k$ by getting the longest possible parts of the program t^i that have a partial parse. For example, we start from the beginning of the program t^i_1 by finding the largest j for which there is a rule $(X \to \alpha \cdot \beta, 0) \in S(j)$. We use this rule for X to replace $t^i_1, t^i_2, \ldots, t^i_j$ in t^i with α, thus getting an abstracted sequence t^a. In the same manner, we use the longest possible partial parses that we can extract from the chart to abstract $t^i_{j+1}, \ldots, t^i_k$, iteratively, until we reach the parse error at location k.

Problem: Multiple Partial Parses. However, each of the states $S(j),\ 0 \leq j \leq k$, holds a large number of partial parses and, thus, our heuristic to choose the longest possible partial parse to abstract programs may not be able to abstract the token sequence fully until the error location k, or not even until the end location n of the program that may not have any more parse errors. Additionally, there may be two or more partial parses in $S(k)$, with different lengths, *e.g.* $\{(X \to \alpha \cdot \beta,\ j),\ (X' \to \alpha' \cdot \beta',\ h)\} \in S(k),\ j \neq h$. We propose selecting the most *probable parse* with the aid of a PCFG.

3.4.2 Probabilistic Context-Free Grammars

We learn a PCFG from a large corpus of programs $[e], e \in L(G)$, that belong to a language $L(G)$, that a grammar G defines, with the learnPCFG procedure as shown in Figure 3.12. We use the learned PCFG with an augmented Earley parser in partialParse to abstract a program $e_\perp$ into a abstract token sequence t^a.

Probabilistic CFG. A PCFG can be defined similarly to a *context-free grammar* $G \doteq (N,\ \Sigma,\ P,\ S)$ as a quintuple $(N,\ \Sigma,\ P,\ S,\ W)$, where:

- N and Σ are finite disjoint alphabets of non-terminals and terminals, respectively.
- P is a finite set of production rules of the form $X \to \alpha$, where $X \in N$ and $\alpha \in (N \cup \Sigma)^*$.

- S is a distinguished start symbol in N.
- W is a finite set of probabilities $p(X \to \alpha)$ on production rules.

Given a dataset of programs $[e], e \in L(G)$ that can be parsed, let $count(X \to \alpha)$ be the number of times the production rule $X \to \alpha$ has been used to generate a final complete parse, while parsing $[e]$, and $count(X)$ be the number of times the non-terminal X has been seen in the left side of a used production rule. The probability for a production rule $X \to \alpha$ is then defined as:

$$p(X \to \alpha) = \frac{count(X \to \alpha)}{count(X)}$$

learnPCFG invokes an instrumented Earley parser to calculate all the values $count(X \to \alpha), \forall X \to \alpha : P$ and $count(X), \forall X : N$. The *instrumented parser* keeps a *global record* of these values, while parsing the dataset $[e]$ of programs. Finally, learnPCFG outputs a PCFG that is based on the original grammar G that was used to parse the dataset with the learned probabilities W.

3.4.3 Abstracted Token Sequences

Given a program $e_\perp$ with a parse error and a learned PCFG, partialParse will generate an abstracted token sequence t^a. The PCFG will be used with an *augmented Earley parser* to disambiguate partial parses and choose one, in order to produce an abstracted token sequence as described in § 3.4.1.

We augment Earley states $(X \to \alpha \cdot \beta,\ j)$ to $(X \to \alpha \cdot \beta,\ j,\ p)$, where p is the probability that $X \to \alpha \cdot \beta$ is a correct partial parse. When there are two (or more) conflicting partial parses $\{(X \to \alpha \cdot \beta,\ j,\ p),\ (X' \to \alpha' \cdot \beta',\ h,\ p')\} \in S(k)$, the augmented parser selects the partial parse with the highest probability $max(p,\ p')$. The augmented parser calculates the probability p for a partial parse $(X \to \alpha \cdot \beta,\ j,\ p)$ in the state $S(k)$, as the product $p_1 \cdot p_2 \cdot \cdots \cdot p_{k-1}$ of the probabilities

$p_1,\ p_2,\ \ldots,\ p_{k-1}$ that are associated with the production rules $(X_1 \to \alpha_1 \cdot \beta_1,\ i_1,\ p_1), (X_2 \to \alpha_2 \cdot \beta_2,\ i_2,\ p_2), \ldots$ that have been used so far to parse the string of tokens α.

3.5 Training Sequence Classifiers

Our next task is to *train* a model that can predict the error production rules that are needed to parse a given program $e_\perp$ (with syntax errors) according to a given grammar G, by using its (abstracted) program token sequence t^a. We define the function predictDL which takes as input a *pre-trained sequence classifier* DLModel and an abstracted token sequence t^a and returns as output a *small subset* of ErrorRules. We train the DLModel offline with the trainDL method with a dataset $[t^a \times \mathsf{ErrorRules}]$ of token sequences t^a and the *exact small set* of error production rules ErrorRules that the ECE-Parser used to generate the *user parse*. We build our classifier DLModel using classic *Deep Neural Networks (*DNN*s)* and parts of state-of-the-art *Sequence-to-Sequence (seq2seq)* models. We leave the high-level details of acquiring the dataset of labeled token sequences and using the predictor for new erroneous programs for section 3.6. In the next few paragraphs, we summarize the recent advances in machine learning that help as build the sequence classifier.

We encode the task of learning a function that will map token sequences of erroneous programs to a small set of error production rules as a *supervised multi-class classification (MCC)* problem. A *supervised* learning problem is one where, given a labeled training set, the task is to learn a function that accurately maps the inputs to output labels and generalizes to future inputs. In a *classification* problem, the function we are trying to learn maps inputs to a discrete set of two or more output labels, called *classes*. We use a *Transformer encoder* to encode the input sequences into abstract vectors that we then directly feed into a DNN classifier to build a *Transformer classifier*.

Neural Networks. A neural network can be represented as a directed acyclic graph whose nodes are arranged in layers that are fully connected by weighted edges. The first layer corresponds to the input features, and the final to the output. The output of an internal node is the sum of the weighted outputs of the previous layer passed to a non-linear *activation function*, which in

recent work is commonly chosen to be the rectified linear unit (ReLU) [85]. In this work, we use relatively *deep neural networks* (DNN) that have proven to make more accurate predictions in recent work [109]. A thorough introduction to neural networks is beyond the scope of this work [45, 87].

Sequence Models. *Seq2seq* models aim to transform input sequences of one domain into sequences of another domain [119]. In the general case, these models consist of two major layers, an *encoder* and a *decoder*. The encoder transforms an input token sequence $x_1, x_2, \ldots, x_n$ into an *abstract vector* $V \in \mathbb{R}^k$ that captures all the essence and context of the input sequence. This vector does not necessarily have some physical meaning and is just an internal representation of the input sequence into a higher dimensional space. The abstract vector is then given as an input to the decoder, which in turn transforms it into an output sequence $y_1, y_2, \ldots, y_n$.

The simplest approach historically uses a Recurrent Neural Network (RNN) [103, 131], which is a natural next step from the classic neural networks. Each RNN unit operates on each input token x_t separately. It keeps an internal *hidden state* h_t that is calculated as a function of the input token x_t and the previous hidden state h_{t-1}. The output y_t is calculated as the product of the current hidden state h_t and an output weight matrix. The activation function is usually chosen as the standard *softmax* function [11, 37]. Softmax assigns probabilities to each output that must add up to 1. Finally, the loss function at all steps of the RNN is typically calculated as the sum of the cross-entropy loss of each step.

Transformers. The Transformer is a DNN architecture that deviates from the recurrent pattern (*e.g.*, RNNs) and is solely relying on *attention mechanisms*. Attention has been of interest lately [8, 57, 125] mainly due to its ability to detect dependencies in the input or output sequences regardless the distance of the tokens. The nature of this architecture makes the Transformer significantly easier to parallelize and thus has a higher quality of predictions and sequence translations after a shorter training period.

The novel architecture of a Transformer [125] is structured as a *stack of N identical layers*. Each layer has two main sub-layers. The first is a *multi-head self-attention mechanism*, and the second is a position-wise fully connected neural network. The output of each sub-layer is $LayerNorm(x+SubLayer(x))$, where $SubLayer(x)$ is the function implemented by each sub-layer, followed by a residual connection around each of the two sub-layers and by layer normalization $LayerNorm(x)$. To facilitate these residual connections, all sub-layers in the model, as well as the input *embedding layers*, produce outputs of the same dimension d_{model}.

Transformer Classifier. For our task, we choose to structure DLModel as a *Transformer Classifier*. We use a state-of-the-art Transformer encoder to represent an abstracted token sequence t^a into an abstract vector $V \in \mathbb{R}^k$. The abstract vector V is then fed as input into a multi-class DNN. We use trainDL to train the DLModel given the training set $[t^a \times \mathsf{ErrorRules}]$. The binary cross-entropy loss function is used per class to assign the loss per training cycle. DLModel predicts error production rules for a new input program t^a. Critically, we require that the classifier outputs *confidence scores* $\mathcal{C}$ that measure how sure the classifier is that a rule can be used to parse the associated input program $e_\perp$. The predictDL function uses the trained DLModel to predict the confidence scores $[\mathsf{ErrorRules} \times \mathcal{C}]$ for all error production rules ErrorRules for a new unknown program $e_\perp$ with syntax errors. The ErrorRules are then sorted based on their predicted confidence score $\mathcal{C}$ and finally the *top-N* rules are returned for error-correcting parsing. N is a small number in the 10s that will give accurate predictions without making the ECE-Parser too slow, as we discuss in section 3.7.

3.6 Building a Fast Error-Correcting Parser

We show how SEQ2PARSE uses the abstracted token sequences from section 3.4 and the trained sequence models from section 3.5 to generate an *error-correcting parser* $(e_\perp \rightarrow e)$, that will parse an input program $e_\perp$ with syntax errors and produce a correct program e. We first describe how we extract a machine-learning-amenable training set from a corpus of fixed programs and finally how we structure everything to train our model.

3.6.1 Learning Error Production Rules

The trainDL method requires a dataset of token sequences t^a that is annotated with an *exact and small set* of error production rules, *i.e.* $[t^a \times \mathsf{ErrorRules}]$. These ErrorRules are just a subset of all the possible error rules that are needed to parse and fix t^a. The straight-forward approach is to use ECEParse with all possible error production rules for each program $e_\perp$ in the dataset. Then, when ECEParse returns with a successful parse, we extract the rules that where used to parse the program $e_\perp$. This approach generates a dataset with the smallest possible set of error rules as labels per program, since the original ECE-Parser returns the minimum-distance edit parse. However, this approach completely ignores the programmer's fix and takes an unreasonable amount of time to parse a dataset with millions of programs, due to the inefficient nature of the ECE-Parser.

We suggest using an $O(ND)$ difference algorithm [84] to get a small but still representative set of error production rules for each program $e_\perp$. We employ this algorithm to find the differences between the input *program token sequence* t^i, which is the lexed program $e_\perp$ and the *fixed token sequence* t^o, which is the lexed program e. This algorithm returns changes between token sequences in the form of *inserted or deleted tokens*. It is possible that this algorithm returns a sequence of deletions followed by a sequence of insertions, which can in turn be interpreted as a *replacement* of tokens. We map these three types of changes to the respective error production

rules. Let t^i be a sequence $t^i_1, t^i_2, \ldots, t^i_n$ and t^o be the updated sequence $t^o_1, t^o_2, \ldots, t^o_m$. We map:

- an inserted output token t^o_j to a *deletion* error $E_{t^o_j} \rightarrow \varepsilon$.
- a deleted input token t^i_k to an *insertion* error $Tok \rightarrow t^i_k$ and the helper rule $E_{t^i_{k+1}} \rightarrow Ins\ t^i_{k+1}$.
- a replaced token t^i_k with t^o_j to a *replacement* error $Repl \rightarrow t^i_k$ and the helper rule $E_{t^o_j} \rightarrow Repl$.

In the case of an insertion error, we also include the helper rules $Ins \rightarrow Tok$ and $Ins \rightarrow Ins\ Tok$, that can derive any nonempty sequence of insertions. To introduce (possible) insertion errors at the end of a program, we include the starting production rules $S' \rightarrow S$ and $S' \rightarrow S\ Ins$.

The above algorithm, so far, adds only the *terminal error productions*. We have to include the *non-terminal error productions* that will invoke the terminal ones. If $X \rightarrow a_0 b_0 a_1 b_1 \ldots a_m b_m$, $m \geq 0$, is a production in P such that a_i is in N^* and b_i is in Σ, then we add the error production $X \rightarrow a_0 X_{b_0} a_1 X_{b_1} \ldots a_m X_{b_m}$, $m \geq 0$ to P', where each X_{b_i}, is either a new non-terminal E_{b_i} that was added with the previous algorithm, or just b_i again if it was not added.

Finally, we further refine the new small set of error productions for each program $e_\perp$ with ECE-Parser, in order to create the final annotated dataset $[t^a \times \mathsf{ErrorRules}]$. The changes that we extracted from the programmers' fixes might include irrelevant changes to the parse error fix, *e.g.* code clean-up. Therefore, filtering with the ECE-Parser is still essential to annotate each program with the appropriate error production rules. We implement this error-rule-extracting approach in the function diffRules, which extracts the token differences between an erroneous program $e_\perp$ and a fixed program e and returns the appropriate error production rules.

3.6.2 Training and Using a Transformer Classifier

Given a (probabilistic) grammar G and a dataset Ds, Algorithm 3 extracts a machine-learning appropriate dataset D_{ML} in order to train a Transformer classifier $Model$ with trainDL. The classifier $Model$ can then be used to predict error rules for new erroneous programs p_{err}.

Algorithm 3 Training Seq2Parse's model DLModel

Input: Probabilistic Grammar G, DataSet Ds
Output: Classifier $Model$

```
1: procedure TRAIN(G, Ds)
2:     D_ML ← ∅
3:     for all p_err × p_fix ∈ Ds do
4:         t^a ← PARTIALPARSE(G, p_err)
5:         rules ← DIFFRULES(p_err × p_fix)
6:         D_ML ← D_ML ∪ (t_a × rules)
7:     Model ← TRAINDL(D_ML)
8:     return Model
```

Algorithm 4 Predicting error production rules with Seq2Parse's model DLModel

Input: Classifier $Model$, Probabilistic Grammar G, Program P
Output: Error Production Rules Rls

```
1: procedure PREDICT(Model, G, P)
2:     t^a ← PARTIALPARSE(G, P)
3:     Rls ← PREDICTDL(Model, t^a)
4:     return Rls
```

The dataset D_{ML} starts as an empty set. For each program pair $p_{err} \times p_{fix}$, we, first, employ partialParse with the PCFG G and an erroneous program p_{err} to extract the abstracted token sequence t^a. Second, we use the token difference algorithm diffRules to extract the specific error *rules* that fix p_{err} based on p_{fix}. The abstracted sequence t^a is annotated with the label *rules* and is added to D_{ML}. The Transformer classifier $Model$ is trained with trainDL and the newly extracted dataset D_{ML}, which is finally returned by the algorithm. Finally, the training procedure can be performed offline and thus won't affect the performance of the final program repair.

Having trained the Transformer classifier $Model$, we can now predict error rules Rls, that will be used by an ECE-Parser, by using the predict procedure defined in Algorithm 4. predict uses the same input grammar G to generate an abstracted token sequence t^a for the program P with the partialParse procedure. Finally, the predictDL procedure predicts a small set of error production rules Rls for the sequence t^a given the pre-trained $Model$.

3.6.3 Generating an Efficient Error-Correcting Parser

Algorithm 5 presents our *neurosymbolic* approach, SEQ2PARSE. This is the high-level algorithm that combines everything that we described so far in the last three sections. SEQ2PARSE first extracts the fixed programs *ps* from the dataset *Ds* to learn a probabilistic context-free grammar $PCFG$ for the input grammar G with learnPCFG. It then trains the Transformer classifier

Algorithm 5 Generating the final ECEP

Input: Grammar G, DataSet Ds
Output: Error-Correcting Parser Prs

1: **procedure** SEQ2PARSE(G, Ds)
2: $ps \leftarrow$ MAP($\lambda.p \rightarrow$ SND(p), Ds)
3: $PCFG \leftarrow$ LEARNPCFG(G, ps)
4: $Model \leftarrow$ TRAIN($PCFG$, Ds)
5: ERULEPREDICTOR $\leftarrow$ PREDICT($Model$, $PCFG$)
6: $Prs \leftarrow (\lambda.p_{err} \rightarrow$ ECEPARSE(ERULEPREDICTOR(p_{err}), p_{err}))
7: **return** Prs

$Model$ to predict error production rules. We define an error rule predictor, ERULEPREDICTOR, using the predict procedure with the pre-trained $Model$ and grammar $PCFG$. Finally, the algorithm returns the ECE-Parser Prs, which we define as a function that takes as input an erroneous program p_{err} that uses the ERULEPREDICTOR to get the set of error rules needed by ECEParse to parse and repair it.

3.7 Evaluation

We have implemented our approach in SEQ2PARSE: a system for repairing parse errors for PYTHON in its entirety. The code for SEQ2PARSE is publicly available at `https://github.com/gsakkas/seq2parse` and a simplified online demonstration is available at `http://seq2parse.goto.ucsd.edu/index.html`. Next, we describe our implementation and an evaluation that addresses four questions:

- **RQ1**: How *accurate* are SEQ2PARSE's predicted error production rules? (§ 3.7.1)
- **RQ2**: How *precisely* can SEQ2PARSE repair parse errors? (§ 3.7.2)
- **RQ3**: How *efficiently* can SEQ2PARSE repair parse errors? (§ 3.7.3)
- **RQ4**: How *useful* are SEQ2PARSE's suggested repairs? (§ 3.7.4)

Training Dataset. For our evaluation, we use the same PYTHON dataset that we used in our error data analysis in section 3.2 gathered from PythonTutor.com [40] between the years 2017 and 2018. The dataset has more than 1,100,000 usable erroneous Python programs and their respective fixes. The programs have an average length of *87 tokens*, while the abstracted token sequences have a much shorter average of *43 tokens*. We choose 15,000 random programs from the dataset for all our tests, and the rest we use as our training set.

We first learn a PCFG on the training set of fixed programs to learn the probabilities for each production rule in the *full* PYTHON *grammar*. SEQ2PARSE then extracts the abstracted token sequences for all programs in the training set. Next, while the full PYTHON grammar has *455 possible terminal error production rules*, in reality, only *340 error rules* are ever used in our dataset and are assigned as *labels*. We arrive at this set of error rules by parsing all the erroneous programs in the training set with the ECE-Parser and the "diff" error rules, as described in subsection 3.6.1.

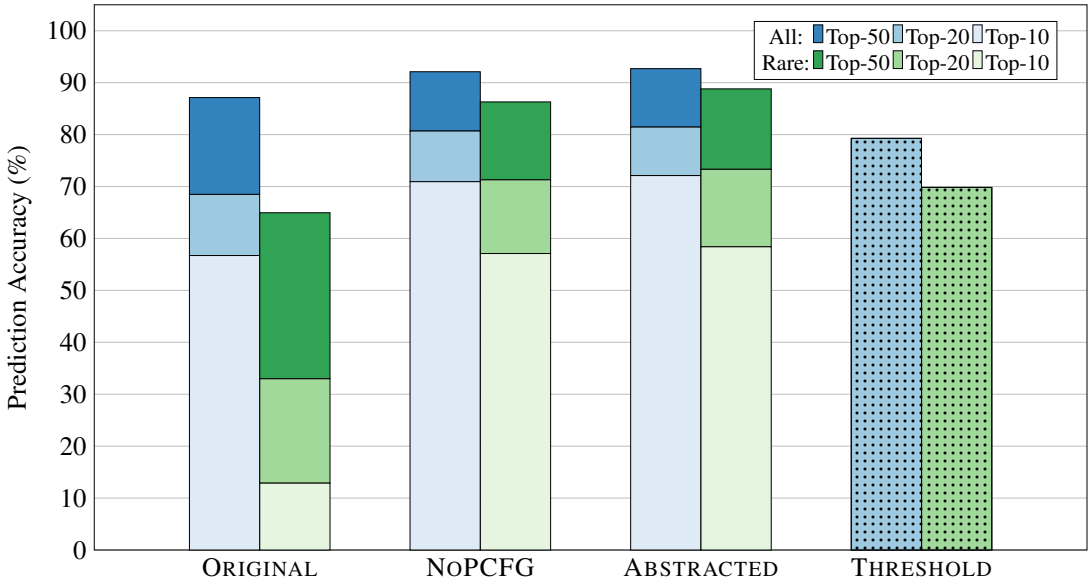


Figure 3.13: Results of our error production rule prediction classifiers for the simple original token sequences and their abstracted versions using the PCFG.

Transformer Classifier. SEQ2PARSE's error rule prediction uses a Transformer classifier with *six* transformer blocks, that each has a fully-connected hidden layer of 256 neurons and 12 attention heads. The output of the transformer blocks is then connected to a DNN-based classifier with *two* fully-connected hidden layers of 256 and 128 neurons respectively. The neurons use rectified linear units (ReLU) as their activation function, while the output layer uses the sigmoid function for each class. Additionally, there are *two input embedding layers* of a length of 128 units, one for input tokens and one for their positions in the sequence. We also limit the input abstracted token sequences to a length of 128 tokens, which covers 95.7% of the training set, without the need of pruning them. Finally, the Transformer classifier was trained using an ADAM optimizer [58], a variant of stochastic gradient descent, on NVIDIA GeForce RTX 3080 Ti for a total of 50 epochs.

3.7.1 RQ1: Accuracy

Figure 3.13 shows the accuracy results of our error production rule prediction experiments. The y-axis describes the *prediction accuracy*, *i.e.* the fraction of test programs for which the

correct *full set* of error rules to repair the program (extracted from the user fix) was predicted in the top-K sorted rules. The ORIGINAL version of our transformer classifier does not consider the abstracted token sequences and used the full ORIGINAL token sequences, whose results are presented in the first two bars of Figure 3.13. The next four bars show our final results using the ABSTRACTED token sequences to train the classifier with NOPCFG and with fully ABSTRACTED sequences. Finally, the last two dotted bars show the results for when a probability THRESHOLD is set in order to select the predicted error rules (instead of picking the static top-K ones) but using again the ABSTRACTED sequences as input. The predicted error rule set ranges between 1–20 elements.

The blue bars show the accuracy on the full test set of ALL 15,000 test programs, while the green bars show the results on a subset of RARE programs, *i.e.* programs that did not include any of the 50 most popular error rules. The RARE programs account for 1233 programs, roughly 8% of our test set.

The ORIGINAL predictor, even with the Top-50 predicted error rules, is less accurate than the Top-20 predictions of the ABSTRACTED, with an accuracy of 87.13%, which drops to 68.48% and 56.71% respectively for the Top-20 and Top-10 predictions. The ABSTRACTED predictor significantly outperforms the ORIGINAL predictor with a 72.11% Top-10 accuracy, 81.45% Top-20 accuracy and 92.70% Top-50 accuracy.

The THRESHOLD predictions are almost as accurate as the ABSTRACTED Top-20 predictions with an accuracy of 79.28% and a median number of selected error rules of 14 (average 14.1). This could potentially mean that this predictor is a valid alternative for the static Top-20 predictions.

The classifiers are also not very *sensitive* in the PCFG probabilities used during abstraction, as shown in the accuracy of the NOPCFG predictor. The NOPCFG predictor has almost 2% less Top-10 and Top-20 accuracy, 70.94% and 80.69% respectively, and less than 1% for Top-50 predictions, with 92.11%.

Finally, we observe that our ABSTRACTED classifiers generalize efficiently for our dataset of erroneous PYTHON programs and are almost as accurate for the RARE programs as the rest of the dataset with a 73.32% Top-20 accuracy (88.81% Top-50 accuracy). The same holds for the THRESHOLD predictions with a 69.83% RARE accuracy. The NOPCFG also has a drop of more than 2% accuracy, with a 71.29% Top-20 accuracy (86.29% Top-50 accuracy).

SEQ2PARSE's transformer classifier learns to encode programs with syntax errors and select candidate error production rules for them effectively, yielding *high accuracies*. By abstracting the tokens sequences, SEQ2PARSE is able to *generalize* better and make more accurate predictions with a *81.45% Top-20 accuracy*.

3.7.2 RQ2: Repaired Program Preciseness

Next we evaluate SEQ2PARSE's end-to-end accuracy and preciseness when restricting SEQ2PARSE's parsing time to *5 minutes* and run our experiments on the 15,000-program test set. Additionally, we use here the highest-performing transformer classifiers, *i.e.* the ABSTRACTED, NOPCFG and THRESHOLD classifiers.

We compare *three versions* of our SEQ2PARSE implementation (ALLPARSES, MINIMUMCOST and THRESHOLD) against two versions of the ECE-Parser with a static selection of the 20 and 50 most popular error production rules in our training set. We make this choice because we observe that the 50 most popular error rules are used as labels for as much as *86%* of the training set. For the ALLPARSES, MINIMUMCOST and THRESHOLD versions, we run our experiments for the ABSTRACTED and NOPCFG classifier predictions.

The MINIMUMCOST ECE-Parser uses the *Top-20 predictions* from our ABSTRACTED or NOPCFG classifier to parse and repair buggy programs. The ALLPARSES and THRESHOLD ECE-Parsers use the THRESHOLD classifier's predicted set of error rules to repair programs.

The ALLPARSES ECE-Parser keeps internally *all possible states* that arise from using the

Error Rule Approach	ABSTRACTED				NOPCFG			
	Parse Accuracy	*Rare Parse Accuracy*	*User Fix Accuracy*	*Median Parse Time*	*Parse Accuracy*	*Rare Parse Accuracy*	*User Fix Accuracy*	*Median Parse Time*
20 Most Popular	79.87%	65.01%	16.31%	7.0 sec	–	–	–	–
50 Most Popular	90.89%	81.26%	18.56%	13.6 sec	–	–	–	–
ALLPARSES	61.46%	59.80%	**34.57%**	7.1 *(14.2)* sec	55.52%	56.42%	30.21%	20.3 *(24.3)* sec
MINIMUMCOST	**94.25%**	**94.01%**	20.55%	5.3 *(12.9)* sec	91.63%	90.89%	17.89%	5.9 *(18.7)* sec
THRESHOLD	94.19%	93.42%	21.19%	**2.1 *(7.0)* sec**	93.70%	91.09%	19.39%	2.5 *(9.1)* sec

Figure 3.14: Experimental results of SEQ2PARSE's repair approaches. The (*parenthesized*) numbers in the Median Parse Time columns represent the median time for larger programs, *i.e.* programs with more than 100 tokens.

predicted error rules similarly to the original ECE-Parser described by [5]. We use a maximum repair cost of 3 edits (*i.e.*, a maximum of 3 insertions, deletions or replacements) to limit the search space. The MINIMUMCOST version, however, always keeps the minimum-edit repair and discards all other states that may lead to a higher cost. This more efficient version of the ECE-Parser allows for a higher maximum cost of 10 edits. We use the same ECE-Parser and cost as in MINIMUMCOST for our THRESHOLD parser. The maximum cost for each parser is a hyperparameter to SEQ2PARSE and is set here arbitrarily to achieve a uniform run time across experiments while obtaining high-quality multiple-edit repairs. Finally, MINIMUMCOST will always return the top 1 repair, while ALLPARSES can generate a large number of repairs and we select to keep only the top 5 repairs after filtering with a static code checker (PYLINT, `https://www.pylint.org/`) as most developers will consider only a few suggestions before falling back to manual debugging [61, 88].

Figure 3.14 shows the percentage of test programs that each of these five versions can parse successfully (*i.e.* the *parse accuracy*), the rare program parse accuracy, and the *user equivalent parse accuracy*, *i.e.* the amount of parses that match the one that the user compiled. We observe that the ABSTRACTED MINIMUMCOST parser *outperforms* every other option with 94.25% parse accuracy and 94.01% rare parse accuracy. It also generates the intended user parse for 20.55% of the set, *i.e.* over 1 out 5 of the cases. The 20 MOST POPULAR parser with 79.87% parse accuracy and 65.01% rare parse accuracy is much less accurate, and is 4.24% less likely to generate the user parse, while the 50 MOST POPULAR is slightly less accurate with 90.89% and

81.26% accuracy, as expected from the usage of a large number of popular error rules. The 50 MOST POPULAR parser has also a high user fix accuracy of 18.56%. The ALLPARSES parser has the lowest parse accuracy of 61.46%, however it manages to generate the user fix 34.57% of the time and also achieve a 59.80% rare accuracy. Finally, the THRESHOLD parser is almost as accurate as the efficient MINIMUMCOST parser with 94.19% and 93.42% parse and rare accuracy, while achieving a slightly higher user fix accuracy of 21.19%.

Additionally, the NOPCFG MINIMUMCOST parser achieves 2.6% lower parse accuracy than then ABSTRACTED version and 3.1% less rare parse accuracy. The NOPCFG THRESHOLD parser also performs only 0.5% less accurately than then ABSTRACTED version and has a 2.3% less rare parse accuracy. Finally, both NOPCFG parsers achieve 2.7% and 1.8% respectively user fix accuracy. These results further confirm that our ECE-Parsers are not very sensitive to the PCFG use in the abstraction phase. However, the NOPCFG ALLPARSES ECE-Parser performs much worse with 55.52% accuracy and 56.42% rare parse accuracy, which highlights the importance of abstracting token sequences with our full algorithm.

SEQ2PARSE can *parse and repair 94.25%* of programs with syntax errors. In addition, it generates *the exact user fix over 20% of the time*.

3.7.3 RQ3: Efficiency

Next we evaluate SEQ2PARSE's efficiency by measuring how many programs it is able to parse. We limit each ECE-Parser to 5 minutes. (In general, the procedure is undecidable, and we conjecture that a longer timeout will diminish the practical usability for developers.) We compare the efficiency of SEQ2PARSE for all the versions of Figure 3.14 using the full test set of 15,000 programs.

Figure 3.15 shows the cumulative distribution function of all SEQ2PARSE approaches'

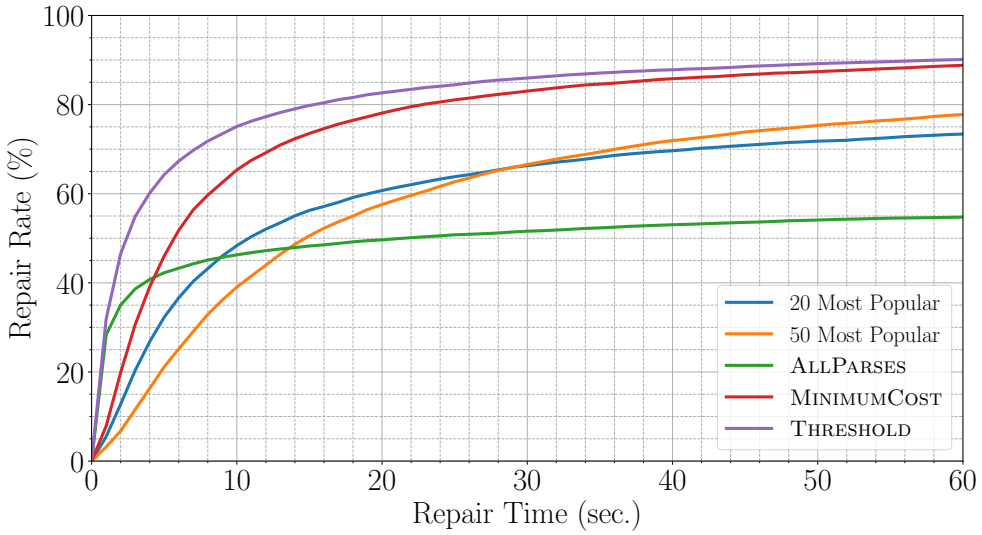


Figure 3.15: The repair rate for all the Abstracted approaches in Figure 3.14.

repair rates over their repair time. We observe that using Threshold predictions with the MinimumCost ECE-Parser is the most efficient and it maintains the highest parse accuracy at all times, with a repair rate of 83.04% within 20 seconds and a median parse time of 2.1 seconds. We also observe that the median parse time for *larger programs* is slightly higher with 7.0 seconds for programs with more than 100 tokens and increases a bit more for more than 500 tokens, with 10.8 seconds. While the ECE-parser uses dynamic programming that may not scale greatly for larger programs, Seq2Parse's scalability is *mostly* proportional to the predicted error rules and the number of syntax errors, and therefore we don't get an exponential explosion in parse time for larger programs.

The MinimumCost with the top 20 error rule predictions is still very efficient with a repair rate of 78.10% within 20 seconds and a median parse time of 5.3 seconds. For larger programs the median parse time is 12.9 seconds for programs with more than 100 tokens and 50.8 seconds for more than 500 tokens. We observe that a fixed-length set of predicted error rules can hinder the ECE-parser, when inaccurate predictions are involved.

We observe that, using a fixed set of the 20 and 50 most popular rules, Seq2Parse

(with the MINIMUMCOST ECE-Parser) repairs 61.41% and 58.61% of the programs respectively within 20 seconds, and has median parse times of 7.0 and 13.6 seconds respectively. The 50 most popular rules admit parsing fewer programs quickly than the 20 most popular.

We also observe that SEQ2PARSE successfully parses around 49.90% of the programs with its ALLPARSES approach in 20 seconds and has a median parse time of 7.1 and 14.2 seconds for all programs and programs with more than 100 tokens respectively. While this approach is much less efficient that the others, it is also able to generate the exact human repair in more than 1 out of 3 cases, representing a valuable quality tradeoff (§ 3.7.2).

> SEQ2PARSE can parse programs with syntax errors for the vast majority of the test set in under 20 seconds with a median parse time of 2.1 seconds.

3.7.4 RQ4: Usefulness

As SEQ2PARSE is intended as an aid for programmers (especially novices) faced with parse errors, we are also interested in subjective human judgments of the quality and helpfulness of our repairs. Around 35% of repairs produced by SEQ2PARSE using its ALLPARSES approach are identical to the historical human repair and thus likely helpful for programmers. However, it may be that SEQ2PARSE's parses (and thus repairs) are still helpful for debugging even when they differ slightly from the human repair (*i.e. non-equivalent* repairs). To investigate this hypothesis, we conduct a human study of the quality and debugging helpfulness of SEQ2PARSE's non-equivalent repairs.

Human Study Setup. We recruited participants from two large public research institutions (UC San Diego and University of Michigan) and through Twitter. The study was online, took around 30 minutes, and participants could enter a drawing for one of two $50 awards. In the study, participants were each asked to rate 15 debugging hints randomly selected from a corpus

of 50 stimuli.[1]

We created the stimuli by selecting 50 buggy programs from our test set for which SEQ2PARSE and the human produced different fixes. Other than ensuring a wide array of difficulty (as assessed by how long the human took to fix the error), programs were selected randomly. Each stimulus consisted of a buggy program, its associated syntax error message, and a potential program fix presented as a *debugging hint*. For each stimulus, we produced two versions: one where the debugging hint was generated by SEQ2PARSE and one where the debugging hint was the historical human fix. Note that, in practice, the historical human fix would *not* be available to a struggling novice in real situations: it represents future or oracular information. Informally, in our comparison, the historical human fixes can be viewed as an upper bound.

Participants rated the quality and helpfulness of each debugging hint using a 1–5 Likert scale. They also indicated if the debugging hint provided helpful information beyond that in the Python error message. Participants were unaware of whether any given hint was generated by a human or SEQ2PARSE, and participants were never shown multiple fixes to the same program. To be included in the analysis, participants had to assess at least four stimuli. Overall, we analyze 527 unique stimuli ratings from $n = 39$ valid participants (246 for human fixes and 281 for SEQ2PARSE).

Overall Results. While humans in our study find that non-equivalent repairs produced by SEQ2PARSE are lower in both quality and debugging helpfulness than those produced manually (2.9/5 helpfulness for tool-produced repairs vs. 3.7/5 for human-produced repairs, $p < 0.001$), humans still often find SEQ2PARSE's fixes helpful for debugging. Participants found that SEQ2PARSE repairs contained helpful debugging information beyond that contained in the Python Error message 48% of the time (134/281). This additional debugging information was helpful in terms of both the content (73% of the time) and location (55% of the time). Additionally,

[1]All human study stimuli are included in our replication package at `https://github.com/gsakkas/seq2parse` and via the human study website `https://dijkstra.eecs.umich.edu/~endremad/APR_HumanEval/`.

SEQ2PARSE fixes are helpful for easy and hard Syntax Errors alike: we found no statistically-significant difference between the helpfulness or quality of SEQ2PARSE's repairs for easy (those repaired by the human in under 40 seconds) or hard parse errors (over 40 seconds). Overall, these results indicate that even when SEQ2PARSE repairs differ from historical human repairs, they can still be helpful for debugging.

Individual Stimuli. Beyond an analysis of SEQ2PARSE's overall quality, we also analyze the helpfulness of each stimulus. Of the 48 programs for which we collected sufficient data to permit statistical comparison, the historical repair was statistically more helpful for debugging than SEQ2PARSE's repair for 33% of stimuli (16/48, $p < 0.05$). However, we found that SEQ2PARSE's repair was actually *more helpful* for debugging than the human's repair for 15% of stimuli (7/48, $p < 0.05$). For the remaining 52% of stimuli, we found no evidence of a statistical difference in the debugging helpfulness of the two repairs.

To better contextualize these results, we provide examples of stimuli with statistically significant differences in debugging helpfulness. In Figure 3.16b, SEQ2PARSE's repair was significantly more helpful than the historical repair: SEQ2PARSE correctly adds parentheses to `print` while the human simply deletes the buggy line, perhaps out of confusion or frustration. Similarly, Figure 3.16a's SEQ2PARSE repair was also better than the human repair. In this case, the user appears to try to implement a function to calculate the greatest common divisor of two integers, but has empty `if` and `elif` statements. To "fix" this bug, the user deletes the `if` and `elif` and modifies the return statement. However, this fix does not correctly calculate the greatest common divisor. SEQ2PARSE, on the other hand, adds a template variable to the `if` and `break` to the `elif`. While this also does not implement greatest common divisor, it is viewed as more helpful than the user repair. This example also demonstrates the beneficial ability of our approach to conduct multi-edit repairs.

Figure 3.16c, on the other hand, shows an example of a more helpful human repair. In

```
# Buggy
def gcdIter(a, b):
    for i in range(1, a+1):
        if a % i == 0:
        elif b % i == 0:
    return i
gcdIter(9, 12)
```

```
# Human
def gcdIter(a, b):
    for i in range(1, a+1):
        return a % i
gcdIter(9, 12)
```

```
# Seq2Parse
def gcdIter(a, b):
    for i in range(1, a+1):
        if a % i == 0: new_var
        elif b % i == 0: break
    return i
gcdIter(9, 12)
```

(a) SEQ2PARSE repair significantly more helpful: 4.3/5 vs 1.0/5, $p = 0.03$

```
aList = [12, 'yz', 'ab'];
aList.reverse();
print "List : ", aList
```

```
aList = [12, 'yz', 'ab']
aList.reverse()
```

```
aList = [12, 'yz', 'ab']
aList.reverse()
print("List : ", aList)
```

(b) SEQ2PARSE repair significantly more helpful: 4.75/5 vs 2.0/5, $p = 0.02$

```
a = int(input(enter a))
print(a***3)
```

```
a = int(input("enter a"))
print(a**3)
```

```
a = int(input(enter)(a))
print(a ** (* 3))
```

(c) Historical human repair significantly more helpful: 1.8/5 vs 4.75/5, $p = 0.01$

Figure 3.16: Three example buggy programs followed by their historical human and SEQ2PARSE repairs. For (a) and (b), SEQ2PARSE's repair was rated more helpful by participants. For (c), the human repair was more helpful.

this case, the human correctly deletes the extraneous * in the power operator while SEQ2PARSE adds parentheses to make a more complex expression, the result of favoring one insertion over one deletion.

35% of SEQ2PARSE's repairs are equivalent to historical repairs. Of the remainder, our human study found 15% to be more useful than historical repairs and 52% to be equally useful. In total, including both equivalent and non-equivalent cases, SEQ2PARSE repairs are at least as useful as historical human-written repairs 78% of the time.

3.8 Related Work

There is a vast literature on automatically repairing or patching programs: we focus on the most closely related work on providing feedback for parse errors.

Error-Correcting Parsers. As we have already demonstrated, error-correcting parses have been proposed for repairing syntax errors and we have extensively described ECE-Parsers [5]. The technique presented by [13] describes another EC-Parser, which is applicable with LR and LL parsing. It uses three phases: first attempts to repair the parse error by symbol insertions, deletions, or substitutions. If that fails, it tries to close one or more open code blocks and if that fails, it removes code surrounding the erroneous symbol. Finally, it uses *deferred parsing* that may be viewed as double parsing, where one main parser moves forward as much as possible, whereas a second parser is k steps behind, so that it can backtrack to a state k steps before efficiently if a phase fails. [124] have shown that the previous approach is not applicable in real-world languages for some specific cases (*e.g.* multiple function definitions) and has suggested an improvement that works with the JAVACC parser generator and a form of *follow-set error recovery*. [20] have suggested an error-correcting version of the popular LR parser. Rather than focusing on error production rules, this method adds *error-repair transitions* along with the regular shift/reduce operations. It employs a simple cost model and heuristics to limit the explosion of the repair search space. Finally, [122] has suggested using *probabilistic parsing* to overcome the drawback of selecting the minimal-edit repair by using a PCFG to select the most *probable* repair parse. However, these approaches are impractical and inefficient for real-world applications, as they can only successfully parse small examples or use tiny grammars. In contrast, SEQ2PARSE relies on pre-trained sequence models to efficiently explore the repair search space for a minimal overhead in real-time parsing.

Sequence Models in Software Engineering. [96] and [128] have suggested using pre-trained auto-regressive transformer models, such as GPT-3 [12], to augment pre-existing program synthesis techniques. They use pre-trained models to acquire semantic power over smaller subproblems that can't be solved with the syntactic power of classic program synthesis. Similar to SEQ2PARSE, their work uses established pre-existing algorithms from the NLP and PL research areas. However, SEQ2PARSE trains its own Transformer-based model to augment an error-correcting parsing algorithm, providing more focused prior knowledge than a pre-trained sequence model, thus making our model highly accurate.

Sequence Models for Parsing. SYNFIX [9] and *sk_p* [92] are two systems that use seq2seq models consisting of Long Short-Term Memory networks (LSTMs). They mostly focus on educational programming tasks in order to learn task-specific patterns for fixing erroneous task solutions. SYNFIX uses a model per task and uses as an input sequence the program prefix until the error locations that the language parser provides. *sk_p* (while it does not solely focus on syntax errors) makes sequence predictions per program line, by considering only the abstracted context lines (previous and next lines). The model is applied to every program line and the predictions with the highest probabilities are selected. SEQ2PARSE manages to parse and repair a large number of programs regardless the task they are trying to solve by encoding the full erroneous programs with a state-of-the-art Transformer model and utilizing an EC-Parser to parse them accordingly, thus achieving a much higher accuracy. Additionally, it uses a real-world dataset of millions of PYTHON programs to learn to effectively parse programs, while SYNFIX and *sk_p* are trained on smaller datasets of correct programs that have errors manually introduced on training, possibly skewing the predictions away from real-world fixes.

DEEPFIX [41] is another seq2seq approach for repairing syntactical errors in C programs. It relies on stacked *gated recurrent units* (GRUs) with attention and applies some simple abstraction over the terminal tokens. The input programs are broken into subsequences for each line and

the model gets as input all the line subsequences with their associated line numbers. DEEPFIX only predicts single line fixes and its predictions are applied iteratively multiple times, if multiple parse errors exist or until the parse error is fixed. DEEPFIX struggles with the same problems as previous work, as it solely relies on the sequence models' capability to learn the full grammar and repair programs with minimal abstraction and prior knowledge over the language.

Lenient parsing [2] presents another sequence model approach. It uses *two seq2seq Transformer models* and trains them with a large corpus of code. One model is trained to repair and create proper nested blocks of code, called BLOCKFIX, and the second one, called FRAGFIX, repairs and parses fragments of code (*e.g.* program statements) within a repaired block. BLOCKFIX tokenizes input program block in a similar manner to our abstracted token sequences, by abstracting identifiers, constants, expressions, etc., and is trained on pairs of valid and manually-corrupted blocks. On the other hand, FRAGFIX repairs on a program-statement level within blocks (mostly focusing on missing semicolons and commas), by using serialized versions of ASTs and error hints manually injected on the ASTs. While this overall approach is mostly automatic, it relies on the manual corruption of a dataset to generate erroneous programs that may not correlate to the errors actual developers make and solely relies on the seq2seq models to learn the underlying language model and make repairs. In contrast, SEQ2PARSE mitigates this problem by learning how programmers fixed programs from a large corpus and by abstracting via partial parses. Additionally, our use of EC-Parsers and the language grammar significantly improves program repairs.

Graph models for parsing. Graph-based Grammar Fix (GGF) [133] suggested using a *Gated Graph Neural Network* encoder for the partial parse trees that can be acquired from a LALR parser and a *GRU* encoder for the parts of the program sequence that are not parsed. This approach aims to better summarize the context of the program in order to train more accurate models. Its models then predict an error location in the program sequence and a token suggestion for the

repair. This single-token repair approach is applied iteratively multiple times until the program correctly parses. While this approach is much more accurate than any previous work, it still lacks the advantages of using a parser with the actual grammar as the final step of the repairing process that SEQ2PARSE takes benefit from and relies again on the model to learn the semantics of the language.

Neural Machine Translation (NMT) for Program Repair. COCONUT [77] proposed a complex architecture that uses a new *context-aware NMT model* that has two separate *Convolutional Neural Network (CNN)* encoders, one for the buggy lines and one for their surrounding lines. It also uses *ensemble learning* to train NMT models of different hyper-parameters to capture different relations between erroneous and correct code. This approach uses a minimal level of abstraction over the input programs, with only a subword-level tokenization to minimize the vocabulary size and make training tractable. CURE [51] suggested a similar *code-aware NMT model* that is pre-trained using unsupervised learning on correct programs. It also uses a programming language GPT [12] model that learns to predict the next token in program sequences and uses beam search to maintain a small set of accurate repairs.

Qualitative Comparison to SEQ2PARSE. SEQ2PARSE performs quite well compared to the aforementioned published state of the art for the particular domain of novice programs. Noting that many of these are on different benchmarks or datasets, permitting only an indirect comparison. we believe that SEQ2PARSE compares favorably in terms of *accuracy* and *efficiency*, since it completely repairs 94.25% of our tests within 2.1 seconds, while generating the exact user fix in more than 1 out 3 of the cases, a metric that most papers ignore.

Specifically, DEEPFIX [41] uses a multi-layer seq2seq model to repair programs that may have up to 5 syntax errors, but initial results, although promising, yield error-free compilation for only *27%* out of the 6971 benchmark programs. *Lenient parsing* [2] leverages a large corpus of code and error seeding to train a transformer-based neural network, resulting in a broadly

applicable approach, but one with potentially lower accuracy in our domain (a top-1 repair accuracy of only *32%* for real student code with up to 3 syntax errors). GGF [133] tries to encode program context in a novel way by using a graph neural network and partial parses, which leads to a higher repair accuracy of *58%* of the syntax errors in a real-world dataset. Lastly, CoCoNuT [77] is a recent state-of-the-art automated repair technique that depends on a different approach of context-aware NMTs and is evaluated on standard software defect benchmarks. While CoCoNuT is able to repair a broader range of defects than syntax errors, it only repairs 509 out of 4456 (11.42%) benchmark defects.

3.9 Conclusion

We have presented *neurosymbolic parse program repair*, a new language-agnostic neurosymbolic approach to automatically repair parse errors. Our approach is to use a dataset of ill-parsed programs and their fixed versions to train a Transformer classifier (neural component) which allows us to accurately predict EC-rules for new programs with syntax errors. In order to make accurate predictions, we abstract the low-level program token sequences using partial parses and probabilistic grammars. A small set of predicted EC-rules is finally used with an ECE-Parser (symbolic component) to parse and repair new ill-parsed programs in a tractable and precise manner. We believe that the *novel* combination of neural and symbolic components helps outperform previous work in terms of repair accuracy and efficiency.

We have implemented our approach in SEQ2PARSE, and demonstrated, using a corpus of 1,100,000 ill-parsed PYTHON programs drawn from two years of data from an online web-based educational compiler, that SEQ2PARSE makes accurate EC-rule predictions 81% of the time when considering the top 20 EC-rules, and that the predicted EC-rules let us parse and repair over 94% of the test set in 2.1 sec median parse time, while generating the user fix in almost 1 out of 3 cases. Finally, we conducted a user study with 39 participants which showed that SEQ2PARSE's edit locations and repairs are useful and helpful, even when they are not equivalent to the user's fix.

3.10 Acknowledgements

Chapter 3, in part, is a reprint of the material as it appears in the Proceedings of the ACM on Programming Languages, Volume 6 (OOPSLA2). Georgios Sakkas, Madeline Endres, Philip J Guo, Westley Weimer, Ranjit Jhala, SPLASH 2022. The dissertation author was the primary investigator and author of this paper.

3.11 Data Availability Statement

All code for extracting ML-appropriate datasets, training the sequence models and repairing and parsing ill-parsed programs with SEQ2PARSE is publicly available at `https://github.com/gsakkas/seq2parse`. Additionally, a simplified online demonstration is available at `http://seq2parse.goto.ucsd.edu/index.html`. Finally, the code is also packaged in the available artifact [105].

Chapter 4

Neurosymbolic Modular Refinement Type Inference

4.1 Introduction

Refinement types are a type-based generalization of Floyd-Hoare logics, where the programmer can specify correctness requirements by decorating classical types (*e.g.*`Int`) with *logical predicates* (*e.g.*`0 <= v`) that provide additional constraints on the values that can inhabit the type, thereby providing a *modular* and *expressive* means of statically enforcing a wide variety of correctness, safety, and security properties of software. Refinement types have been developed for various languages, from the ML family [101, 120, 126, 135], to C [93, 100, 107], Ruby [54], Rust [35, 65], TypeScript [127], Scala [43], Solidity [121], Racket [55]. A recent paper presented a user study of 30 developers using refinement types for Java [36] that concluded that "LiquidJava helped users detect and fix more bugs, and that Liquid (Refinement) Types are easy to interpret and learn with few resources."

Sadly, as with other expressive and modular program verification tools like ESCJava [30] or Dafny [66], the wider usage of refinement types is hindered by the fact that to effectively use refinement types across their codebase, developers must laboriously *annotate* all the functions in their code with potentially complex type specifications that specify the behavior of that function to the rest of the code. The expressiveness of refinement contracts means that (unlike in classical type systems where often a type can be uniquely determined from the code) there is an infinite space of possible specifications for each function, which makes it tricky for the developer to determine the right one. The problem is exacerbated by modularity which means that the refinement type or contract specified for a function `f` may be "correct" for `f` in isolation, but may not suffice to verify `f`'s *clients*, and so the developer has to go back and forth changing the annotations of functions to get the entire codebase to verify.

In this chapter, we present LHC[1], a neurosymbolic agent that uses *large language models* (LLMs) to automatically generate refinement type annotations for the functions in an entire codebase, using the refinement type checker LIQUIDHASKELL as an oracle to verify the

[1] Stands for *Liquid Haskell Copilot* or LHCOPILOT

correctness of the generated specifications. We develop our approach via three contributions.

1. Agent. Our main contribution is an agent that systematically traverses the codebase's call-graph to generate each function's refinement type annotation. If we think of the refinement type annotation as the analog of a procedure *summary*, then we can think of our agent as a neurosymbolic program analysis, that combines a "bottom-up" analysis which uses neural LLMs to generate refinement type annotations (summaries) for functions, with a "top-down" analysis that kicks in when the LLM fails to generate correct types, that instead uses a symbolic predicate abstraction technique to generate refinement types from predicate *templates* obtained from the failed LLM predictions. Thus, even where the LLM fails to generate the correct type, its predictions can be used to generate an abstract domain that allows the symbolic analysis to succeed.

2. Dataset. Our second contribution is a dataset comprising three Haskell packages: a suite of programs which are part of a tutorial on refinement types, a Haskell implementation of the Salsa20 cipher, and a widely used library that implements Byte-Strings with low-level pointer operations. The dataset includes a diverse set of functions, totalling about 5KLoC annotated with refinement types that enforce a variety of correctness properties ranging from data structure invariants to low-level memory safety. This dataset was curated to deliberately *exclude* code present in the popular open-source code LLM training dataset *The Stack* [70, 75], to ensure that successful type generation is not simply due to memorization.

3. Evaluation. Our final contribution is an evaluation of LHC on our dataset, using a variety of pre-trained LLMs, including StarCoder and CodeLlama which were *not* trained on the code in our dataset. We demonstrate that by fine-tuning these LLMs on a small set of about 9,000 LIQUIDHASKELL programs, we can greatly improve the agent. We show how by combining the bottom-up generation of the neural models with top-down symbolic inference using qualifiers

from the LLMs predictions, LHC can automatically generate refinement types for up to 94% of the functions across entire libraries. Furthermore, the entire generation process can be completed in just a few hours, a significant improvement over the several days or weeks of human effort that originally went into annotating the packages, thereby indicating that LLMs can drastically shrink the human effort needed to use formal verification.

4.2 Background

We start with some preliminaries showing how refinement types can be used to specify and verify properties of programs § 4.2.1, and how LLMs can be used to automatically generate the type annotations required for verification § 4.2.2.

4.2.1 Refinement Type Checking with LIQUIDHASKELL

Specification. Refinement type checkers like LIQUIDHASKELL let the programmer specify correctness requirements decorating classical types with *logical predicates* — typically drawn from an SMT-decidable theory — which provide additional constraints on the values that can inhabit the type. A refined *base* type of the form `{v:T|p(v)}` defines the set of values `v` of type `T` such that additionally, the constraint `p(v)` is true of the value `v`. For example, the type `{v:Int | 0 <= v}` specifies the set of *non-negative* integer values. A refined *function* type of the form `x:{In|pre(x)} -> {v:Out|post(v, x)}` can specify pre- and post-conditions for the underlying functions via constraints on the `In`put and `Out`put types. For example, the type `x:{Int|0 <= x} -> {v:Int|v >= x}` specifies a function that requires non-negative inputs, and ensures that the returned value is at least as large as the input `x`.

Refinement type checkers also allow the programmer to specify properties of data using *measure* functions [53, 126], which are pure and total functions that map data types (such as lists, trees, *etc.*) to SMT-decidable values (such as integers, booleans, sets *etc.*). For example, the measure `notEmp` (Figure 4.1) defines a boolean predicate on lists that is true if the list is non-empty,

```
measure notEmp :: [a] -> Bool
  notEmp []     = False
  notEmp (_:_) = True
```

Figure 4.1: An example measure for refinement types.

and we can use it to specify that a particular function should only be called with non-empty lists:

```
{-@ head :: {v:[a] | notEmp v} -> a @-}
head (x:_) = x
head []    = error "empty list" -- runtime crash
```

Figure 4.2: Using measures in LIQUIDHASKELL.

Verification. Refinement type checkers like LIQUIDHASKELL verify the specifications by generating *verification conditions* (VCs) — logical formulas whose validity, determined by an SMT solver [86], ensures that the program is type-safe. For example, consider the code for the `head` function shown above, and assume that `error` — which aborts the program with a run-time panic — is a library function that is given the type

```
{-@ error :: {v:String | False} -> a @-}
```

That is, the precondition of `error` says it can only be called with `String` messages such that the predicate `False` holds. Since there are no such `String`s, the program will only verify if at compile-time, the refinement type checker can prove that `error` is never actually called. LIQUIDHASKELL verifies the code for `head` by generating the VC:

$$\forall v.\ notEmp(v) \ \Rightarrow\ \neg notEmp(v) \ \Rightarrow\ False$$

The first antecedent comes from the precondition that the input list is a non-empty list, the second antecedent comes from the fact that in the second case (where we call `error`) the input list is matched against `[]` whose measure is `False`, and the consequent *False* arises from the pre-condition of `error`. The SMT solver proves the above VC valid to verify that `head` will never crash on non-empty lists.

Modularity and Annotations. Refinement type checking is *modular* in that when we check a client (*e.g.*`head`) that calls a function (*e.g.*`error`) the only information known about the callee (`error`) is its *type signature*. This means that to analyze an entire package or module, the programmer must *annotate* all the functions of the module with (refinement) type signatures.

```haskell
type NonZero = {v:Int | v /= 0}

type NEList a = {v:[a] | notEmp v}

{-@ divide :: Int -> NonZero -> Int @-}
divide :: Int -> Int -> Int
divide _ 0 = error "divide-by-zero"
divide x n = x `div` n

{-@ size :: xs:_ -> {v:Nat | notEmp xs => v>0} @-}
size :: [a] -> Int
size []     = 0
size (_:xs) = 1 + size xs

{-@ average :: NEList Int -> Int @-}
average xs = divide total elems
  where
    total = sum xs
    elems  = size xs
```

Figure 4.3: Haskell module with multiple dependent functions.

For example, consider the code in Figure 4.3 which shows a small Haskell module that implements a function that computes the `average` of a list of integers by computing the `sum` of the integers and then invoking `divide` with the `size` of the list. The `divide` function panics with `error` when the divisor is `0`, and otherwise calls the mathematical `div` operator. The `size` function recursively traverses the input list to count the number of elements in it.

To verify this module, the programmer must annotate each of the three functions with a type signature. First, for `divide` they must specify that the second argument is `NonZero` — so that LIQUIDHASKELL can verify that `error` will not be called at run-time. Second, for `size` they must specify that the function returns a strictly positive result *if* the input is non-empty. Finally, for `average` they must specify that the input list is itself non-empty, which lets LIQUIDHASKELL determine — using the annotation for `size` — that `total` is strictly positive, and hence that the call to `divide` is also safe.

Symbolic Type Inference with Qualifiers. Refinement type checkers require type annotations in many places, *e.g.* for (recursive) functions, polymorphic type instantiation and so on. These

can be viewed as type-based generalizations of the classic problem of having to specify pre- and post-conditions and loop- annotation invariants in Floyd-Hoare style verifiers like ESCJava or Dafny [30, 67]. As with loop invariants, refinement type inference is undecidable in general, but the type-based setting allows LIQUIDHASKELL to use a form of abstract interpretation called *predicate abstraction* [38, 101]. Here, the programmer provides a set of *qualifiers* — predicate fragments or templates — that LIQUIDHASKELL can then automatically conjoin to infer refinement types. In our running example in Figure 4.3, we could provide templates:

```
qualif Qual0(v: a): v > 0
qualif Qual1(v: a, xs: b): notEmp xs => v > 0
```

and then simply annotate `average` and `size` with the *wildcard* types: `average :: _ -> _ -> _` and `size :: _ -> _` after which LIQUIDHASKELL will be able to *automatically* infer the refinement type annotations needed to verify the module [101]. Additionally, LIQUIDHASKELL can automatically *extract* qualifiers from annotated type specifications. For example if the programmer wrote a specification `x:Int -> {v:[a] | x < len v} -> a` then LIQUIDHASKELL would automatically extract a qualifier `Qual2(v: a, x: b): x < len v` and then use it for subsequent type inference.

4.2.2 Neural Type Inference with LLMs

Even with symbolic refinement type inference, there is a substantial burden on the programmer as they must be able to either write down the types for all functions *or* divine a set of suitable qualifiers from which types can be inferred. We aim to reduce this burden by using large language models (LLMs), specifically code-specific models, trained on large tracts of source code, to assist the programmer in generating the necessary type annotations needed to verify entire modules.

Constructing prompts. LHC infers refinement types for entire modules by repeatedly crafting prompts that can guide the LLM to generate accurate and relevant refinement types for each function. In the case of refinement types, LHC uses LLMs that have been pre-trained on infilling *missing* (*masked*) parts of programs, and generate prompts that provide context about the Haskell code while indicating where the refinement type is missing. For Figure 4.3, LHC builds the following LLM prompt to generate a refinement type for `divide`

```
-- Fill in the masked refinement type
{-@ type NonZero = {v:Int | v /= 0} @-}

{-@ measure notEmp :: [a] -> Bool
      notEmp []     = False
      notEmp (_:_)  = True      @-}

{-@ type NEList a = {v:[a] | notEmp v} @-}

{-@ divide :: <mask> @-}
divide :: Int -> Int -> Int
divide _ 0 = error "divide-by-zero"
divide x n = x `div` n
```

Figure 4.4: Refinement type prompt for querying LLMs.

Few-shot prompting with function dependencies. *Few-shot prompting* is a technique used in the context of LLMs where the model is provided with *some examples* (typically between one and a few dozen) to illustrate the task it needs to perform. This approach helps the LLM understand the pattern and context of the task, improving its performance on similar tasks. In contrast, *zero-shot prompting* provides no specific examples to the LLM, relying entirely on the model's pre-trained knowledge to perform the task based on a descriptive prompt. Few-shot prompting generally yields better results than zero-shot prompting as it gives the LLM concrete examples to learn from, thereby reducing ambiguity and increasing accuracy.

In the context of refinement types, few-shot prompting can be particularly useful. For instance, when asking a Code LLM to generate refinement type annotations for Haskell code, a few-shot prompt would include several examples of functions annotated with appropriate

refinement types. This helps the LLM learn the patterns and constraints associated with these types. By seeing specific examples, the LLM can more accurately predict and infill the missing refinement type annotations in new, un-annotated code.

Specifically, LHC adds all the functions and their types in the prompt, that the target function depends on. For our example in Figure 4.3, when we query a LLM to generate types for `average`, the functions `divide` and `size` with their type signatures will also be added to the prompt as extra examples to help the LLM generate the correct type for `average`.

4.3 Overview

Let's look at an overview of how LHC systematically infers the refinement type annotations needed to automatically verify a given Haskell codebase, by traversing the call-graph of the codebase, in a bottom-up order, prompting the LLM to *generate* new type predictions that can be locally *verified* by LIQUIDHASKELL, and then *back-jumping* to a dependency when the predictions fail.

4.3.1 Initialization

The input for LHC is an *un-annotated* program, *i.e.* a program where some functions are not yet annotated with a refinement type. Based on the running example the initial program is the code from Figure 4.3 where we removed the orange specifications for the *target* functions `divide`, `size` and `average`. Helper type aliases, such as `NEList` and `NonZero`, are standard in LIQUIDHASKELL and very commonly used by more complicated refinement types in order to simplify them. Therefore, here, they are left untouched for more context when prompting the LLM to get more accurate predictions.

LHC State. During the entire type inference process, LHC maintains a global state S that captures the current state S_f of each function f which corresponds to a triple $(\mathsf{fuel}, \mathsf{type}, \mathsf{predicts})$. The fuel represents the maximum number of times that LHC will attempt to infer a type for f before giving up and asking the programmer to provide a type. The type represents the current type that the function f has been assigned and against which the implementation of f has been verified, or $\bot$ if no such type has been assigned. The $\mathsf{predicts}$ queue stores all the predicted types from a LLM that are yet *to be* tried. For our example, the initial global state S maps each of `divide`, `size` and `average` to a triple where fuel is 10, type is $\bot$ and $\mathsf{predicts}$ is empty.

Dependencies. Next, we identify the potential *dependencies* between the different functions, because the order that we generate types and verify them matters for the correctness of our approach. We build the program's *call-graph* and get all function dependencies, where deps_f is the set of functions that f calls. For our example, `average` calls `divide` and `size`, each of which have no dependencies. Thus, the call-graph has a depth of 2: the dependencies of `divide` and `size` are empty, and of `average` is `[divide, size]`.

Worklist. Finally, LHC maintains a worklist wkl with all the functions that are yet to be explored and verified by our approach. We initialize the wkl with all the *root functions*, *i.e.* functions that have *no dependencies*. These roots will be the starting points at which LHC will infer types. For our example we initialize wkl with `[divide, size]`.

4.3.2 Building the LLM prompt

LHC starts by popping `divide` from the working list wkl. Since we have no type predictions so far, we need to call the LLM to generate some. For that we make the following prompt (as described in § 4.2.2):

```
measure notEmp :: [a] -> Bool
  notEmp []     = False
  notEmp (_:_) = True

type NonZero  = {v:Int | v /= 0}

type NEList a = {v:[a] | notEmp v}

{-@ divide :: <mask> @-}
divide :: Int -> Int -> Int
divide _ 0 = error "divide-by-zero"
divide x n = x `div` n
```

Figure 4.5: LLM prompt for `divide`.

The prompt contains the program up to the function that we are generating refinement types for, where we add a "dummy" refinement type **`<mask>`** that the LLM needs to fill in.

DEPENDENCIES Optimization. As described in § 4.2.2, this optimization is a few-shot prompt technique where *all dependencies* that the target function calls, are added in the prompt. While `divide` and `size` don't have any dependencies and their prompts remain as described above, the prompt for `average` would include both of these functions when the DEPENDENCIES optimization is enabled, since `averag` calls both of them.

4.3.3 Generating type predictions

Given the above prompt for `divide`, the LLM will generate the following types for example where the 3rd one is the correct one. [2]

```
divide :: NonZero -> Pos -> Pos -- rejected
divide :: NonZero -> Nat -> Pos -- rejected
divide :: Int -> NonZero -> Int -- correct
divide :: {v:Int | v /= 0} -> Nat -> Nat
divide :: NonZero -> Int -> {v:Int | v > 0}
```

Figure 4.6: Refinement type predictions for `divide`.

4.3.4 Verifying types

The next step is to identify a type from the prediction queue above, that is *locally correct*, *i.e.* against which LIQUIDHASKELL will verify the *given* function (here, `divide`). We iteratively check `divide` against each of the candidate types, decreasing `divide`'s fuel each time, until we reach a locally correct type that is verified by LIQUIDHASKELL. After this step, the global state is updated so that for `divide`, we have two remaining type predictions in the predicts, the fuel has been decreased by 3 since we tested that many type predictions and the current type is updated to

[2]We consider here the top 5 predictions but in reality can generate up to 50 types in total

`Int -> NonZero -> Int.`

function	fuel	type/predicts
`divide`	7	**`Int -> NonZero -> Int`** /
		`NonZero -> Nat -> Nat`
		`NonZero -> Int -> Nat`
`size`	10	- /-
`average`	10	- /-

QUALIFIERS Optimization. As described in § 4.2.1, the programmer can provide a set of *qualifiers* and a *wildcard type* for the target function in order to enable LIQUIDHASKELL to automatically infer the appropriate refinement type. When we enable the QUALIFIERS optimization, we add the wildcard type for the target function at the end of the list of predicted types, *i.e.* `divide :: _ -> _ -> _` for our example, in order for this type to be tested as a last resort if all other types fail verification. Additionally, we *extract automatically* all possible predicates from the predicted refinement types to add the corresponding qualifiers in the program. For example, for `divide` we would extract `Qual1(v: a): v /= 0` and `Qual2(v: a): v > 0` from the last two predictions from § 4.3.3

4.3.5 Updating the working list

After we locally verify `divide` with one of the predicted types, we look up all functions f that call `divide`, *i.e.* the functions f such that deps_f contains `divide`, and we add them to the working list wkl if *all their dependencies* are resolved, *i.e.* all functions in deps_f have also been locally verified against their current type. In this case `average` calls `divide` but `average` also depends on `size`, which we have yet to explore. Therefore, *no new functions* are added to the working list wkl, which now, only contains `size`.

As in § 4.3.2 and § 4.3.3, we perform the same steps for `size` to generate type predictions:

```
size :: xs:[a] -> {v:Int | v > 0}
size :: xs:[a] -> {v:Int | v >= 0}
size :: xs:[a] -> {v:Int | v = size xs}
size :: xs:[a] -> {v:Nat | notEmp xs => v>0}
size :: xs:[a] -> {v:Nat | v = len xs}
```

Figure 4.7: Refinement type predictions for `size`.

The first predicted type `xs:[a] -> {v:Int | v > 0}` is rejected by LIQUIDHASKELL as it is not locally correct as `size` can return `0` on an empty list. The second type `xs:[a] -> {v:Int | v >= 0}`, however, is *locally* correct – the output of `size` is always non-negative. (Note, however, it is not the type that is needed to verify the whole module, in particular, that is needed to verify `average` which we still have not explored. We show next how this issue is resolved in our approach.) At this point, the global state is updated to

function	**fuel**	**type/predicts**
`divide`	7	**`Int -> NonZero -> Int`** /
		`NonZero -> Nat -> Nat`
		`NonZero -> Int -> Nat`
`size`	8	**`xs:[a] -> v:Int | v >= 0`** /
		`xs:[a] -> v:Int | v = size xs`
		`xs:[a] -> v:Nat | notEmp xs => v > 0`
		`xs:[a] -> v:Nat | v = len xs`
`average`	10	- /-

which corresponds to the *partially annotated* program:

At this stage, since all of `average`'s dependencies are also locally verified, we can add it to the working list wkl, so that we can generate and try types for it as well.

```
{-@ divide :: Int -> NonZero -> Int @-}
divide :: Int -> Int -> Int
divide _ 0 = error "divide-by-zero"
divide x n = x `div` n

{-@ size :: xs:[a] -> {v:Int | v >= 0} @-}
size :: [a] -> Int
size []     = 0
size (_:xs) = 1 + size xs

average xs = divide total elems
  where
    total  = sum xs
    elems  = size xs
```

Figure 4.8: Partially annotated program, where `average` is yet to be annotated.

4.3.6 Back-jumping to a dependency when predictions fail

Now, LHC repeats the same generate-and-check procedure for `average` as it did for `divide` and `size`. However, this time, the 5 LLM-predicted types are invalid, in that *none* of them can be locally verified against the implementation of `average`. This could mean one of two things:

(1) One of `average`'s function dependencies has a type that is either too *weak* (*i.e.* does not specify what the client requires in its post-condition), or too *strong* (*i.e.* has a pre-condition that rejects the actual inputs provided by the client).

(2) All type predictions for `average` were wrong.

In this case, condition (1) holds as `size` had indeed a problematic type as we hinted earlier, which made `average` impossible to verify. However, at this stage, LHC cannot distinguish between (1) or (2) — *i.e.* we do not *know* which condition actually holds and therefore we need to make a decision based on the global state.

Since all of the predictions in the predicts queue for `average` are exhausted, and we have available fuel for it, we choose to *back-jump* to one of `average`'s function dependencies, in particular, the one that has the *highest remaining fuel*, *i.e.* the one that has been the least tested

and thus has the most candidate type predictions left. The dependencies that we can potentially back-jump to are not just the immediate parent functions in the call graph, but any possible *ancestor*, which in this case includes `divide` and `size`.

In this case, we would indeed jump back to the problematic `size` that has the highest fuel of 8. In this process we clear *all current types* for functions that directly or transitively call `size` and we end up with the following state where `average`'s fuel has now fallen to 5, as we made 5 unsuccessful local verification attempts for it and additionally `average` and `size` have no assigned type.

function	**fuel**	**type/predicts**
`divide`	7	**`Int -> NonZero -> Int`** /
		`NonZero -> Nat -> Nat`
		`NonZero -> Int -> Nat`
`size`	8	**`-`** /
		`xs:[a] -> v:Int | v = size xs`
		`xs:[a] -> v:Nat | notEmp xs => v > 0`
		`xs:[a] -> v:Nat | v = len xs`
`average`	5	`-` /`-`

We now repeat the process of trying type predictions from the predicts for `size` until we get another locally correct type. Of the three remaining predictions predicts the first of these is

rejected by LIQUIDHASKELL's local verification, but the second is accepted yielding the state:

function	fuel	type/predicts
divide	7	**Int -> NonZero -> Int** /
		NonZero -> Nat -> Nat
		NonZero -> Int -> Nat
size	6	**xs:[a] -> v:Nat | notEmp xs => v > 0** /
		xs:[a] -> v:Nat | v = len xs
average	5	- /-

At this point, we again add `average` to the working list wkl, and proceed to generate fresh candidates, and to locally verify them. In this second time, the LLM generates the candidates: and LIQUIDHASKELL rejects the first type to locally verify the second candidate `NEList Int -> Int`, thereby completing the verification of the whole program.

```
average :: xs:[Int] -> {v:Int | size xs > 0}
average :: NEList Int -> Int
average :: NEList a -> Int
average :: xs:NEList Int -> Int
average :: {v:[Int] | notEmp v} -> Int
```

Figure 4.9: Refinement type predictions for `average`.

4.3.7 Asking the user for a type

Suppose that instead, the LLM generated a slate of incorrect types for `average` such that while trying out the generated candidates, the fuel for `average` falls to 0. In this case, we are potentially in condition (2), where all the LLM predicted types are wrong (*i.e.* fail to locally verify). In this scenario, LHC falls back to ask the user to provide a type for `average`.

If the verification fails even with the user-provided type, then we back-jump again to one of the dependencies and repeat the whole process until the function is verified *with* the

user-provided type. That is, we presume that the user-provided type is correct, and we get the LLM to generate types for the *other* functions, so that the whole program verifies. Of course, our goal is to *minimize* the number of times we have to resort to asking the user for a type signature. In our example, assuming the user provides the correct type `NEList Int -> Int` the verification of the whole program succeeds and we return the fully annotated program (shown in Figure 4.3) to the user.

Algorithm 6 LHC's algorithm

Input: Code Repository R
Output: Verified Code Repository R'

```
1:  procedure VERIFYCODEREPO(R)
2:      S ← [f ↦ {fuel = N, type = ⊥, predicts = ∅}]
3:      deps ← BUILDCALLGRAPH(R)
4:      wkl ← {f ∈ R | deps_f = ∅}
5:      while wkl ≠ ∅ do
6:          f ← POPTOP(wkl)
7:          ps ← GETPREDICTIONS(S, f)
8:          t ← TRYPREDICTIONS(S, f, ps)
9:          if t = ⊥ then
10:             wkl ← wkl ∪ BACKJUMP(S, f)
11:         else
12:             S_f.type ← t
13:             wkl ← wkl ∪ SOLVEDCALLERS(S, f, deps)
14:     return R(S)
```

4.4 Algorithm

We describe here the full algorithm of the LHC agent: an interactive approach to verifying a *code repository R*, comprising a set of functions, by automatically annotating all the functions in R with refinement types against which the entire repository verifies.

Algorithm 6 presents LHC's high-level iterative algorithm.

Global State. We define as S the *global state*, that maps each function f to its current state S_f which is a triple of the form $(\mathsf{fuel},\ \mathsf{type},\ \mathsf{predicts})$. The first element, fuel, is an integer representing the upper bound on the number of remaining verification attempts for f. The second element, type, is the current type that has been assigned to and locally verified for f or $\bot$ denoting that no type has been assigned. The third element, predicts, is a *priority queue* of the *predicted types* for f against which f has not yet been locally verified. The *global state* S is initialized such that for each f in the repository, S_f for each function f has a maximum the S_f.fuel is some maximum N, the S_f.type is $\bot$, and S_f.predicts is empty.

Call Graph. BUILDCALLGRAPH generates the repository's *call-graph* returning returns all function dependencies deps_f which is the set of functions that f calls, which we also call the *immediate dependencies* of f. For example, the program in Figure 4.3 has the following dependencies $\mathsf{deps}_{\texttt{divide}} = \emptyset$, $\mathsf{deps}_{\texttt{size}} = \emptyset$, $\mathsf{deps}_{\texttt{average}} = [\texttt{divide}, \texttt{size}]$.

Worklist. Next, we initialize a *worklist* wkl of functions that our procedure is going to operate on. The worklist wkl is initialized with all functions $f \in R$, such that the function f has no dependencies. These are the *root functions* of the repository from which LHC starts generating and checking types.

Main Loop. The main body of the algorithm is on lines 5 to 13, which iterates till all functions are assigned types and wkl is empty. In each iteration we pop the top function f from the wkl stack. Then, we get new or existing *type predictions* ps (§ 4.4.1) for the function f. We try to *locally verify* f with a type from the predictions ps (§ 4.4.2) until we successfully find a type against which f locally verifies in the module with the current state S, *i.e.* with the currently assigned types for the transitive dependencies of f. If none of the predictions ps verified the function f, thus $\mathsf{t} = \bot$, we *back-jump* to f's least tested dependency (§ 4.4.3) to continue the iterations from there. If instead, a type t that locally verified the function f is found, then we update the state S_f.type with the new type t, and add in wkl the functions in SOLVEDCALLERS(S, f, deps). These are all the functions $g \in R$ such that (a) g calls f (*i.e.* $f \in \mathsf{deps}_g$), and (b) all the dependencies of g have an assigned type, (*i.e.* $\forall h \in \mathsf{deps}_g.\ \mathsf{S}_h.\mathsf{type} \neq \bot$). We then continue to the next iteration, ensuring all functions in wkl have been locally verified.

4.4.1 Generating type predictions

Algorithm 7 describes the procedure of generating *new type predictions* for a function f given a global state S. Initially, we check if we have any remaining type predictions in the

Algorithm 7 GETPREDICTIONS's algorithm

Input: State S, Function f
Output: LLM type predictions ps

```
1: procedure GETPREDICTIONS(S, f)
2:     if S_f.predicts = ∅ then
3:         R ← PROGRAM(S)
4:         prompt ← MAKEPROMPT(R, f)
5:         S_f.predicts ← LLMGuess(prompt)
6:     ps ← S_f.predicts
7:     return ps
```

Algorithm 8 TRYPREDICTIONS's algorithm

Input: State S, Function f, Type Predictions ps
Output: Successful type t or Failure $\perp$

```
1: procedure TRYPREDICTIONS(S, f, ps)
2:     while ps ≠ ∅ and S_f.fuel > 0 do
3:         (t, ps) ← GETNEXT(ps)
4:         S_f.fuel ← S_f.fuel − 1
5:         if LHVerify(S, f, t) then
6:             return t
7:     if S_f.fuel = 0 then
8:         return UserHint(f)
9:     return ⊥
```

function's queue S_f.predicts and if there are, no new types are generated and the remaining queue is just returned. Otherwise, we retrieve the relevant functions from the codebase, given the current state S, replacing any types that have already been locally verified (*i.e.* already assigned to the type field in S) and make a prompt for function f as discussed in § 4.2.2. This prompt is sent to the LLM to generate new type predictions, which are then added to f's prediction queue S_f.predicts.

4.4.2 Trying type predictions

Algorithm 8 presents the process of trying new type predictions ps for a function f given a global state S, to find the first prediction against which f locally verifies. In each iteration, we first check the *remaining* fuel for the current function f. If it has reached 0 then we ask the user

Algorithm 9 BACKJUMP's algorithm

Input: State S, Function f
Output: Transitive dependency f' with max remaining fuel

```
1:  procedure BACKJUMP(S, f)
2:      allDeps ← ∅, wkl ← {f}
3:      while wkl ≠ ∅ do
4:          f' ← POPTOP(wkl)
5:          allDeps ← allDeps ∪ {f'}
6:          wkl ← wkl ∪ {g | g ∈ deps_f', g ∉ allDeps}
7:      f' ← ⊥, maxfuel ← 0
8:      for all g ∈ allDeps − {f} do
9:          if maxfuel < S_g.fuel then
10:             f' ← g, maxfuel ← S_g.fuel
11:     S ← RESETDEPENDENCIES(S, f')
12:     return f'
```

to provide a type, as we discussed in (§ 4.3.7). If there is more fuel left, then we decrement f's fuel S_f.fuel by one. Then, we get the next prediction t from the queue ps, and assign the type to f and query LIQUIDHASKELL to try to locally verify the program using the other types already assigned in S. If the verification process is successful, we return the locally verified type t, and otherwise we continue to the next iteration. If we have tried all the predictions in the queue ps and none of them locally verified the function f, then we return $\bot$ signalling that none of the candidate predictions were successful.

4.4.3 Back-jumping to the least tested dependency

Algorithm 9 outlines the BACKJUMP procedure, which identifies and returns the transitive dependency of a function f that has the maximum remaining fuel. The algorithm begins by finding all transitive dependencies of f. This is achieved using a worklist wkl, initialized with f. The algorithm iteratively processes each function in the worklist by popping the top function f', adding it to the set of all dependencies allDeps, and then including all its immediate dependencies $\mathsf{deps}_{f'}$ that are not already in allDeps back into the worklist wkl. This loop continues until the worklist is empty, ensuring that all transitive dependencies of f are collected.

Once all transitive dependencies are identified, the next step is to determine the dependency with the maximum remaining fuel. The algorithm initializes f' to $\bot$ and $\mathsf{maxfuel}$ to 0. It then iterates over each function g in $\mathsf{allDeps}$ excluding f. If the remaining fuel $\mathsf{S}_g.\mathsf{fuel}$ for a function g is greater than $\mathsf{maxfuel}$, it updates f' to g and $\mathsf{maxfuel}$ to $\mathsf{S}_g.\mathsf{fuel}$. This ensures that the function with the maximum remaining fuel among the transitive dependencies of f is selected.

After identifying the function f' with the maximum remaining fuel, the global state S is reset for this function and its dependencies using the RESETDEPENDENCIES procedure, thereby restarting the verification process for f' from scratch. For each function f that calls f', RESETDEPENDENCIES will set $\mathsf{S}_f.\mathsf{type}$ to $\bot$ and recursively will be applied to each caller f to reset all functions that indirectly depend on f'. This allows the type verification process to strategically back-jump to f' and restart with updated predictions and fuel. The selected function f' is then finally returned.

4.5 Evaluation

Next, we describe our implementation of LHC (LHCOPILOT), and an evaluation that addresses three research questions:

RQ1: How *accurate* are LLMs at generating *single* refinement types? (§ 4.5.1)

RQ2: How *precisely* can LHC verify *whole* codebases? (§ 4.5.2)

RQ3: How *efficiently* can LHC verify a given codebase? (§ 4.5.3)

Large Language Models. For our evaluation, we focus on *Code LLMs*, *i.e.* LLMs that have been pre-trained specifically for generating code. Such models don't usually require special system or instruction prompts to effectively generate the target code, unlike CHATGPT or GPT-4O. Our emphasis is also on *smaller and public* LLMs for two important reasons. First, they can be more *robust* in generating types for the hundreds of times it is necessary to verify the given codebase (and likely just as effective [15]). Second, and more importantly, to guard against the possibility of data-leakage (memorization), we wish to ensure that our benchmarks are *not* in the training set for the models. Therefore, we use the following LLMs; STARCODER-3B [70], CODELLAMA-7B [102] and STARCODER2-15B [75] with 3, 7 and 15 billion trainable parameters respectively. All the models are open-source and have been pre-trained on public datasets of code [70, 75]. In each case, the training data does not include the benchmarks considered here. We run all experiments with STARCODER-3B and CODELLAMA-7B on an NVIDIA GeForce RTX 3080 Ti with 12 GB VRAM and an NVIDIA A100 PCIe with 80 GB VRAM for STARCODER2-15B.

Fine-tuning datasets. We fine-tune our models using *The Stack v2* [75], a public dataset of open-souce code with permissive licenses, that includes a vast number of programming languages. While the dataset consists of less than 0.2% of Haskell programs, those programs amount to

over 1 million. Of those programs, only 3350 used LIQUIDHASKELL and refinement types. From these programs, we extract a dataset of 8903 *unique refinement types*, and we use them to fine-tune the LLMs. Specifically, we use QLORA [23], an efficient fine-tuning approach that reduces memory usage enough to fine-tune much larger LLMs on smaller GPUs while preserving full 16-bit fine-tuning task performance. QLORA backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters (LORA [48]). We fine-tune STARCODER-3B and CODELLAMA-7B for 20 epochs, while the larger STARCODER2-15B for 10 epochs due to limited time and resources and its excessive cost.

Benchmarks. We explore different benchmarks in order to answer our research questions. First, for single refinement type prediction (RQ1), we evaluate the different LLMs on a new benchmark based on the publicly available online LIQUIDHASKELL tutorial [98]. We extract *68 refinement types* from the LHTUTORIAL, corresponding to exercises with hidden solutions, into separate programs along with their relevant context, such as our running example in Figure 4.3. For each refinement type in this benchmark, we build a LLM prompt that includes the implemented Haskell function with its type signature and any surrounding context from the corresponding exercise. For example, any relevant functions, comments or possible input-output test cases that can help verify the correctness of the target function. Additionally, functions called by the target function that are already annotated with refinement types were also provided when available.

For the rest of our evaluation (RQ2 and RQ3), we use two public Haskell libraries, HSALSA20 and BYTESTRING. HSALSA20 is a Haskell implementation of the Salsa20 cipher with refinement types used to track the sizes of various ciphers and keys. BYTESTRING is a Haskell library for representing and efficiently operating on byte-strings via pointer operations that is widely used across the Haskell ecosystem; we use refinement types to statically track pointer arithmetic and ensure low-level memory safety. HSALSA20 was annotated with refinement types by its developer, including *96* such annotated functions, while we annotated 45 BYTESTRING

functions with refinement types, in order to establish a ground truth type for these benchmarks. (While there are several other LIQUIDHASKELL repositories available online, they are in the training data for the LLMs we consider, and hence, are excluded from our evaluation.)

Emulating User Interaction. We selected benchmarks that are already fully annotated with refinement types for each function, so that we could emulate the user interaction — UserHint in § 4.4.2 — in our experiments using these ground truth types. That is, when all the predicted types fail for a given target function, we use the ground truth type used to annotate the function in the original benchmark *as* the type that was provided by the user.

Baselines. We acknowledge the importance of comparing our approach against a baseline. However, as discussed in section 4.6, symbolic methods for program generation have historically faced significant challenges in this domain. Recent advances in Machine Learning (ML) and large language models (LLMs) have enabled substantial improvements, making this problem more tractable. A purely random generation approach would be of limited utility given the immense size of the search space; the refinement type space is considerably larger and more complex than the space of standard Haskell types. Consequently, there aren't any meaningful baselines in the existing literature for generating refinement types and verifying entire codebases effectively until now.

4.5.1 RQ1: Single type prediction accuracy

Table 4.1 presents the cumulative results on the LHTUTORIAL. We have manually categorized the 68 programs with single target functions that need to be annotated with a refinement type as *Easy*, *Medium* and *Hard*, based on the complexity of the ground truth refinement type. For each target function we generate 50 refinement types using the pre-trained LLMs and their fine-tuned versions. We also show the number of functions that the LLMs successfully verified

Table 4.1: LHTUTORIAL results: 68 total single type benchmark, where we divided the user-intended type into 3 difficulty categories. We also present the *pass@k* metrics for the full benchmark.

LLM	*Total* (68)	*Easy* (17)	*Medium* (30)	*Hard* (21)	*pass@1*	*pass@5*	*pass@10*	*pass@20*	*pass@50*
STARCODER-3B	37	14	16	7	12.91%	30.19%	38.30%	45.82%	54.41%
+ FINE-TUNED	62	17	28	17	65.88%	82.25%	85.22%	87.63%	91.18%
CODELLAMA-7B	41	15	20	6	11.68%	32.01%	42.01%	50.85%	60.29%
+ FINE-TUNED	60	15	27	18	47.97%	71.76%	78.67%	83.62%	88.24%
STARCODER-15B	42	16	19	7	18.79%	41.67%	50.21%	56.78%	61.76%
+ FINE-TUNED	62	16	28	18	57.65%	78.79%	84.26%	88.19%	91.18%

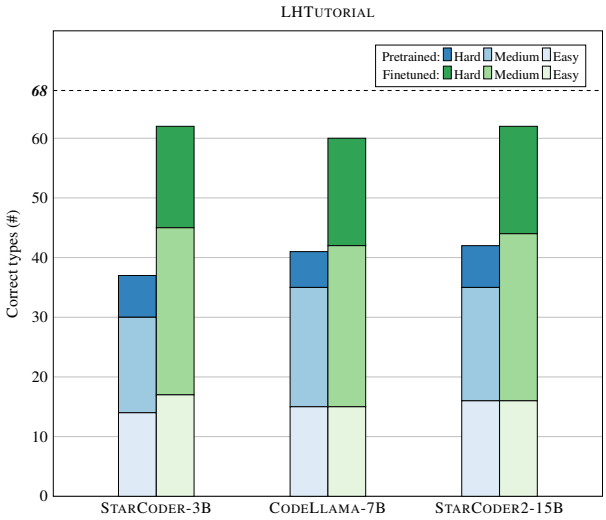


Figure 4.10: Pretrained and fine-tuned LLM accuracy in generating single refinement types for the LHTUTORIAL benchmark.

at Figure 4.10. We observe that all models showcase great performance for the Easy types, with slight improvements when fine-tuned. However, we observe great improvement for all LLMs for the Medium and Hard categories. Specifically, STARCODER-3B shows a 12-program improvement at the Medium category, while CODELLAMA-7B and STARCODER2-15B show a 7- and 9-program improvement respectively. We see a similar improvement for the Hard programs with an increase of 10, 12 and 11 programs respectively.

We also present the *pass@k* results in Table 4.1. [62] introduced the *pass@k* metric, and the Codex paper [17] popularized it recently, specifically for evaluating code generation LLMs. To calculate *pass@k*, *k* code samples are generated per problem and the problem is considered solved if any sample is correct, where *pass@k* is the total fraction of problems solved. However, as it has been observed in [17], computing *pass@k* this way can have high variance. Instead, to

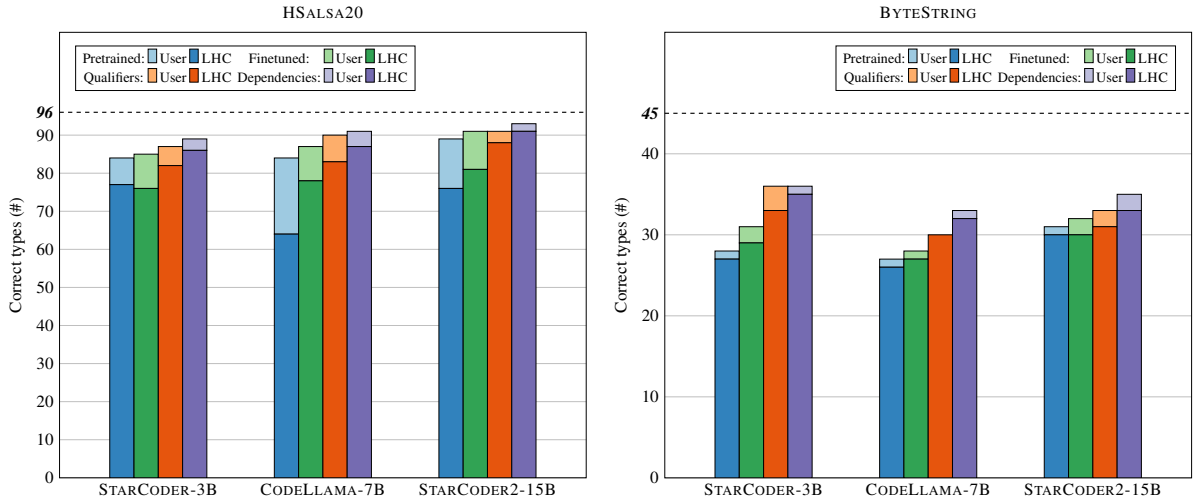


Figure 4.11: LHC accuracy in generating and verifying refinement types for our Haskell benchmarks.

evaluate *pass@k*, $n \geq k$ samples per task are generated (in this paper, we use $n = 50$ and $k \leq 50$), and the number of correct samples $c \leq n$ is counted in order to calculate the unbiased estimator:

$$pass@k = \mathop{\mathbb{E}}_{problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

pass@50 here represents the total *accuracy* of each LLM for the LHTUTORIAL, *i.e.* the percentage of the target functions that the LLMs generated at least one correct refinement type.

By fine-tuning, we observe great improvement for all *pass@k* metrics, reaching a *pass@10* of 85% and a *pass@50* of 91% in some cases. We also see that there isn't a big improvement for $k > 10$ samples, especially for the fine-tuned models, indicating that sampling even 10 refinement types per function can yield very accurate results with a fine-tuned LLM.

Fine-tuned LLMs — even with a small number of trainable parameters — learn to encode LIQUIDHASKELL programs and can generate correct refinement types for *91% of the target functions*, an improvement of up to *35%* from the out-of-the-box models.

Table 4.2: HSALSA20 verification results (*96 functions*)

LLM	*Automatically Verified*	*Correct Types Unverified*	LHC *Iterations*	*Verification Time (mins)*
STARCODER-3B	77	+7	562	257
+ FINE-TUNED	76	+9	583	227
+ QUALIFIERS	82	+5	400	295
+ DEPENDENCIES	**86**	**+3**	**337**	**238**
CODELLAMA-7B	64	+20	760	246
+ FINE-TUNED	78	+9	485	252
+ QUALIFIERS	83	+7	365	239
+ DEPENDENCIES	**87**	**+4**	**334**	**201**
STARCODER-15B	76	+13	496	351
+ FINE-TUNED	81	+10	472	336
+ QUALIFIERS	88	+3	314	358
+ DEPENDENCIES	**91**	**+2**	**258**	**254**

4.5.2 RQ2: Whole codebase precision

Next, we evaluate LHC on the two Haskell codebases HSALSA20 and BYTESTRING. We evaluate here *four* different approaches with each LLM, presented in Table 4.2 for HSALSA20 and Table 4.3 for BYTESTRING. First, we use the *original* pretrained LLMs as a backend for LHC in order to generate the various refinement types. Second, we *fine-tune* the models as described before. Next, we enable the QUALIFIERS verification optimization, an optimization which automatically extracts *all qualifiers* from the LLM predicted types and adds corresponding pragmas in the context of the program, as described in § 4.3.4. This optimization also adds the relevant wildcard type as a last candidate type, that is tested when all LLM-predicted types are exhausted, which tells LIQUIDHASKELL to perform symbolic type inference using the qualifiers. Finally, we enable the DEPENDENCIES prompt optimization for additional context. This optimization uses few-shot prompting to add the verified dependency functions to the LLM prompts (§ 4.3.2).

Automatically Verified. This metric, in the first column of Table 4.2 and Table 4.3, indicates the number of functions that were automatically annotated and verified by LHC without human

Table 4.3: BYTESTRING verification results (*45 functions*)

LLM	*Automatically Verified*	*Correct Types Unverified*	LHC *Iterations*	*Verification Time (mins)*
STARCODER-3B	27	**+1**	95	73
+ FINE-TUNED	29	+2	104	52
+ QUALIFIERS	33	+3	99	49
+ DEPENDENCIES	**35**	**+1**	**93**	**42**
CODELLAMA-7B	26	+1	114	87
+ FINE-TUNED	27	+1	**110**	84
+ QUALIFIERS	30	**+0**	137	90
+ DEPENDENCIES	**32**	+1	114	**80**
STARCODER-15B	30	**+1**	95	115
+ FINE-TUNED	30	+2	**88**	**110**
+ QUALIFIERS	31	+2	101	121
+ DEPENDENCIES	**33**	+2	100	111

intervention. For example, STARCODER-3B in its fine-tuned version correctly inferred types for 76 functions in HSALSA20 and verifies up to 86 functions when we include the QUALIFIERS and DEPENDENCIES optimizations. CODELLAMA-7B shows similar improvement. However, the larger STARCODER2-15B reaches up to 91 functions that are correctly annotated by LHC.

Correct Unverified Type Predictions. This metric shows the number of times that the LLMs generated a *correct* refinement type for a function, but LHC *ran out of fuel* (as discussed in § 4.4.2) and the function was not automatically (and locally) verified by LHC — thus *not included* in the previous metric. In a real-world setting, however, before asking the user to manually write a refinement for these unsuccessful functions, as a preliminary step we could provide the list of LLM-predicted types and have the *user select or approve* one in order to mitigate the manual effort. While a higher number for this metric indicates less dependency on manual input, it still represents wasted cycles for LHC, where it couldn't arrive to the correct combination of refinement types in order to locally verify the faulty ones. We observe that all LLMs have relatively high numbers when QUALIFIERS and DEPENDENCIES were not used. When we use both optimizations, the numbers go as low as only 2 for STARCODER2-15B, since LHC was able

to automatically verify the vast majority of the functions when using this LLM, as we showed earlier.

Cumulative results. Figure 4.11 also shows the cumulative results of the previous two metrics, *i.e.* the total number of functions LHC was able to *generate a correct type* for and were either automatically or manually verified. We observe that for HSALSA20 even without the last two optimizations, LHC generates a significant fraction — nearly 80% — of correct types, but the optimizations nevertheless improve the accuracy of types automatically generated by LHC to more than 90%. However, for the more complicated module BYTESTRING though, we observe that without the QUALIFIERS and DEPENDENCIES optimizations, we can only generate 60% of the types, and the addition of symbolic qualifier inference and context provides a significant improvement, allowing LHC to generate correct types for up to 77% of the functions.

LHC can automatically generate formally verified types for up to *94%* of a codebase's functions.

4.5.3 RQ3: Efficiency

The third and fourth columns of Table 4.2 and Table 4.3 summarize how *efficiently* LHC can verify codebases.

LHC Iterations. This metric counts the number of iterations that LHC needs to verify the full module. Fewer iterations suggest a more efficient verification process. We observe that, for HSALSA20 that has more functions to verify and deeper dependencies, there is a significant improvement in the time we spent verifying them when we use QUALIFIERS and DEPENDENCIES. On the other hand, for BYTESTRING, we observe a slight overhead, for unclear reasons, but perhaps due increasing the size of the prompts and getting diminishing results from the extra

context in terms of efficiency.

Verification Time. The last column in the tables Table 4.2 and Table 4.3 provides an overview of the total time in minutes that LHC spent in fully verifying the modules. At a high-level, it is remarkable that LHC is able to annotate and verify entire codebases in 1-4 hours, as typically this work takes days or weeks for a human to do. (Of course, this comes with the caveat that with these benchmarks we know that suitable types exist, and we emulated human assistance when the LLM got stuck). The results also indicate that the verification time varies significantly depending on the LLM used and the specific optimizations applied. For instance, in the HSALSA20 results, we see that the baseline STARCODER-3B model required 257 minutes for verification, which was reduced to 227 minutes with fine-tuning, and further adjusted to 295 minutes with qualifiers, but significantly drops to 238 minutes with dependency enhancements. Similarly, CODELLAMA-7B and STARCODER2-15B models show notable reductions in verification time when dependencies are included. In the BYTESTRING results, the trend is consistent for STARCODER-3B and CODELLAMA-7B. However, STARCODER2-15B shows no significant improvement.

LHC can, with modest emulated human assistance, annotate and verify entire codebases in a few hours. The fine-tuning, QUALIFIERS and DEPENDENCIES optimizations greatly enhance verification efficiency by reducing verification time by an average *18%* (and up to *42%*) across real-world codebases.

4.5.4 Threats to Validity

We note three threats to the validity of our results. First, we have only considered three codebases: the LHTUTORIAL, HSALSA20, and BYTESTRING. It is entirely possible that larger or more complex codebases may require annotations that cannot be generated so effectively by LLMs. Second, our approach currently doesn't support *mutually recursive functions*. A

potential solution to this is to *break the cycles* created by these mutually recursive functions by either (1) requiring the programmer to specify a type for one of the functions of the cycle, or (2) speculatively breaking the cycle and letting our backtracking mechanism synthesize the types. However, we have not tried either of these approached as mutually recursive functions don't occur in our benchmarks. Third, we have emulated human assistance in our evaluation, meaning on our benchmarks we know *a priori* that suitable refinement type annotations exist. In a more realistic scenario using LHC on a new codebase, such types may not exist because bugs in the code may require it to be modified, or because of limitations in the verifier (LIQUIDHASKELL) itself. We defer the evaluation of LHC on new codebases with actual users to future work (but note that this is challenging as realistically, annotation requires days or weeks of human effort).

4.6 Related Work

Finally, we discuss some related lines of work on using machine learning and LLMs to automate program verification.

Generating Proof Annotations. LHC is most closely related to several other neurosymbolic approaches to generating the annotations needed for formal verification. CODE2INV [113], [60], and Pei et al. [90] present techniques to synthesize loop invariants [32] using neural networks and fine-tuned LLMs, respectively. Kamath et al. [52] and [132] integrates LLMs natively with automated reasoners — ESCJava [30] and ESBMC [34] — to additionally check whether the generated invariants are actually inductive and, optionally, further to repair the invariants by querying the LLMs and reducing the proposed invariants into an inductive set using the HOUDINI algorithm [31]. Several studies have also explored the generation of annotations and formal postconditions using advanced techniques. Similarly, NL2POSTCOND [26] investigates the transformation of natural language intent into formal method postconditions using LLMs, proposing metrics for assessing the quality of these transformations. Finally, LAUREL [82] automatically generates helper assertions for proofs written in DAFNY by leveraging LLMs as well. All the above focus on a single goal: generating loop invariants, or contracts for single functions in isolation. In contrast, LHC is an *interprocedural* method that aims to generate interdependent refinement type annotations (which generalize invariants, pre- and post-conditions) across the whole codebase.

Generating Code from Specifications. A different line of work looks at using LLMs to synthesizing *code* from formal specifications in proof-oriented languages. Misu et al. [81] and CLOVER [118] use LLMs to synthesize verified DAFNY code corresponding to natural language and formal specifications. Similarly, Chakraborty et al. [15] investigate using LLMs to synthesize F* programs (and proofs) from dependent type specifications. In contrast to the above, LHC's

goal is not to generate code, but only the refinement contract annotations needed for modular verification of existing code(bases).

LLMs for Proof Generation. Several groups have looked into using machine learning techniques to automate proofs written in tactic-based interactive theorem provers. Sanchez et al. [108] uses machine learning techniques to generate COQ proofs. BALDUR [29] explores the use of LLMs to generate entire Isabelle/HOL proofs for program verification, a departure from traditional proof assistants that generate one proof statement at a time. Complementing this, Yang et al. [136] introduces LEANDOJO, a tool that combines LLMs with retrieval-augmented mechanisms to enhance theorem proving in the LEAN environment, demonstrating improved proof generation capabilities. Wu et al. [134] evaluates the performance of LLMs in autoformalization in Isabelle/HOL, introducing a neural theorem prover trained on autoformalized statements. Unlike the above, LHC is designed to work in the setting of SMT-based "auto-active" verification, where the only programmer input is the code and the type specification; the rest is handled by the SMT solver.

In-Context Prompting. In-context prompting techniques have been explored to enhance the few-shot learning capabilities of LLMs. Liu et al. [72] investigates the relationship between in-context example selection and GPT-3's few-shot performance, introducing a retrieval model for better example selection. Su et al. [117] proposes a framework for selective annotation to improve accuracy in in-context learning scenarios. Lu et al. [76] demonstrates the importance of prompt order, using model-generated sequences to find optimal prompts, while Sorensen et al. [116] introduces an information-theoretic approach to prompt engineering, maximizing mutual information to select the best prompts without relying on model weights or ground truth labels. LAUREL [82] introduced a lemma similarity metric to import potentially related lemmas to the current proof. These lines of work inspired our DEPENDENCIES optimization, where we augment our prompts with additional context. Unlike the above, LHC does not rely on similarity heuristics

but instead, it includes the function dependencies of the target function we are trying to locally verify.

4.7 Acknowledgements

Chapter 4, in part, is a reprint of the material as it will appear in the Proceedings of the 2025 IEEE/ACM 46th International Conference on Software Engineering. Georgios Sakkas, Pratyush Sahu, Kyeling Ong, Ranjit Jhala, ICSE 2025. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This dissertation has explored neurosymbolic approaches to enhancing the programming and debugging experience by developing tools that leverage both symbolic reasoning and machine learning techniques. Through the introduction of three core systems — **RITE**, **SEQ2PARSE**, and **LHC** — we have demonstrated the effectiveness of neurosymbolic methods in addressing diverse challenges in automated program repair and verification across different programming languages. RITE uses analytic program repair to provide meaningful type error feedback in OCaml, demonstrating that similar errors can often be solved by applying similar repairs. Meanwhile, SEQ2PARSE tackles syntax errors in Python by blending neural classifiers with error-correcting parsers, successfully addressing challenges associated with selecting relevant error-correction rules. Finally, LHC introduces a neurosymbolic approach to refinement type inference in Haskell, reducing the labor-intensive process of manual annotations required for verification.

Each tool has been evaluated for its accuracy, efficiency, and usability, showing promising results across large datasets and user studies. These tools not only improve error localization and repair accuracy but also offer developers insights into the nature of their errors. By employing

machine learning alongside traditional programming language methods, we aim to minimize the need for human intervention in the debugging and verification processes, significantly streamlining software development workflows.

5.2 Future Work

Future research could broaden the applicability of neurosymbolic methods to a wider range of programming languages, error types, and real-world software development scenarios. While this dissertation focuses on OCaml, Python, and Haskell, expanding support to languages like Java, C++, and Rust, which are widely used in industry, would enable these tools to address a broader array of software systems. For instance, extending LHC to infer refinement types in Rust could enhance its safety guarantees through automated verification of ownership and borrowing rules, thus reducing memory safety vulnerabilities. Similarly, adapting SEQ2PARSE for JavaScript could mitigate the challenges posed by its dynamically typed nature, where syntax errors can manifest at runtime environments too.

Another promising direction is to integrate neurosymbolic techniques into more complex environments such as *distributed systems*, *concurrent programming*, and *embedded systems*. These domains present unique challenges, including race conditions, deadlocks, and resource constraints, which traditional debugging tools often struggle to address. For example, a neurosymbolic tool could leverage neural models to detect potential concurrency issues and symbolic reasoning to validate fixes, thus enabling developers to manage complex parallel execution patterns effectively.

In the context of *tokenization* and *model pre-training*, our exploration with SEQ2PARSE revealed the potential of tailoring token sequences for specific tasks. Building on this, a future avenue involves developing *custom tokenization strategies* that better represent programming languages. Current tokenizers, like OpenAi's Tiktoken, are largely optimized for natural language processing, particularly English, and may not be well-suited for code-related tasks [91]. For

example, they can be suboptimal for handling nested syntactic structures inherent in programming languages. A specialized tokenizer could, for instance, preserve the hierarchical nature of abstract syntax trees (ASTs), potentially enhancing the performance of Large Language Models (LLMs) in tasks such as *code completion, code summarization*, and *program synthesis*.

Moreover, *semi-supervised learning* and *reinforcement learning* present opportunities to reduce the dependency on large, labeled datasets. Semi-supervised approaches could leverage *self-training* or *consistency regularization* to make use of unlabeled code samples, while reinforcement learning could optimize model behavior based on real-time developer feedback. For example, an interactive debugging assistant could use reinforcement learning to adaptively refine error messages and suggested fixes based on developer responses, leading to more personalized and effective debugging experiences.

Additionally, future work could explore integrating *active learning* pipelines, where models actively query developers for guidance on uncertain cases, improving their performance over time with minimal manual effort. This approach would be especially beneficial in domains where labeled data is scarce or expensive to obtain, such as *domain-specific languages* (DSLs) or *legacy codebases*.

Lastly, there is significant potential for expanding neurosymbolic tools beyond error repair and type inference into broader areas of *program synthesis*, *code optimization*, and *adaptive learning environments* for novice programmers. For example, a neurosymbolic system could dynamically adjust the difficulty of coding exercises based on a student's progress, providing personalized feedback and targeted hints. This capability could revolutionize programming education, making it more accessible and effective for learners with diverse backgrounds and skill levels.

By pursuing these future directions, the neurosymbolic approaches developed in this dissertation could significantly advance the state of automated software development tools, reducing the cognitive load on developers, improving software reliability, and making programming more

accessible across a diverse range of applications and user groups.

Bibliography

[1] Alireza Ahadi, Raymond Lister, Shahil Lal, and Arto Hellas. Learning programming, syntax errors and institution-specific factors. In *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18, pages 90–96, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Toufique Ahmed, Premkumar Devanbu, and Vincent J Hellendoorn. Learning lenient parsing & typing via indirect supervision. *Empirical Software Engineering*, 26(2), mar 2021.

[3] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: for the student programs, from the student programs. In *International Conference on Software Engineering: Software Engineering Education and Training*, pages 78–87, 2018.

[4] A. V. Aho and S. C. Johnson. Lr parsing. *ACM Comput. Surv.*, 6(2):99–124, jun 1974.

[5] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, 1:305–312, 1972.

[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs, 2017.

[7] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, Oct 2019.

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2015.

[9] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks, 2016.

[10] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network, 2016.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*, pages 209–210. Springer-Verlag, Berlin, Heidelberg, 2006.

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

[13] Michael G. Burke and Gerald A. Fisher. A practical method for lr and ll syntactic error diagnosis and recovery. *ACM Trans. Program. Lang. Syst.*, 9(2):164–197, mar 1987.

[14] Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. *ACM SIGPLAN Notices*, 48(10):33–52, 2013.

[15] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. Towards neural synthesis for smt-assisted proof-oriented programming, 2024.

[16] Nigel P Chapman. *LR Parsing: Theory and Practice*. Cambridge University Press, New York, NY, USA, 1987.

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

[18] Sheng Chen and Martin Erwig. Counter-factual typing for debugging type errors. In *Principles of Programming Languages*, POPL '14, pages 583–594, New York, NY, USA, 2014. ACM.

[19] Michael Collins. Probabilistic context-free grammars (pcfgs). *Lecture Notes*, 2013.

[20] Rafael Corchuelo, José A. Pérez, Antonio Ruiz, and Miguel Toro. Repairing syntax errors in lr parsers. *ACM Trans. Program. Lang. Syst.*, 24(6):698–710, nov 2002.

[21] Benjamin Cosman, Madeline Endres, Georgios Sakkas, Leon Medvinsky, Yao-Yuan Yang, Ranjit Jhala, Kamalika Chaudhuri, and Westley Weimer. *PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization*, pages 1047–1053. Association for Computing Machinery, New York, NY, USA, 2020.

[22] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 75–80, New York, NY, USA, 2012. Association for Computing Machinery.

[23] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[24] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Laerning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, 2020.

[25] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970.

[26] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proc. ACM Softw. Eng.*, 1(FSE), 2024.

[27] Madeline Endres, Georgios Sakkas, Benjamin Cosman, Ranjit Jhala, and Westley Weimer. Infix: Automatically repairing novice program inputs. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, pages 399–410. IEEE Press, 2019.

[28] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235 – 271, 1992.

[29] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1229–1241, New York, NY, USA, 2023. Association for Computing Machinery.

[30] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[31] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, page 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.

[32] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[33] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

[34] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. Esbmc 5.0: an industrial-strength c model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 888–891, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. Refinedrust: A type system for high-assurance verification of rust programs. *Proc. ACM Program. Lang.*, 8(PLDI):1115–1139, 2024.

[36] Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. Usability-oriented design of liquid types for java. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1520–1532. IEEE, 2023.

[37] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, pages 180–184. MIT Press, 2016. `http://www.deeplearningbook.org`.

[38] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[39] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *Programming Language Design and Implementation*, 2018.

[40] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. Association for Computing Machinery.

[41] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017.

[42] Christian Haack and J B Wells. Type error slicing in implicitly typed Higher-Order languages. In *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 284–301. Springer Berlin Heidelberg, 7 April 2003.

[43] Jad Hamza, Nicolas Voirol, and Viktor Kuncak. System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA):166:1–166:30, 2019.

[44] Momchil Hardalov, Ivan Koychev, and Preslav Nakov. Towards automated customer support. In *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 48–59. Springer, 2018.

[45] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer New York, 2009.

[46] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Learning @ Scale*, pages 89–98, 2017.

[47] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012.

[48] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[49] Lakshmi S Iyer, Babita Gupta, and Nakul Johri. Performance, scalability and reliability issues in web applications. *Industrial Management & Data Systems*, 105(5):561–576, 2005.

[50] Frederick Jelinek, John D Lafferty, and Robert L Mercer. Basic methods of probabilistic context free grammars. In *Speech Recognition and Understanding*, pages 345–360. Springer, 1992.

[51] Nan Jiang, Thibaud Lutellier, and Lin Tan. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, may 2021.

[52] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models, 2023.

[53] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, 2009.

[54] Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. Refinement types for ruby. *CoRR*, abs/1711.09281, 2017.

[55] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 296–309, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, pages 802–811, 2013.

[57] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush. Structured attention networks. *CoRR*, abs/1702.00887, 2017.

[58] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 22 December 2014.

[59] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[60] Naoki Kobayashi, Taro Sekiyama, Issei Sato, and Hiroshi Unno. Toward neural-network-guided program synthesis and verification, 2021.

[61] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *International Symposium on Software Testing and Analysis*, pages 165–176. ACM, 2016.

[62] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[63] Sarah K. Kummerfeld and Judy Kay. The neglected battle fields of syntax errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ACE '03, pages 105–111, AUS, 2003. Australian Computer Society, Inc.

[64] Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. volume 52, pages 219–234, 01 2011.

[65] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[66] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[67] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[68] Eelco Lempsink. Generic type-safe diff and patch for families of datatypes. Master's thesis, Universiteit Utrecht, 2009.

[69] Benjamin S Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *Programming Language Design and Implementation*, pages 425–434. ACM, 2007.

[70] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you, 2023.

[71] Henry H Liu. *Software performance and scalability: a quantitative approach*. John Wiley & Sons, 2011.

[72] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3?, 2021.

[73] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. A practical framework for type inference error explanation. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 781–799, 19 October 2016.

[74] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 298–312, New York, NY, USA, 2016. Association for Computing Machinery.

[75] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes,

Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024.

[76] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity, 2022.

[77] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 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, pages 101–114, New York, NY, USA, 2020. Association for Computing Machinery.

[78] Matias Martinez, Laurence Duchien, and Martin Monperrus. Automatically extracting instances of code change patterns with AST analysis. In *2013 IEEE international conference on software maintenance*, pages 388–391. IEEE, 2013.

[79] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[80] Philippe McLean and R. Nigel Horspool. A faster earley parser. In *CC*, 1996.

[81] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards ai-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering*, 1(FSE):812–835, July 2024.

[82] Eric Mugnier, Emmanuel Anaya Gonzalez, Ranjit Jhala, Nadia Polikarpova, and Yuanyuan Zhou. Laurel: Generating dafny assertions using large language models, 2024.

[83] Jonathan P. Munson and Elizabeth A. Schilling. Analyzing novice programmers' response to compiler error messages. *J. Comput. Sci. Coll.*, 31(3):53–61, January 2016.

[84] Eugene W Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[85] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning*, pages 807–814, 2010.

[86] C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980.

[87] Michael A Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[88] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis*, pages 199–209. ACM, 2011.

[89] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. In *Object Oriented Programming Systems Languages & Applications*, pages 525–542. ACM, 2014.

[90] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[91] Aleksandar Petrov, Emanuele La Malfa, Philip H. S. Torr, and Adel Bibi. Language model tokenizers introduce unfairness between languages, 2023.

[92] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs, 2016.

[93] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: verifying systems C code with separation-logic refinement types. *Proc. ACM Program. Lang.*, 7(POPL):1–32, 2023.

[94] Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, 18(1), oct 2017.

[95] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. Skalpel: A type error slicer for standard ML. *Electron. Notes Theor. Comput. Sci.*, 312:197–213, 24 April 2015.

[96] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. Multi-modal program inference: A marriage of pre-trained language models and component-based synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[97] Sanguthevar Rajasekaran and Marius Nicolae. An error correcting parser for context free grammars that takes less than cubic time, 2014.

[98] Niki Vazou Ranjit Jhala, Eric Seidel. Programming with refinement types, 2014.

[99] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. Learning quick fixes from code repositories, 2018.

[100] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.

[101] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 159–169, New York, NY, USA, 2008. Association for Computing Machinery.

[102] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[103] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[104] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. Harnessing evolution for multi-hunk program repair. In *International Conference on Software Engineering*, pages 13–24, 2019.

[105] George Sakkas, Ranjit Jhala, Westley Weimer, and Madeline Endres. gsakkas/seq2parse: Oopsla2 2022 code release, September 2022.

[106] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 16–30, New York, NY, USA, 2020. Association for Computing Machinery.

[107] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: automating the foundational verification of C code with refined ownership types. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 158–174. ACM, 2021.

[108] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2020, page 1–10, New York, NY, USA, 2020. Association for Computing Machinery.

[109] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan 2015.

[110] Eric L Seidel and Ranjit Jhala. A Collection of Novice Interactions with the OCaml Top-Level System, June 2017.

[111] Eric L Seidel, Ranjit Jhala, and Westley Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *International Conference on Functional Programming*, pages 228–242, 2016.

[112] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. Learning to blame: Localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.*, 1(OOPSLA):60:1–60:27, October 2017.

[113] X. Si, A. Naik, H. Dai, M. Naik, and L. Song. Code2inv: A deep learning framework for program verification. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2020.

[114] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *Acm Sigplan Notices*, 48(6):15–26, 2013.

[115] Dowon Song, Myungho Lee, and Hakjoo Oh. Automatic and scalable detection of logical errors in functional programming assignments. *Proc. ACM Program. Lang.*, 3(OOPSLA):188:1–188:30, October 2019.

[116] Taylor Sorensen, Joshua Robinson, Christopher Rytting, Alexander Shaw, Kyle Rogers, Alexia Delorey, Mahmoud Khalil, Nancy Fulda, and David Wingate. An information-theoretic approach to prompt engineering without ground truth labels. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2022.

[117] Hongjin Su, Jungo Kasai, Chen Henry Wu, Weijia Shi, Tianlu Wang, Jiayi Xin, Rui Zhang, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Selective annotation makes language models better few-shot learners, 2022.

[118] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation, 2024.

[119] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[120] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in f. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016 ; Conference date: 20-01-2016 Through 22-01-2016.

[121] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. Soltype: refinement types for arithmetic overflow in solidity. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[122] Richard A. Thompson. Language correction using probabilistic grammars. *IEEE Transactions on Computers*, C-25(3):275–286, 1976.

[123] Frank Tip and T B Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, 10(1):5–55, January 2001.

[124] P. van der Spek, N. Plat, and C. Pronk. Syntax error repair for a java-based parser generator. *SIGPLAN Not.*, 40(4):47–50, April 2005.

[125] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc., 2017.

[126] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 269–282, New York, NY, USA, 2014. Association for Computing Machinery.

[127] P. Vekris, B. Cosman, and R. Jhala. Refinement types for typescript. In *PLDI*, 2016.

[128] Gust Verbruggen, Vu Le, and Sumit Gulwani. Semantic programming by example with pre-trained models. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[129] Mitchell Wand. Finding the source of type errors. In *Principles of Programming Languages*, pages 38–43, 1986.

[130] Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *Programming Language Design and Implementation*, pages 481–495, 2018.

[131] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[132] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification, 2024.

[133] Liwei Wu, Fei Li, Youhua Wu, and Tao Zheng. *GGF: A Graph-Based Method for Programming Language Syntax Error Correction*, page 139–148. Association for Computing Machinery, New York, NY, USA, 2020.

[134] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models, 2022.

[135] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.

[136] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models, 2023.

[137] Danfeng Zhang and Andrew C Myers. Toward general diagnosis of static errors. In *Principles of Programming Languages*, pages 569–581, 2014.